Pre-Ordered Metric Spaces for Program Improvement

JENNIFER HACKETT, University of Nottingham
GRAHAM HUTTON, University of Nottingham

Improvement theory allows inequational proofs of program efficiency to be written in a similar manner to equational proofs of program correctness. However, existing theories of improvement are operational in nature, in contrast to the denotational approaches that tend to be used for correctness. Moreover, they are tied to specific languages and cost models, with each such choice requiring a new theory, and while these theories have commonalities, it is difficult to formalise the ways in which they are linked. 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 that is generic and intuitive, and resolves previously problematic issues in the area of space efficiency. We show how our theory can be used both to establish generic improvement results, and to reason about specific programming examples.

ACM Reference Format:
Jennifer Hackett and Graham Hutton. 2020. Pre-Ordered Metric Spaces for Program Improvement. 1, 1 (March 2020), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, when we want to run our programs, what they *do* matters. 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 traditional 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 *abstract* away from the details of the different operational theories by means of a unified *denotational* framework. 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], which 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. The article makes the following contributions:

Authors' addresses: Jennifer HackettUniversity of Nottingham, jennifer.hackett@nottingham.ac.uk; Graham HuttonUniversity of Nottingham, graham.hutton@nottingham.ac.uk.

2020. XXXX-XXXX/2020/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- 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 on a range of examples, including a semantics that mirrors the operational nature of improvement theory in a denotational manner.

To make the ideas accessible to a broad audience, 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 range of practical examples of how our generic framework can be instantiated to produce improvement theories for specific semantic models. We also prove efficiency properties of programs in these models, starting with a theory that closely follows the operational character of improvement theory. Section 8 surveys 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 for reasoning about intensional properties of programs, most importantly their resource usage. We say a program term M is "improved by" another term N, written $M \gtrsim N$, if N is "better" (in some sense) than M in any program context. Formally, we define the relation $\gtrsim$ as follows:

$$M \gtrsim N \quad \Leftrightarrow \quad \forall \mathbb{C}.\ \mathbb{C}[M] \Downarrow^{n} \ \Rightarrow\ \mathbb{C}[N] \Downarrow^{\leq n}$$

That is, "for any context $\mathbb{C}$, if $\mathbb{C}[M]$ terminates with some cost n, then $\mathbb{C}[N]$ will terminate with cost at most n". The 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 $M \lesssim\!\!\!\gtrsim N$, defined simply by $M \gtrsim N \wedge N \gtrsim M$.

The original motivation for improvement theory was 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, we usually have an operation that represents one unit of cost, together with rules for manipulating these costs. For example, in the call-by-need time theory of Moran and Sands [1999a], the *tick* operation adds one unit of time cost to a term, written $\checkmark$. The rules for manipulating ticks are then called *tick algebra*. For example, Figure 1 illustrates a number of rules that are valid in Moran and Sands' tick algebra. (Note that in these rules V denotes a term in value form, c_i is a constructor of arity $a(i)$, and all variables are assumed to be distinct.)

$$\textbf{let } x = V \textbf{ in } \mathbb{C}[x] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \textbf{let } x = V \textbf{ in } \mathbb{C}[\checkmark V] \tag{value-β}$$

$$\checkmark \textbf{case } M \textbf{ of } \{pat_i \to N_i\} \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \textbf{case } M \textbf{ of } \{pat_i \to \checkmark N_i\} \tag{case-$\checkmark$}$$

$$\begin{aligned}&\textbf{case } (\textbf{case } M \textbf{ of } \{pat_i \to N_i\}) \textbf{ of } \{pat_j \to L_j\} \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \\ &\quad \textbf{case } M \textbf{ of } \{pat_i \to \textbf{case } N_i \textbf{ of } \{pat_j \to L_j\}\}\end{aligned} \tag{case-case}$$

$$\checkmark M \mathrel{\underset{\sim}{\triangleright}} M \tag{$\checkmark$-elim}$$

$$\begin{aligned}&\textbf{case } c_i\ x_1\ x_2 \ldots x_{a(i)} \textbf{ of } \{\ldots; c_i\ y_1\ y_2 \ldots y_{a(i)} \to M; \ldots\} \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \\ &\qquad\qquad M[x_1/y_1, x_2/y_2, \ldots, x_{a(i)}/y_{a(i)}]\end{aligned} \tag{case-cons}$$

$$\textbf{case } \checkmark M \textbf{ of } \{pat_i \to \textbf{case } N_i\} \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \checkmark \textbf{case } M \textbf{ of } \{pat_i \to \textbf{case } N_i\} \tag{$\checkmark$-float}$$

Fig. 1. Rules from Moran and Sands' tick algebra.

2.1 Improvement Induction

One particularly useful proof rule that appears in a number of improvement theories is *improvement induction*. This technique 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:

$$\frac{M \mathrel{\underset{\sim}{\triangleright}} \checkmark \mathbb{C}[M] \qquad \checkmark \mathbb{C}[N] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} N}{M \mathrel{\underset{\sim}{\triangleright}} N}$$

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:

$$\begin{aligned}(+\!\!+) = \lambda xs\ ys \to \textbf{case } xs \textbf{ of}&\\ [\,] \quad\quad\ &\to ys\\ (x : xs') &\to x : (xs' +\!\!+ ys)\end{aligned}$$

Note that the definition takes the form of a lambda abstraction, and so is in value form. Using this 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 because $+\!\!+$ 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 and call-by-need tick algebra. To verify the improvement $(xs +\!\!+ ys) +\!\!+ zs \mathrel{\underset{\sim}{\triangleright}} xs +\!\!+ (ys +\!\!+ zs)$, we first define the following context, in which $[-]$ denotes the hole where a term will be substituted:

$$\begin{aligned}\mathbb{C} = \textbf{case } xs \textbf{ of}&\\ [\,] \quad\ &\to ys +\!\!+ zs\\ x : xs &\to x : [-]\end{aligned}$$

Expanding the definition of $+\!\!+$ using value-β will immediately give us that $xs +\!\!+ (ys +\!\!+ zs)$ is cost equivalent to $\checkmark\mathbb{C}[xs +\!\!+ (ys +\!\!+ zs)]$. Hence, by improvement induction, it suffices to show that $(xs +\!\!+ ys) +\!\!+ zs \mathrel{\underset{\sim}{\triangleright}} \checkmark\mathbb{C}[(xs +\!\!+ ys) +\!\!+ zs]$, which can itself be verified by the following calculation in tick algebra. For clarity, we highlight the ticks being manipulated in each step.

$$
\begin{array}{ll}
& (xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \{\text{value-}\beta \text{ for append}\} \\
& \checkmark\mathbf{case}\ xs \mathbin{+\!\!+} ys\ \mathbf{of} \\
& \qquad [\,] \quad \rightarrow zs \\
& \qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \{\text{value-}\beta \text{ for append}\} \\
& \checkmark\mathbf{case}\ \checkmark\mathbf{case}\ xs\ \mathbf{of} \\
& \qquad\qquad\qquad [\,] \quad \rightarrow ys \\
& \qquad\qquad\qquad x : xs \rightarrow x : (xs \mathbin{+\!\!+} ys) \\
& \quad \mathbf{of} \\
& \qquad [\,] \quad \rightarrow zs \\
& \qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \{\text{case-}\checkmark\} \\
& \checkmark\mathbf{case}\ \mathbf{case}\ xs\ \mathbf{of} \\
& \qquad\qquad [\,] \quad \rightarrow \checkmark ys \\
& \qquad\qquad x : xs \rightarrow \checkmark x : (xs \mathbin{+\!\!+} ys) \\
& \quad \mathbf{of} \\
& \qquad [\,] \quad \rightarrow zs \\
& \qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \{\text{case-case}\} \\
& \checkmark\mathbf{case}\ xs\ \mathbf{of} \\
& \qquad [\,] \quad \rightarrow \mathbf{case}\ \checkmark ys\ \mathbf{of} \\
& \qquad\qquad\qquad [\,] \quad \rightarrow zs \\
& \qquad\qquad\qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
& \qquad x : xs \rightarrow \mathbf{case}\ \checkmark x : (xs \mathbin{+\!\!+} ys)\ \mathbf{of} \\
& \qquad\qquad\qquad [\,] \quad \rightarrow zs \\
& \qquad\qquad\qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
\underset{\sim}{\rhd} & \{\checkmark\text{-elim}\} \\
& \checkmark\mathbf{case}\ xs\ \mathbf{of} \\
& \qquad [\,] \quad \rightarrow \mathbf{case}\ \checkmark ys\ \mathbf{of} \\
& \qquad\qquad\qquad [\,] \quad \rightarrow zs \\
& \qquad\qquad\qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
& \qquad x : xs \rightarrow \mathbf{case}\ x : (xs \mathbin{+\!\!+} ys)\ \mathbf{of} \\
& \qquad\qquad\qquad [\,] \quad \rightarrow zs \\
& \qquad\qquad\qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \{\text{case-cons}\} \\
& \checkmark\mathbf{case}\ xs\ \mathbf{of} \\
& \qquad [\,] \quad \rightarrow \mathbf{case}\ \checkmark ys\ \mathbf{of} \\
& \qquad\qquad\qquad [\,] \quad \rightarrow zs \\
& \qquad\qquad\qquad r : rs \rightarrow r : (rs \mathbin{+\!\!+} zs) \\
& \qquad x : xs \rightarrow x : ((xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs) \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \{\checkmark\text{-float}\} \\
& \checkmark\mathbf{case}\ xs\ \mathbf{of}
\end{array}
$$

$$
\begin{array}{ll}
& \quad\quad \begin{array}{lll} [\,] & \to & \checkmark\, \mathbf{case}\ ys\ \mathbf{of} \\ & & \quad \begin{array}{lll} [\,] & \to & zs \\ r : rs & \to & r : (rs +\!\!+\ zs) \end{array} \\ x : xs & \to & x : ((xs +\!\!+\ ys) +\!\!+\ zs) \end{array} \\
\underset{\sim}{\lhd\!\rhd} & \{\text{value-}\beta \text{ for append}\} \\
& \checkmark\, \mathbf{case}\ xs\ \mathbf{of} \\
& \quad\quad \begin{array}{lll} [\,] & \to & ys +\!\!+\ zs \\ x : xs & \to & x : ((xs +\!\!+\ ys) +\!\!+\ zs) \end{array} \\
\equiv & \\
& \checkmark \mathbb{C}[(xs +\!\!+ ys) +\!\!+\ zs]
\end{array}
$$

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 $M \underset{\sim}{\triangleright} \checkmark\mathbb{C}[M] \underset{\sim}{\triangleright} \checkmark\mathbb{C}[\checkmark\mathbb{C}[M]] \underset{\sim}{\triangleright} \dots$ by repeated application of $\mathbb{C}$, which will tend toward a 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 *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 $d : X \times X \to \mathbb{R}^+$ (where $\mathbb{R}^+$ is the set of non-negative real numbers) satisfying the following three properties [Searcóid 2006]:

$$d(x, y) = 0 \iff x = y \qquad \text{(identity of indiscernibles)}$$

$$d(x, y) = d(y, x) \qquad \text{(symmetry)}$$

$$d(x, z) \leq d(x, y) + d(y, z) \qquad \text{(triangle inequality)}$$

Given this definition, a metric space is simply a pair (X, d) where d is a metric on X. Sometimes we 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 d is clear.

Metric spaces allow us to generalise the usual notion of *limit* to arbitrary sets. A sequence of values $x_1, x_2, x_3 \dots$ 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 a value 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 $f : X \to Y$ 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 $f : X \rightarrow Y$ is a *contraction* if there is a constant $0 < c < 1$ such that for any $x_1, x_2 \in X$, we have $d(f(x_1), f(x_2)) \leq c \cdot d(x_1, x_2)$. 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 by $\mathsf{fix}\ f$.

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

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:

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

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

$$
\begin{array}{ll}
& f(\mathsf{fix}\ f) \\
= & \{\text{definition of } \mathsf{fix}\ f\} \\
& f(\displaystyle\lim_{n \to \infty} x_n) \\
= & \{f \text{ contraction, so must be continuous}\} \\
& \displaystyle\lim_{n \to \infty} f(x_n) \\
= & \{\text{definition of } x_{n+1}\} \\
& \displaystyle\lim_{n \to \infty} x_{n+1} \\
= & \{\text{definition of } \mathsf{fix}\ f\text{, limits}\} \\
& \mathsf{fix}\ f
\end{array}
$$

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

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 distance function in such a metric space-based semantics typically corresponds to some notion of "distinguishability", with programs that are closer together simulating one another more closely or for a longer period of time.

The metric space approach to semantics was first explored by de Bakker and Zucker [1982] for 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 that of *domain theory*, which is based upon lattices or complete partial orders. The domain-theoretic ordering on programs often corresponds to a notion of definedness. In this context, the meaning of a program is the *least* solution to its defining equations (as opposed to the unique solution as with metric spaces), generally the least fixed point of a monotone function. Domain theory was first introduced by Scott [1970]. In certain cases, the domain-theoretic and metric-based approaches to semantics coincide [de Bakker and Meyer 1987].

In lattice-based denotational semantics, there is a powerful technique for reasoning about recursive programs, in the form of the *Knaster-Tarski theorem* [Tarski 1955]:

Theorem 3.2 (Knaster-Tarski theorem). *For any complete lattice $(L, \leq)$ and monotone $f : L \longrightarrow L$, the least fixed and least pre-fixed points of f exist and are equal. Similarly, the greatest fixed and post-fixed points exist and are also equal.*

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

3.3 Pre-Ordered Metric Spaces

We can combine the metric space and domain theoretic approaches to semantics using the idea of a *pre-ordered metric space*. Formally, a pre-ordered metric space is a metric space (X, d) equipped with a reflexive and transitive relation (a pre-order) $\prec$ 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 $x_i \prec y_i$ for all i, we require that the limits share the same relationship, i.e. $\lim x_i \prec \lim y_i$. 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 $f : X \longrightarrow X$, if $f(x) \prec x$ then* $\mathsf{fix}\ f \prec x$. *In other words,* $\mathsf{fix}\ f$ *is the least pre-fixed point of f. Dually,* $\mathsf{fix}\ f$ *is the greatest* post-*fixed point of f.*

Proof. Consider following two infinite sequences:

$$\begin{array}{lllll} x & x & x & x & \ldots \\ x & f(x) & f(f(x)) & f(f(f(x))) & \ldots \end{array}$$

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 $\mathsf{fix}\ f$ by the definition of a pre-ordered metric space. The dual case follows from the above by reversing the pre-order $\prec$. □

A number of different approaches to combining orders and metric spaces have been explored in the literature [Amini-Harandi et al. 2013; Nieto and Rodríguez-López 2005; Ran and Reurings 2004; Smyth 1988]. The definition of pre-ordered metric space that we use in this article is a natural one, equivalent to requiring that the relation $\prec$ be *closed* in the topological sense. It is perhaps most similar to the *regular* pre-ordered metric spaces of Amini-Harandi et al. [2013], but while our pre-ordered metric spaces are regular, it is unclear if the reverse implication also holds.

4 IMPROVING WITH METRICS

4.1 Denotational Improvement Semantics

Recall the improvement induction rule:

$$\frac{M \mathrel{\underset{\sim}{\triangleright}} \checkmark\mathbb{C}[M] \qquad \checkmark\mathbb{C}[N] \mathrel{\underset{\sim}{\triangleleft\triangleright}} N}{M \mathrel{\underset{\sim}{\triangleright}} N}$$

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, $M \mathrel{\underset{\sim}{\triangleright}} \checkmark\mathbb{C}[M]$, suggests that M will be a pre-fixed point of some function corresponding to the context $\checkmark \cdot \mathbb{C}$, while the second premise, $\checkmark\mathbb{C}[N] \mathrel{\underset{\sim}{\triangleleft\triangleright}} N$, suggests that N will be the unique fixed point of the same function. Therefore, the above theorem will give us that $M \mathrel{\underset{\sim}{\triangleright}} N$. To formalise this idea, given a language with a set of terms *Trm*, a set of contexts *Ctx* and a special context $\checkmark$, a *denotational improvement semantics* consists of three components,

- A pre-ordered metric space $(X, d, \prec)$
- A *term* interpretation $[\![-]\!] : \mathit{Trm} \to X$
- A *context* interpretation $[\![-]\!] : \mathit{Ctx} \to (X \to X)$

such that

- For all contexts $\mathbb{C}$ and terms M, $[\![\mathbb{C}]\!]$ is monotone and non-expansive and $[\![\mathbb{C}]\!]([\![M]\!]) = [\![\mathbb{C}[M]]\!]$
- $[\![\checkmark]\!]$ is a contraction

Given the above, we can define the improvement relation by $M \mathrel{\underset{\sim}{\triangleright}} N \Leftrightarrow [\![M]\!] \prec [\![N]\!]$. (Note that two orderings in this definition 'point' in opposite directions, because improvement $\mathrel{\underset{\sim}{\triangleright}}$ points to the *improved* term, whereas $\prec$ is a refinement ordering where the *better* term is greater.) We will write δ for $[\![\checkmark]\!]$. 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. The first premise, $M \mathrel{\underset{\sim}{\triangleright}} \checkmark\mathbb{C}[M]$, can be read as $[\![M]\!] \prec (\delta \cdot [\![\mathbb{C}]\!])([\![M]\!])$, i.e. that $[\![M]\!]$ is a post-fixed point of the contraction $\delta \cdot [\![\mathbb{C}]\!]$. The second states that $\delta \cdot [\![\mathbb{C}]\!]([\![N]\!]) = N$, meaning that $[\![N]\!]$ is the unique fixed point of this function. Since this function is a monotone contraction, Theorem 3.3 applies gives $[\![M]\!] \prec [\![N]\!]$ and hence $M \mathrel{\underset{\sim}{\triangleright}} N$. □

Gustavsson and Sands [1999, 2001] present an improvement theory for space that lacks this kind of induction rule. Our proof explains 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 use the same amount of memory, but produce different results. For any gadget $\checkmark$, there will be some context $\mathbb{C}$ that re-uses the space it uses, implying that $\checkmark\mathbb{C}[M] \mathrel{\underset{\sim}{\triangleleft\triangleright}} \mathbb{C}[M]$ for any M. This means that $\delta \cdot [\![\mathbb{C}]\!]$ will be the identity, i.e. not a contraction, but this contradicts the requirement that δ be a contraction. Therefore, if we wish to use improvement induction for 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:

$$\begin{array}{l}
\textbf{let}\ \ x_0 = \bot\ \ \textbf{in}\ \ N[x_0/x] \\
\textbf{let}\ \ x_0 = \bot;\ x_1 = M[x_0/x]\ \ \textbf{in}\ \ N[x_1/x] \\
\textbf{let}\ \ x_0 = \bot;\ x_1 = M[x_0/x];\ x_2 = M[x_1/x]\ \ \textbf{in}\ \ N[x_2/x] \\
\vdots
\end{array}$$

The symbol $\bot$ is a suitable element to start the chain, usually a bottom element for the underlying improvement order. In each successive term we have "unwound" the recursive definition a little further, allowing for one more level of recursion than the previous.

Similarly, in our *denotational* approach, we must have a way to characterise programs in terms of finite approximations. In the setting 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 $\mathbb{C}$ that contain no occurrences of x, we have $[\![\textbf{let}\ x = \mathbb{D}[x]\ \textbf{in}\ \mathbb{C}[x]]\!] = [\![\mathbb{C}]\!](\mathsf{fix}\ (\delta \cdot [\![\mathbb{D}]\!]))$. The use of the contraction δ ensures that the fixed point $\mathsf{fix}\ (\delta \cdot [\![\mathbb{D}]\!])$ exists and is unique.

Furthermore, if there is a sequence of terms $M_0, M_1, M_2 \ldots$ where $[\![M_n]\!] = [\![\mathbb{C}]\!]((\delta \cdot [\![\mathbb{D}]\!])^n [\![\bot]\!])$, we say that the term **let** $x = \mathbb{D}[x]$ **in** $\mathbb{C}[x]$ has *unwindings* $M_0, M_1, M_2 \ldots$, and the semantics has *syntactic continuity*. Note that the interpretations of these unwindings will form a Cauchy sequence whose limit is $[\![\mathbb{C}]\!](\mathsf{fix}\ (\delta \cdot [\![\mathbb{D}]\!]))$. The exact form of the terms M_i will vary depending on the particular language and semantics.

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:

$$\frac{\exists k.\ \forall n \geq k.\ M_n \mathrel{\underset{\sim}{\triangleright}} N_n}{M \mathrel{\underset{\sim}{\triangleright}} N}$$

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:

$$\frac{\exists k.\ \forall n \geq k.\ M_n \mathrel{\underset{\sim}{\triangleright}} N_n \mathrel{\underset{\sim}{\triangleright}} P_n \quad M \mathrel{\underset{\sim}{\triangleleft\triangleright}} P}{M \mathrel{\underset{\sim}{\triangleleft\triangleright}} N \mathrel{\underset{\sim}{\triangleleft\triangleright}} P}$$

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.

4.4 Improvement Theorem

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

$$\frac{\textbf{let}\ f = V\ \textbf{in}\ V \mathrel{\underset{\sim}{\triangleright}} \textbf{let}\ f = V\ \textbf{in}\ W}{\textbf{let}\ f = V\ \textbf{in}\ N \mathrel{\underset{\sim}{\triangleright}} \textbf{let}\ f = W\ \textbf{in}\ N}$$

This rule establishes when it is safe to replace a recursive value V with another value W. Note that this is strictly stronger than simply applying congruence to a proof of $V \underset{\sim}{\triangleright} W$, as this allows us to prove the relationship between V and W in a context that contains the recursive binding $x = V$. In this way, we can turn an improvement of only the outer level of a recursive binding into one that improves at every level.

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

$$f(\mathsf{fix}\ (\delta \cdot f)) \prec g(\mathsf{fix}\ (\delta \cdot f)) \Rightarrow \mathsf{fix}\ (\delta \cdot f) \prec \mathsf{fix}\ (\delta \cdot g)$$

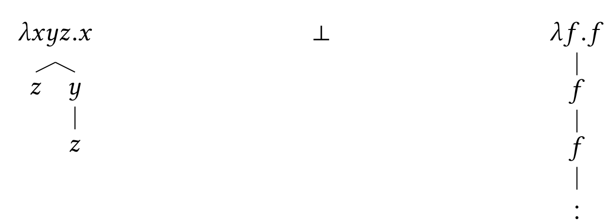
Given the precondition $f(\mathsf{fix}\ (\delta \cdot f)) \prec g(\mathsf{fix}\ (\delta \cdot f))$, we can show by induction that $(\delta \cdot f)^n(\mathsf{fix}\ (\delta \cdot f)) \prec (\delta \cdot g)^n(\mathsf{fix}\ (\delta \cdot f))$ for all n. The result follows from the properties that $\lim_{n\to\infty}(\delta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

5 EXAMPLE: CLOCKED BÖHM TREES

Böhm trees [Barendregt 1984] are a denotational model of the lambda calculus in which terms without normal form are represented by an "infinite" normal form. Formally, the Böhm tree $BT(M)$ of a term M can be defined as follows:

- If M has no head normal form, then $BT(M) = \bot$;
- If M has head normal form $\lambda x_1 \dots x_n.yM_1 \dots M_m$, where the x_i are distinct but y may be one of the x_i, then $BT(M) = \lambda x_1 \dots x_n.y(BT(M_1) \dots BT(M_m))$.

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



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

Böhm trees are a useful model because any two terms that are not β-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 β-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 that are required to reach the head normal form. We can then define an interpretation of lambda terms as clocked Böhm trees as follows:

- If M has no head normal form, then $[\![M]\!] = \bot$;
- If M reaches head normal form $\lambda x_1 \dots x_n.yM_1 \dots M_m$ in k head reductions, where the x_i are distinct but y may be one of the x_i, then $[\![M]\!] = \lambda x_1 \dots x_n.y[k]([\![M_1]\!] \dots [\![M_m]\!])$.

In turn, given a context $\mathbb{C}$, we can inductively define its interpretation in terms of clocked Böhm trees by

$$
\begin{aligned}
[\![x]\!](T) &= [\![x]\!] \\
[\![\mathbb{C}\mathbb{D}]\!](T) &= app([\![\mathbb{C}]\!](T), [\![\mathbb{D}]\!](T)) \\
app(\lambda x.T_1, T_2) &= \delta(T_1[x := T_2]) \\
app(x[k](T_{1,1} \ldots T_{1,m}), T_2) &= x[k](T_{1,1} \ldots T_{1,m} T_2) \\
[\![\lambda x.\mathbb{C}]\!](T) &= \lambda x.[\![\mathbb{C}]\!](T)
\end{aligned}
$$

where *app* is the application operator for clocked Böhm trees, $\lambda x.T$ 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 may add one to the root of the resulting tree, to account for the cost of the extra head reduction. A straightforward induction shows $[\![\mathbb{C}]\!]([\![M]\!]) = [\![\mathbb{C}[M]]\!]$.

There is a natural ordering on clocked Böhm trees, where $T_1 \prec T_2$ if T_1 is $\perp$, 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 $T_{1,i} \prec T_{2,i}$ for all i. For example, the trees below are both fixed-point combinators, but the right has lower cost than the left:

$$
\begin{array}{c}
\lambda f.f[2] \\
| \\
f[2] \\
| \\
f[2] \\
| \\
\vdots
\end{array}
\qquad\qquad\qquad
\begin{array}{c}
\lambda f.f[1] \\
| \\
f[1] \\
| \\
f[1] \\
| \\
\vdots
\end{array}
$$

We can also define a metric on these trees. Given two clocked Böhm trees, T_1 and T_2, let $\mathit{diff}(T_1, T_2)$ 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 ∞. We treat $\perp$ as a node of infinite cost for this purpose. For example, the *diff* of the trees above is 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.

We now define $d(T_1, T_2) = 2^{-\mathit{diff}(T_1,T_2)}$, with the convention that $2^{-\infty} = 0$, which turns 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 $\perp$ or eventually stabilising. Furthermore, limits will respect the order $\prec$ by a similar argument. Therefore, the set of clocked Böhm trees forms a pre-ordered metric space. We write δ for the contractive function that adds one to the cost at the root of a tree. Note that $[\![\checkmark]\!] = \delta$, where $\checkmark = (\lambda x.x)[-]$, and that the interpretations of contexts will be non-expansive and monotone. Hence, clocked Böhm trees form a denotational improvement semantics for the lambda calculus.

5.1 Fixed-Point Combinators

Now that we have such a denotational improvement semantics, 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

$$\theta \;=\; (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

First, note that Turing's combinator reduces to $\lambda y.y(\theta y)$ in one step, so $\theta \mathrel{\underset{\sim}{\lhd\!\rhd}} \checkmark\lambda y.y(\theta y)$. By improvement induction, if we can show Y improves $\lambda y.y(Yy)$ then we will have Y improves θ. We calculate as follows:

$$
\begin{array}{ll}
& Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \\
\underset{\sim}{\lhd\!\rhd} & \{\beta\text{-reduction}\} \\
& \lambda f.\checkmark f((\lambda x.f(xx))(\lambda x.f(xx))) \\
\underset{\sim}{\lhd} & \{\text{adding a } \checkmark\} \\
& \lambda f.\checkmark f(\checkmark(\lambda x.f(xx))(\lambda x.f(xx))) \\
\underset{\sim}{\lhd} & \{\text{pushing } \checkmark \text{ inwards is an improvement}\} \\
& \checkmark \lambda f.f(\checkmark(\lambda x.f(xx))(\lambda x.f(xx))) \\
\underset{\sim}{\lhd\!\rhd} & \{\beta\text{-reduction}\} \\
& \checkmark \lambda f.f((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))f) \equiv \lambda f.f(Yf)
\end{array}
$$

The preconditions of improvement induction are satisfied, so we can conclude that $\theta \underset{\sim}{\rhd} Y$. Functional programmers will be pleased to learn that Curry improves Turing!

6 EXAMPLE: NONDETERMINISM WITH FAILURE

We can also use our framework to reason about resources other than space and time. One interesting example is that of *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 total functional language with two extra operators: **choice** and **fail**. The first of these operators non-deterministically chooses between one of two options, while the second allows for the program to backtrack if it turns out one of the previous choices was incorrect. Our cost function simply counts the number of times that **choice** is invoked during evaluation.

For our semantics, we will use *non-empty bags*. Formally, a bag over a set X is a function $f : X \to \mathbb{N}$. 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 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 $A \to \mathcal{B}^+(B \cup \{\bot\})$. 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 by:

$$b_1 \prec b_2 \;\Leftrightarrow\; \forall o \in B.\; b_1(o) \geq b_2(o) \;\wedge\; b_1(\bot) \geq b_2(\bot)$$

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,

$$d_{\mathcal{B}}(b_1, b_2) \;=\; \sup\{2^{-\min\{b_1(o), b_2(o)\}} \mid o \in B \cup \{\bot\}, b_1(o) \neq b_2(o)\}$$

and this metric can also be lifted to functions:

$$d(f, g) \;=\; \sup\{d(f(i), g(i)) | i \in A\}$$

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 ✓ to be **choice** $[-]$ **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.

6.1 Sorting

Now we can apply our theory to an example. Consider the following nondeterministic sorting program which arbitrarily permutes the input list and checks if the result is ordered:

$$\mathit{sort} \;=\; \mathit{checkOrdered} \cdot \mathit{permute}$$

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:

$$\begin{array}{l}
\mathit{permute}\ \mathit{xs} \;=\; \textbf{if}\ \mathit{null}\ \mathit{xs}\ \textbf{then}\ []\ \textbf{else}\\
\quad \textbf{let}\ (r, \mathit{rs}) = \mathit{selectRand}\ \mathit{xs}\ \textbf{in}\ r : \mathit{permute}\ \mathit{rs}
\end{array}$$

$$\begin{array}{lcl}
\mathit{selectRand}\ [x] & = & (x, [\,])\\
\mathit{selectRand}\ (x : \mathit{xs}) & = & \textbf{choice}\ (x,\mathit{xs})\\
\multicolumn{3}{l}{\quad (\textbf{let}\ (r, \mathit{rs}) = \mathit{selectRand}\ \mathit{xs}\ \textbf{in}\ (r, x : \mathit{rs}))}
\end{array}$$

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:

$$\begin{array}{lcl}
\mathit{checkOrdered}\ [] & = & []\\
\mathit{checkOrdered}\ (x : \mathit{xs}) & = & \textbf{if}\ \mathit{all}\ (\geq x)\ \mathit{xs}\ \textbf{then}\ x : \mathit{checkOrdered}\ \mathit{xs}\ \textbf{else}\ \textbf{fail}
\end{array}$$

While the above definition for *sort* is valid, 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:

$$\begin{array}{ll}
 & \mathit{sort}\ \mathit{xs}\\
\underset{\sim}{\lhd\!\rhd} & \{\text{definition of } \mathit{sort}\}\\
 & \mathit{checkOrdered}\ (\mathit{permute}\ \mathit{xs})\\
\underset{\sim}{\lhd\!\rhd} & \{\text{definition of } \mathit{permute}\}\\
 & \textbf{if}\ \mathit{null}\ \mathit{xs}\ \textbf{then}\ \mathit{checkOrdered}\ []\ \textbf{else}\\
 & \quad \textbf{let}\ (r, \mathit{rs}) = \mathit{selectRand}\ \mathit{xs}\ \textbf{in}\\
 & \qquad \mathit{checkOrdered}\ (r : \mathit{permute}\ \mathit{rs})\\
\underset{\sim}{\lhd\!\rhd} & \{\text{definition of } \mathit{checkOrdered}\}\\
 & \textbf{if}\ \mathit{null}\ \mathit{xs}\ \textbf{then}\ []\ \textbf{else}\\
 & \quad \textbf{let}\ (r, \mathit{rs}) = \mathit{selectRand}\ \mathit{xs};\ \ \mathit{rs}' = \mathit{permute}\ \mathit{rs}\ \textbf{in}\\
 & \qquad \textbf{if}\ \mathit{all}\ (\geq r)\ \mathit{rs}'\\
 & \qquad\ \ \textbf{then}\ r : \mathit{checkOrdered}\ \mathit{rs}'
\end{array}$$

$$
\begin{array}{ll}
& \qquad\qquad \mathbf{else}\;\; \mathbf{fail} \\
\mathrel{\underset{\sim}{\triangleright}} & \{ \mathit{all} \text{ is order-independent} \} \\
& \mathbf{if}\;\; \mathit{null}\;\; \mathit{xs}\; \mathbf{then}\; [\,]\; \mathbf{else} \\
& \quad \mathbf{let}\;\; (r, \mathit{rs})\; = \mathit{selectRand}\; \mathit{xs};\;\; \mathit{rs}'\; = \mathit{permute}\; \mathit{rs}\; \mathbf{in} \\
& \qquad \mathbf{if}\;\; \mathit{all}\;\; (\geq r)\;\; \mathit{rs} \\
& \qquad\quad \mathbf{then}\; r\; :\; \mathit{checkOrdered}\; \mathit{rs}' \\
& \qquad\quad \mathbf{else}\;\; \mathbf{fail} \\
\mathrel{\underset{\sim}{\triangleleft\triangleright}} & \{ \text{substituting } \mathit{rs}' \} \\
& \mathbf{if}\;\; \mathit{null}\;\; \mathit{xs}\; \mathbf{then}\; [\,]\; \mathbf{else} \\
& \quad \mathbf{let}\;\; (r, \mathit{rs})\; = \mathit{selectRand}\; \mathit{xs}\; \mathbf{in} \\
& \qquad \mathbf{if}\;\; \mathit{all}\;\; (\geq r)\;\; \mathit{rs} \\
& \qquad\quad \mathbf{then}\; r\; :\; \mathit{checkOrdered}\; (\mathit{permute}\; \mathit{rs}) \\
& \qquad\quad \mathbf{else}\;\; \mathbf{fail} \\
\mathrel{\underset{\sim}{\triangleleft\triangleright}} & \{ \text{definition of } \mathit{sort} \} \\
& \mathbf{if}\;\; \mathit{null}\;\; \mathit{xs}\; \mathbf{then}\; [\,]\; \mathbf{else} \\
& \quad \mathbf{let}\;\; (r, \mathit{rs})\; = \mathit{selectRand}\; \mathit{xs}\; \mathbf{in} \\
& \qquad \mathbf{if}\;\; \mathit{all}\;\; (\geq r)\;\; \mathit{rs}\;\; \mathbf{then}\; r\; :\; \mathit{sort}\;\; \mathit{rs}\;\; \mathbf{else}\;\; \mathbf{fail}
\end{array}
$$

Note the highlights where we switch from rs' to rs. In summary, we have shown that *sort* is a pre-fixed point of:

$$
\begin{array}{l}
\mathbb{C}\;\; = \;\; \lambda \mathit{xs} \rightarrow \mathbf{if}\; \mathit{null}\; \mathit{xs}\; \mathbf{then}\; [\,]\;\; \mathbf{else} \\
\qquad\qquad \mathbf{let}\;\; (r, \mathit{rs})\; = \mathit{selectRand}\; \mathit{xs}\; \mathbf{in} \\
\qquad\qquad\quad \mathbf{if}\;\; \mathit{all}\;\; (\geq r)\;\; \mathit{rs}\;\; \mathbf{then}\; r\; :\; [-]\; \mathit{rs}\;\; \mathbf{else}\;\; \mathbf{fail}
\end{array}
$$

That is, $\mathit{sort} \mathrel{\underset{\sim}{\triangleright}} \mathbb{C}[\mathit{sort}]$. Moreover, due to the use of the nondeterministic function *selectRand*, the interpretation of $\mathbb{C}$ 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 *sort*, i.e. for which $\mathit{sort} \mathrel{\underset{\sim}{\triangleright}} f$. However, because $\mathbb{C}$ 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 $\mathbb{C}$, i.e. that we have $\mathbb{C}[f] \mathrel{\underset{\sim}{\triangleright}} f$.

We can use this proof obligation as a starting point to derive a program for *selection sort*, based on a function *selectMin* that satisfies the following specification:

$$
\begin{array}{ll}
& \mathbf{let}\;\; (r, \mathit{rs})\; = \;\; \mathit{selectRand}\; \mathit{xs}\; \mathbf{in} \\
& \quad \mathbf{if}\;\; \mathit{all}\;\; (\geq r)\;\; \mathit{rs}\;\; \mathbf{then}\; (r,\;\; \mathit{rs})\;\; \mathbf{else}\;\; \mathbf{fail} \\
\mathrel{\underset{\sim}{\triangleright}} & \\
& \mathit{selectMin}\; \mathit{xs}
\end{array}
$$

This specification states that *selectMin* behaves in a similar manner to *selectRand*, with the key difference that it will only return values r that are minimal in the input list, whereas *selectRand* returns any value from the list. Based on such a function, we now proceed to derive our sorting function, starting from the proof obligation $\mathbb{C}[f] \mathrel{\underset{\sim}{\triangleright}} f$:

$$
\begin{array}{ll}
& \mathbb{C}[f] \\
\equiv & \{ \text{definition of } \mathbb{C} \} \\
& \lambda \mathit{xs} \rightarrow \mathbf{if}\; \mathit{null}\; \mathit{xs}\; \mathbf{then}\; [\,]\;\; \mathbf{else} \\
& \quad \mathbf{let}\;\; (r, \mathit{rs})\; = \mathit{selectRand}\; \mathit{xs}\; \mathbf{in}
\end{array}
$$

$$
\begin{array}{l}
\quad\quad \textbf{if } \mathit{all}\ (\geq r)\ \mathit{rs}\ \textbf{then } r : f\ \mathit{rs}\ \textbf{else fail} \\
\underset{\sim}{\triangleright} \quad \{\text{factoring out } r : f\ \mathit{rs}\} \\
\lambda \mathit{xs} \rightarrow \textbf{if } \mathit{null}\ \mathit{xs}\ \textbf{then } [\,]\ \textbf{else} \\
\quad \textbf{let } (r, \mathit{rs}) = \mathit{selectRand}\ \mathit{xs}\ \textbf{in} \\
\quad\quad \textbf{let } (r', \mathit{rs}') = \textbf{if } \mathit{all}\ (\geq r)\ \mathit{rs} \\
\quad\quad\quad\quad\quad\quad \textbf{then } (r,\ \mathit{rs}) \\
\quad\quad\quad\quad\quad\quad \textbf{else fail} \\
\quad\quad\quad \textbf{in } r' : f\ \mathit{rs}' \\
\underset{\sim}{\triangleleft\!\triangleright} \quad \{\text{rearranging lets}\} \\
\lambda \mathit{xs} \rightarrow \textbf{if } \mathit{null}\ \mathit{xs}\ \textbf{then } [\,]\ \textbf{else} \\
\quad \textbf{let } (r', \mathit{rs}') = \textbf{let } (r, \mathit{rs}) = \mathit{selectRand}\ \mathit{xs}\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{if } \mathit{all}\ (\geq r)\ \mathit{rs} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{then } (r,\ \mathit{rs}) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{else fail} \\
\quad\quad \textbf{in } r' : f\ \mathit{rs}' \\
\underset{\sim}{\triangleright} \quad \{\text{specification of } \mathit{selectMin}\} \\
\lambda \mathit{xs} \rightarrow \textbf{if } \mathit{null}\ \mathit{xs}\ \textbf{then } [\,]\ \textbf{else} \\
\quad \textbf{let } (r', \mathit{rs}') = \mathit{selectMin}\ \mathit{xs}\ \textbf{in } r' : f\ \mathit{rs}' \\
\underset{\sim}{\triangleright} \quad \{\text{defining } f \text{ to be this term}\} \\
f
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{ssort}\ \mathit{xs} \ = \ \textbf{if } \mathit{null}\ \mathit{xs}\ \textbf{then } [\,]\ \textbf{else} \\
\quad \textbf{let } (r, \mathit{rs}) \ = \ \mathit{selectMin}\ \mathit{xs}\ \textbf{in } r : \mathit{ssort}\ \mathit{rs}
\end{array}
$$

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.

7 EXAMPLE: COST-IN-CONTEXT SEMANTICS

Escardó [1999] showed how metric spaces could be used to give a denotational semantics for typed functional programs, with distance at base types defined based on the difference in cost to produce a result. Inspired by this idea, we develop 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 an abstract "cost function" rather than any concrete semantics.

In order to realise this, let *Trm* be the set of valid terms in a programming language, *Ctx* be the set of 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 : $\mathit{Trm} \longrightarrow \mathbb{N}_\omega$, where $\mathbb{N}_\omega = \mathbb{N} \cup \{\infty\}$, as follows:

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

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

$$
[\![M]\!](\mathbb{C}) \ = \ \mathrm{cost}(\mathbb{C}[M])
$$

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 the improvement relation derived from this order will coincide with the operational improvement relation of Moran and Sands [1999a]:

$$
\begin{aligned}
& [\![N]\!] \leq^\star [\![M]\!] \\
\Leftrightarrow \quad & \{\text{definition of } \leq^\star\} \\
& \forall \mathbb{C}.\ [\![N]\!](\mathbb{C}) \leq [\![M]\!](\mathbb{C}) \\
\Leftrightarrow \quad & \{\text{definition of } [\![-]\!]\} \\
& \forall \mathbb{C}.\ \mathrm{cost}(\mathbb{C}[N]) \leq \mathrm{cost}(\mathbb{C}[M]) \\
\Leftrightarrow \quad & \{\text{definition of cost, properties of } \leq\} \\
& \forall \mathbb{C}.\ \mathbb{C}[M] \Downarrow^n \ \Rightarrow\ \mathbb{C}[N] \Downarrow^{\leq n} \\
\Leftrightarrow \quad & \{\text{Moran and Sands' definition of } \trianglerighteq\} \\
& M \trianglerighteq N
\end{aligned}
$$

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 $Ctx \to \mathbb{N}_\omega$ by:

$$d(p, q) \ = \ \sup\{r^{\min\{p(\mathbb{C}), q(\mathbb{C})\}} \mid \mathbb{C} \in Ctx,\ p(\mathbb{C}) \neq q(\mathbb{C})\}$$

(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 manner, we capture Escardó's idea that the greater the cost required to distinguish terms, the closer they are. It is easy to show that under the above definition our semantic domain forms a complete metric space, with limits computed pointwise. Because the ordering $\leq^\star$ is pointwise, limits respect the ordering, and hence $(Ctx \to \mathbb{N}_\omega, d, \leq^\star)$ is a pre-ordered metric space.

Each context $\mathbb{C}$ induces a map $[\![\mathbb{C}]\!]$ on $Ctx \to \mathbb{N}_\omega$, defined by $[\![\mathbb{C}]\!](p) = \lambda \mathbb{D}.\ p(\mathbb{D}[\mathbb{C}[-]])$. This map is clearly monotone, and a simple calculation shows that it is non-expansive:

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

The map $[\![\mathbb{C}]\!]$ is the denotational analogue of the context $\mathbb{C}$, in the sense that $[\![\mathbb{C}]\!]([\![M]\!]) = [\![\mathbb{C}[M]]\!]$.

Finally, there is a function δ on $Ctx \to \mathbb{N}_\omega$ that adds one unit of cost in all contexts, defined by $\delta(p) = \lambda\mathbb{C}.\ p(\mathbb{C}) + 1$. It is clear that $[\![\checkmark]\!] = \delta$, and we can show δ is a contraction:

$$
\begin{aligned}
& d(\delta(p), \delta(q)) \\
= \quad & \{\text{definitions of } d, \delta\} \\
& \sup\{r^{\min\{p(\mathbb{C})+1, q(\mathbb{C})+1\}} \mid \mathbb{C} \in Ctx, p(\mathbb{C}) + 1 \neq q(\mathbb{C}) + 1\} \\
= \quad & \{\text{arithmetic}\} \\
& \sup\{r \cdot r^{\min\{p(\mathbb{C}), q(\mathbb{C})\}} \mid \mathbb{C} \in Ctx,\ p(\mathbb{C}) \neq q(\mathbb{C})\} \\
= \quad & \{\text{multiplication distributes over sup}\} \\
& r \cdot \sup\{r^{\min\{p(\mathbb{C}), q(\mathbb{C})\}} \mid \mathbb{C} \in Ctx,\ p(\mathbb{C}) \neq q(\mathbb{C})\} \\
= \quad & \{\text{definition of } d\} \\
& r \cdot d(p, q)
\end{aligned}
$$

To summarise, we have a pre-ordered metric space $(Ctx \to \mathbb{N}_\omega, d, \leq^\star)$ and interpretation functions $[\![-]\!] : Trm \to (Ctx \to \mathbb{N}_\omega)$ and $[\![-]\!] : Ctx \to (Ctx \to \mathbb{N}_\omega) \to (Ctx \to \mathbb{N}_\omega)$, 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 $[\![\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ x]\!] = \mathsf{fix}\ (\delta \cdot [\![\mathbb{D}]\!])$, which we do by showing that the left hand side is a fixed point of $\delta \cdot [\![\mathbb{D}]\!]$, which is a contraction:

$$
\begin{aligned}
& [\![\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ x]\!] \\
= \quad & \{\text{lookup costs one tick}\} \\
& [\![\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ \checkmark\mathbb{D}[x]]\!] \\
= \quad & \{x \text{ free in } \mathbb{D}\} \\
& [\![\checkmark\mathbb{D}[\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ x]]\!] \\
= \quad & \{\text{interpretation of contexts}\} \\
& [\![\checkmark\mathbb{D}]\!]([\![\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ x]\!])
\end{aligned}
$$

Now we can verify the continuity condition:

$$
\begin{aligned}
& [\![\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ \mathbb{C}[x]]\!] \\
= \quad & \{x \text{ free in } \mathbb{C}\} \\
& [\![\mathbb{C}[\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ x]]\!] \\
= \quad & \{\text{interpretation of contexts}\} \\
& [\![\mathbb{C}]\!]([\![\mathbf{let}\ x = \mathbb{D}[x]\ \mathbf{in}\ x]\!]) \\
= \quad & \{\text{see above}\} \\
& [\![\mathbb{C}]\!](\mathsf{fix}\ (\delta \cdot [\![\mathbb{D}]\!]))
\end{aligned}
$$

This reasoning is invalid for call-by-need semantics, as lookup loses sharing between recursion levels. However, we can recover continuity for call-by-need by restricting ourselves to bindings where the right side is 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 $\mathbf{let}\ f = V\ \mathbf{in}\ \mathbb{C}[f]$ as limits of sequences of finite unwindings, the interpretations 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 unwindings of a recursive binding, if the recursive call is a tail call or occurs in a thunk, then the space used could be the same. Because the sequence may not 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. In particular, 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

$$\mathrm{cost}(M) \;=\; f(\mathrm{cost}_{time}(M), \mathrm{cost}_{space}(M))$$

where cost_{time} and cost_{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 more-or-less any way that we please, but we cannot ignore time altogether if we wish to retain continuity.

7.1 Compact Reverse

Consider a simple *reverse* function on lists, and a more efficient version *reverse′* that uses an accumulator:

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

$$\begin{array}{l} \mathit{reverse}'\ \mathit{xs} \;=\; \mathit{revcat}\ \mathit{xs}\ [\,]\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{revcat}\ [\,]\ \ \ \ \ \ \ \ \mathit{ys} & = & \mathit{ys} \\ \mathit{revcat}\ (x : \mathit{xs})\ \mathit{ys} & = & \mathit{revcat}\ \mathit{xs}\ (x : \mathit{ys}) \end{array} \end{array}$$

Intuitively, *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, *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 *reverse'* also has another advantage: it uses less space. However, as mentioned before, space improvement on its own will not produce a denotational improvement semantics, so we cannot use techniques such as improvement induction to verify this. However, the semantic domain is still a pre-ordered metric space, and we can still use a metric space-based technique to prove this improvement.

Consider a Cauchy sequence $x_1, x_2, \ldots$ in a pre-ordered metric space that is *increasing*, i.e. $x_n \prec x_{n+1}$ for any n. If this sequence has a limit, this will also be an upper bound to the sequence. Dually, the limit of a *decreasing* sequence will be a *lower* bound. Therefore, one way to prove an improvement between two programs is to construct an increasing or decreasing Cauchy sequence that starts with one and becomes the other in the limit. We call such a sequence a *chain of unfoldings*, because it will typically be constructed from the unfoldings of one of the programs.

The following is a chain of unfoldings that starts with *reverse'* and ends with the less efficient *reverse*:

$$M_0 = \mathit{reverse}'\,\mathit{xs}$$

$$\begin{array}{l} M_1 = \textbf{case}\ \mathit{xs}\ \textbf{of} \\ \qquad\qquad \begin{array}{lcl} [\,] & \to & [\,] \\ x : \mathit{xs}' & \to & \mathit{reverse}'\ \ \mathit{xs}'\ +\!\!+\ [x] \end{array} \end{array}$$

$$
\begin{aligned}
M_2 = \;&\textbf{case}\ xs\ \textbf{of} \\
&\quad [\,] \quad\;\; \rightarrow [\,] \\
&\quad x : xs' \rightarrow (\textbf{case}\ xs'\ \textbf{of} \\
&\qquad [\,] \qquad\; \rightarrow [\,] \\
&\qquad x' : xs'' \;\; \rightarrow \mathit{reverse}'\ \ xs''\ \ +\!\!+\ \ [x']) \ +\!\!+\ [x]
\end{aligned}
$$

$$\vdots$$

The interpretations of these terms form a Cauchy sequence with limit $[\![\mathit{reverse}\ xs]\!]$, because each subsequent term mimics more of the space behaviour of *reverse*; note that each level of recursion strictly increases space usage. Therefore, it suffices to show that the sequence forms a chain of reversed improvements, i.e. $M_{n+1} \gtrsim M_n$ for all $n \geq 0$. The ordering must be 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, $M_1 \gtrsim M_0$. Our proof of the base case makes use of a lemma, namely that $\mathit{revcat}\ \ xs\ [\,]\ +\!\!+\ [x] \gtrsim \mathit{revcat}\ \ xs\ [x]$. We use the following auxiliary definition:

$$
\begin{aligned}
&\mathit{revcat}'\ \ x\ xs\ ys\ \ =\ \ \textbf{case}\ xs\ \textbf{of} \\
&\quad [\,] \qquad\; \rightarrow [\,]\ +\!\!+\ [x] \\
&\quad x' : xs' \;\rightarrow \mathit{revcat}'\ \ x\ \ xs'\ \ (x' : ys)
\end{aligned}
$$

Note that the term $\mathit{revcat}\ \ xs\ [\,]\ +\!\!+\ [x]$ is cost-equivalent to $\mathit{revcat}'\ \ x\ xs\ [\,]$, as unrolling $\mathit{revcat}\ \ xs\ [\,]$ and distributing $+\!\!+\ [x]$ over the cases results in unrollings of the latter. Next, we consider the following chain of unfoldings:

$$N_0 = \mathit{revcat}\ xs\ [x]$$

$$
\begin{aligned}
N_1 = \;&\textbf{case}\ xs\ \textbf{of} \\
&\quad [\,] \qquad\; \rightarrow [\,]\ +\!\!+\ [x] \\
&\quad x' : xs' \;\rightarrow \mathit{revcat}\ \ xs'\ \ ([x'] +\!\!+ [x])
\end{aligned}
$$

$$
\begin{aligned}
N_2 = \;&\textbf{case}\ xs\ \textbf{of} \\
&\quad [\,] \qquad\; \rightarrow [\,]\ +\!\!+\ [x] \\
&\quad x' : xs' \;\rightarrow \textbf{case}\ xs'\ \textbf{of} \\
&\qquad [\,] \qquad\quad \rightarrow [x']\ +\!\!+\ [x] \\
&\qquad x'' : xs'' \;\; \rightarrow \mathit{revcat}\ xs''\ ([\,x'', x'\,]\ +\!\!+\ [x])
\end{aligned}
$$

$$\vdots$$

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

Now we return to our proof obligation, $M_1 \gtrsim M_0$:

$$
\begin{aligned}
&\quad M_1 \\
\equiv &\quad \{\text{definition of } M_1\} \\
&\quad \textbf{case}\ xs\ \textbf{of} \\
&\qquad [\,] \qquad \rightarrow [\,]
\end{aligned}
$$

$$
\begin{array}{ll}
& \quad x : xs' \;\rightarrow\; \mathit{reverse'}\;\; xs' \;+\!\!+\; [x] \\
\underset{\sim}{\triangleleft\triangleright} & \{\text{definition of } \mathit{reverse'}\} \\
& \textbf{case}\; xs\; \textbf{of} \\
& \quad [\,] \qquad\; \rightarrow [\,] \\
& \quad x : xs' \;\rightarrow\; \mathit{revcat}\;\; xs'\;\, [\,] \;+\!\!+\; [x] \\
\underset{\sim}{\triangleright} & \{\text{above lemma}\} \\
& \textbf{case}\; xs\; \textbf{of} \\
& \quad [\,] \qquad\; \rightarrow [\,] \\
& \quad x : xs' \;\rightarrow\; \mathit{revcat}\;\; xs'\;\, [x] \\
\underset{\sim}{\triangleleft\triangleright} & \{\text{definition of } \mathit{revcat}\} \\
& \mathit{revcat}\;\; xs\;\, [\,] \\
\underset{\sim}{\triangleleft\triangleright} & \{\text{definition of } \mathit{reverse'}\} \\
& \mathit{reverse'}\;\; xs \\
\equiv & \{\text{definition of } M_0\} \\
& M_0
\end{array}
$$

Finally, because every step in the chain M_n is a reversed improvement, the first element of the chain *reverse′ xs* must be an improvement of its limit *reverse xs*, which establishes that *reverse′* is indeed a space improvement of *reverse*.

8 RELATED WORK

We partition our review of related work into two 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, based on the idea of an inductively-defined *size function*. In this case, the 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 for complexity analysis.

Ghica [2005] introduces *slot games*, a denotational cost semantics based on 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 show that if terms M and N are β-convertible, 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 symmetry requirement) on complexity functions for programs, showing that divide-and-conquer recurrences correspond to fixed points of contractive functions on this space. This allows metric 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 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 CSP and 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 implemented, and this implementation is proved correct with respect to the semantics. The same authors later extend this work to deal with graphical user interfaces [Krishnaswami and Benton 2011a].

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. We have proved a number of important results from the literature in our framework, and gave 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 straightforward 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.

Handley and Hutton [2018] recently developed a mechanical assistant for proofs of call-by-need time improvement. A similar tool could be developed based on our metric space framework, to allow programmers to benefit from our approach and techniques without the need 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.

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 by how much. This may require us to move to *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 *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.

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 *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.

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

Alireza Amini-Harandi, Majid Fakhar, Hamid Reza Hajisharifi, and Nawab Hussain. 2013. Some New Results on Fixed and Best Proximity Points in Preordered Metric Spaces. *Fixed Point Theory and Applications* (2013).

Stefan Banach. 1922. Sur Les Opérations Dans Les Ensembles Abstraits et Leur Application Aux Équations Intégrales. *Fundamenta Mathematicae* 3, 1 (1922).

Hendrik Pieter Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland Amsterdam.

Richard Bird and Oege de Moor. 1997. *Algebra of Programming.* Prentice Hall.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *ICFP*.

J. W. de Bakker and J.-J. Ch. Meyer. 1987. Order and Metric in the Stream Semantics of Elemental Concurrency. *Acta Informatica* 24, 5 (1987).

J. W. de Bakker and J. I. Zucker. 1982. Processes and the Denotational Semantics of Concurrency. *Information and Control* 54, 1 (1982).

Joerg Endrullis, Dimitri Hendriks, Jan Willem Klop, and Andrew Polonsky. 2014. Discriminating Lambda-Terms Using Clocked Böhm Trees. *Logical Methods in Computer Science* (2014).

Martín Hötzel Escardó. 1999. A Metric Model of PCF. In *Workshop on Realizability Semantics and Applications.*

Dan R. Ghica. 2005. Slot Games: A Quantitative Model of Computation. In *POPL*.

Andy Gill and Graham Hutton. 2009. The Worker/Wrapper Transformation. *JFP* 19, 2 (2009).
Jörgen Gustavsson and David Sands. 1999. A Foundation for Space-Safe Transformations of Call-by-Need Programs. *Electronic Notes on Theoretical Computer Science* (1999).
Jörgen Gustavsson and David Sands. 2001. Possibilities and Limitations of Call-by-Need Space Improvement. In *ICFP*.
Jennifer Hackett and Graham Hutton. 2014. Worker/Wrapper/Makes It/Faster. In *ICFP*.
Jennifer Hackett and Graham Hutton. 2015. Programs for Cheap!. In *Logic in Computer Science*.
Jennifer Hackett and Graham Hutton. 2018. Parametric Polymorphism and Operational Improvement. In *ICFP*.
Martin Handley and Graham Hutton. 2018. Improving Haskell. In *Trends in Functional Programming*.
Robert Harper. 2014. The Structure and Efficiency of Computer Programs. (2014). Working paper.
Neelakantan R. Krishnaswami and Nick Benton. 2011a. A Semantic Model for Graphical User Interfaces. In *ICFP*.
Neelakantan R Krishnaswami and Nick Benton. 2011b. Ultrametric Semantics of Reactive Programs. In *LICS*.
Andrew Moran and David Sands. 1999a. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. (1999). Extended version of [Moran and Sands 1999b], available at http://tinyurl.com/ohuv8ox.
Andrew Moran and David Sands. 1999b. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *Principles of Programming Languages*.
Juan J Nieto and Rosana Rodríguez-López. 2005. Contractive Mapping Theorems in Partially Ordered Sets and Applications to Ordinary Differential Equations. *Order* (2005).
Andre CM Ran and Martine CB Reurings. 2004. A Fixed Point Theorem in Partially Ordered Sets and Some Applications to Matrix Equations. *American Mathematical Society* (2004).
S. Romaguera and M. Schellekens. 1999. Quasi-Metric Properties of Complexity Spaces. *Topology and its Applications* 98, 1 (1999).
Salvador Romaguera and Oscar Valero. 2009. A Quantitative Computational Model for Complete Partial Metric Spaces Via Formal Balls. *MSCS* 19, 3 (2009).
David Sands. 1991. Operational Theories of Improvement in Functional Languages. In *Glasgow Workshop on Functional Programming*.
David Sands. 1995. Total Correctness by Local Improvement in Program Transformation. In *POPL*.
M. Schellekens. 1995. The Smyth Completion: A Common Foundation for Denotational Semantics and Complexity Analysis. *ENTCS* 1 (1995).
Manfred Schmidt-Schauß and David Sabel. 2015. Improvements in a Functional Core Language with Call-By-Need Operational Semantics. In *PPDP*.
Dana Scott. 1970. Outline of a Mathematical Theory of Computation. In *Information Science and Systems*.
Mícheál Ó Searcóid. 2006. *Metric Spaces*. Springer.
M. B. Smyth. 1988. Quasi-Uniformities: Reconciling Domains with Metric Spaces. In *MFPS*.
Alfred Tarski. 1955. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.* (1955).
Kathryn Van Stone. 2003. *A Denotational Approach to Measuring Complexity in Functional Programs*. Ph.D. Dissertation. Carnegie Mellon University.