A Cost Semantics for Self-Adjusting Computation

Ruy Ley-Wild [*]
Carnegie Mellon University
rleywild@cs.cmu.edu

Umut A. Acar [†] Matthew Fluet
Toyota Technological Institute at Chicago
{acar,fluet}@tti-c.org

Abstract

Self-adjusting computation is an evaluation model in which programs can respond efficiently to small changes to their input data by using a change-propagation mechanism that updates computation by re-building only the parts affected by changes. Previous work has proposed language techniques for self-adjusting computation and showed the approach to be effective in a number of application areas. However, due to the complex semantics of change propagation and the indirect nature of previously proposed language techniques, it remains difficult to reason about the efficiency of self-adjusting programs and change propagation.

In this paper, we propose a cost semantics for self-adjusting computation that enables reasoning about its effectiveness. As our source language, we consider a direct-style λ-calculus with first-class mutable references and develop a notion of trace distance for source programs. To facilitate asymptotic analysis, we propose techniques for composing and generalizing concrete distances via trace contexts (traces with holes). We then show how to translate the source language into a self-adjusting target language such that the translation (1) preserves the extensional semantics of the source programs and the cost of from-scratch runs, and (2) ensures that change propagation between two evaluations takes time bounded by their relative distance. We consider several examples and analyze their effectiveness by considering upper and lower bounds.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages.

Keywords Self-adjusting computation, cost semantics.

1. Introduction

In many applications it can be important or even necessary to efficiently update the output of a computation as the input undergoes small changes over time. This problem, broadly known as *incremental computation*, has been studied extensively in both the algorithms and programming languages communities.

In the algorithms community, researchers devised algorithms that are optimized to take advantage of specific small input changes. Over the course of hundreds of papers on this topic (see *e.g.*, Chiang and Tamassia 1992; Eppstein et al. 1999; Agarwal et al. 2002 for surveys), important advances have been made. Those results show that it is often possible to update computations asymptotically faster (often by a linear factor) than re-computing from scratch. However, incremental algorithms can be difficult to design and analyze, especially for sophisticated problems, (*e.g.*, 3D motion simulation (Guibas 1998)). These algorithms can also be difficult to implement and use, because of inherent complexity and non-compositionality.

Over the same period of time, the programming languages community has made significant progress on run-time and compile-time approaches to incremental computation (*e.g.*, Demers et al. 1981; Pugh and Teitelbaum 1989; see Ramalingam and Reps 1993 for a survey). The goal of this line of work is to derive incremental programs from static programs automatically or semi-automatically. The idea is to maintain certain information during an execution that can be used to efficiently update the output after changes to the input. Recent work on self-adjusting computation (*e.g.* Acar et al. 2006b,a; Ley-Wild et al. 2008b) proposed a general-purpose change-propagation mechanism that can closely match asymptotic performance bounds achieved by algorithmic techniques. Self-adjusting computation has been shown to be effective in various applications (*e.g.*, Acar et al. 2004, 2006a,c, 2008c,b). For example, recent work (Acar et al. 2008b) proposed a solution to simulating moving convex hulls in 3D, a problem that has resisted ad hoc approaches for a decade (Guibas 1998).

Reasoning about the effectiveness of self-adjusting programs, however, remains difficult. In particular, there is no cost model for self-adjusting computation. Previous applications of the approach often give only experimental results to illustrate performance gains (*e.g.*, Acar et al. 2006a,c, 2008b). Giving asymptotic bounds requires integrating change propagation into the algorithm by considering a low-level machine model akin to the RAM model (*e.g.*, Acar et al. 2004). As a result, the bounds derived do not directly apply to the code as written. More importantly, the approach does not provide a source-level reasoning mechanism. The main difficulty in reasoning about a self-adjusting program is understanding how the program responds to changes to its data. One reason for this is the complexity of the update mechanism; another is the nature of previously proposed linguistic techniques.

To see the first difficulty, consider executing a program with some input and later changing the input. In self-adjusting computation, as the program executes, information about the execution (such as data and control dependencies) is recorded. After the input is changed, the output is updated by performing *change propagation* to find the parts of the computation affected by the change using the recorded dependence information and updating stale computation by re-executing code. When re-executing code, change propagation may reuse previous computations with a form of com-

* This author was partially supported by a Bell Labs Graduate Fellowship and a gift from Intel.

† This author was partially supported by a gift from Intel.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

$$\begin{tikzcd}
\sigma^{\mathsf{s}}; e^{\mathsf{s}} \arrow[r, "\Downarrow^{\mathsf{s}}", "c^{\mathsf{s}}"'] \arrow[d, dotted, "\text{translation}"'] & v^{\mathsf{s}}; T^{\mathsf{s}} \arrow[d, dotted, "\text{translation}"] \\
\sigma^{\mathsf{t}}; e^{\mathsf{t}} \arrow[r, "\Downarrow^{\mathsf{t}}", "c^{\mathsf{t}} \in O(c^{\mathsf{s}})"'] & v^{\mathsf{t}}; T^{\mathsf{t}} \arrow[d, equal, "\text{consistency}"] \\
\sigma^{\mathsf{t}}; T_0^{\mathsf{t}} \arrow[r, "\curvearrowright^{\mathsf{t}}"] & v^{\mathsf{t}}; T^{\mathsf{t}}
\end{tikzcd}
\qquad
\begin{tikzcd}
T_1^{\mathsf{s}} \arrow[r, dashed, leftrightarrow, "\ominus^{\mathsf{s}}", "d^{\mathsf{s}}"'] \arrow[d, dotted, "\text{translation}"'] & T_2^{\mathsf{s}} \arrow[d, dotted, "\text{translation}"] \\
T_1^{\mathsf{t}} \arrow[r, dashed, leftrightarrow, "\ominus^{\mathsf{t}}", "d^{\mathsf{t}} \in O(d^{\mathsf{s}})"'] \arrow[d, equal] & T_2^{\mathsf{t}} \arrow[d, equal] \\
T_1^{\mathsf{t}} \arrow[r, dashed, leftrightarrow, "\curvearrowright^{\mathsf{t}}", "d^{\mathsf{t}}"'] & T_2^{\mathsf{t}}
\end{tikzcd}$$

Figure 1. The left diagram illustrates the correspondence between the source and target from-scratch runs and the consistency of change propagation in the target. The right diagram illustrates the correspondence between distance in the source and target, and the time for change propagation in the target.

putation memoization. Since change-propagation re-executes parts of the program code and reuses other parts of the execution, it is hard to reason about its complexity. In particular, the user may need to reason about the contexts in which sub-expressions are evaluated to distinguish changed and unchanged data, which can be difficult even with limited forms of computation reuse techniques such as lazy evaluation (*e.g.*, Wadler and Hughes 1987; Sands 1990a,b).

Other difficulties arise from the nature of the previously proposed linguistic facilities. These approaches require the programmer to mark all data that change over time and identify their dependencies, delimit the static scope of the operation that reads changeable data (essentially identifying control dependencies), and apply memoization by carefully considering whether the data dependencies are local or non-local (*e.g.* Acar et al. 2006a). Depending on the choice of the scope for the primitives and the use of memoization, the programmer may observe drastically different performance.

In this paper, we propose a cost semantics for self-adjusting computation. We consider a natural source language, give a cost semantics for the language, and develop techniques for reasoning about the similarity of executions. We then show techniques for compiling source programs into a self-adjusting target language that preserves both the extensional (meaning) and the intensional (cost) semantics of the source programs. By offering a natural, high-level source language, we eliminate the burden of restructuring a program for self-adjusting computation. By offering a cost-semantics and a translation mechanism, we provide realistic source-level reasoning techniques that guarantee performance.

Figure 1 illustrates our approach. Our source language is a λ-calculus with first-class references. Its cost semantics evaluates expressions (e^{s}) in the context of stores (σ^{s}) in the usual way, and produces a trace of the evaluation (T^{s}) and a step count (c^{s}). We quantify the similarity between evaluations of source programs with a *trace distance* ($T_1^{\mathsf{s}} \ominus^{\mathsf{s}} T_2^{\mathsf{s}} = d^{\mathsf{s}}$ states that the distance between the traces T_1^{s} and T_2^{s} is d^{s}). Intuitively, the trace measures the "edit distance" between evaluations. To give an effective distance, we show that it suffices to record function calls and store operations in the trace. We don't record complete stores, or evaluation contexts for sub-expressions.[1] Since our language is stateful, recording complete stores would lead to a coarse distance measure that overestimates distance significantly; requiring evaluation contexts makes reasoning difficult. To enable proving asymptotic complexity bounds on distance, in addition to just concrete evaluations, we develop a notion of trace contexts, which are traces with holes that can be filled with other traces. We prove that, under certain conditions, distance is additive under substitution: the distance between traces obtained via substitution into two contexts is the same the distance between the substituted traces themselves plus the distance between the contexts.

We compile the source language into a self-adjusting target language. The target language has mutable modifiable references and is in continuation-passing style; its syntax combines ideas from recent work on imperative self-adjusting computation (Acar et al. 2008a) and on compiling self-adjusting programs (Ley-Wild et al. 2008b). Evaluation of a target expression (e^{t}) takes place in the context of a store (σ^{t}) and yields a value (v^{t}) and a trace (T^{t}). The semantics include a change-propagation mechanism ($\curvearrowright^{\mathsf{t}}$) that can replay a trace from a previous run (*e.g.*, T_0^{t}) in a store (σ^{t}) to produce a value and a trace that are consistent with a from-scratch execution, while reusing the work from the initial trace (T_0^{t}). We give a cost semantics for the target language that counts steps of evaluation (but not steps of change propagation). As in the source, we define a distance for traces ($\ominus^{\mathsf{t}}$) and bound the time for change propagation by the distance between the computation traces before and after propagation.

We connect the source and target languages by providing a compilation mechanism that translates source programs into target programs. The *adaptive cps* (ACPS) translation extends recent work (Ley-Wild et al. 2008b) with support for imperative references and yields provably efficient self-adjusting programs. In particular, we prove the following properties of the translation (cf. Figure 1).

- **Extensional semantics:** The translation preserves the evaluation of source programs (top left square).
- **Intensional semantics:** The translation preserves the asymptotic cost of from-scratch runs (top left square).
- **Consistency of change propagation.** Change propagation (in the target) preserves the extensional semantics of from-scratch runs (bottom left square).
- **Trace distances.** Translated programs have asymptotically the same trace distance as their source (top right square).
- **Change propagation time.** Time for change propagation (in the target) coincides with source trace distance (right diagram).

To prove the first two properties, we generalize a folklore theorem about cps to show that an ACPS-compiled program preserves the evaluation and asymptotic complexity of a source program. The ACPS translation is more complicated than the standard translation because it threads continuations through the store. We give a simple, structural proof of the consistency of change propagation by casting it as a full replay mechanism. This simplification is made possible by the translation itself—earlier work had to use step-indexed logical relations for capturing the correspondence between stateful programs (Acar et al. 2008a). We prove the fourth property by establishing a relation between the traces of the source and the target programs. This property also bounds the time for change propagation (the last property) by showing that change propagation in the target takes time proportional to the target distance.

There are several properties of trace distance that we would like to note. First, trace distance is a relation. By defining it relationally, we allow the approach to apply to any concrete implementation technique consistent with the semantics: our main theorems state that our translation can match any source distance computed relationally. Second, trace distance is sensitive to the choice of locations. This is because trace distance compares concrete evaluations. Previous implementations of self-adjusting computations can often choose locations to minimize the trace distance. Since our theorems can match any distance computed, they apply to existing implemen-

[1] For some time, we thought that evaluation contexts, which describe how results are used, were necessary. We use evaluation contexts to prove our meta-theoretic results, but they are *not* necessary for source-level reasoning about programs.

tations. The problem of whether an implementation can efficiently achieve the minimum possible distance is not well understood: this is undecidable in general but these impossibility results typically involve programs that don't arise in practice.

Due to space restrictions, we refer the reader to the companion technical report (Ley-Wild et al. 2008a) for the details of the proofs and Twelf code.

2. An Overview of Derivation Distances

We give a high-level overview of derivation distance and contexts. As a simple example, we consider a `map` function.

Our source language is a λ-calculus with references. This language is general-purpose (Turing-complete) and expressive: it allows writing both structured programs (*e.g.*, iterative divide-and-conquer list algorithms) as well as unstructured programs (*e.g.*, graph algorithms). In this language, we can define linked lists and implement a `map` function for them as follows.

```
datatype 'a cell = nil | :: of 'a * 'a list
withtype 'a list = 'a cell ref

fun map (f : 'a -> 'b) (l : 'a list) : 'b list =
  case !l of
    nil => ref nil
  | h::t => let mt = map f t in ref ((f h)::mt) end
```

This essentially-standard implementation of `map` with pointer-based lists is actually self-adjusting: using the techniques described in this paper (Section 6), we can compile it to a self-adjusting program. The resulting self-adjusting program can be run with some input list. Afterwards, any of the contents of the references can be changed and the output can be updated via change propagation. For example, consider a specialization `mapA` of `map` that maps integers to letters of the alphabet. Consider running `mapA` with input `[1,3]` to obtain `[a,c]` and then changing the input to `[1,2,3]` by writing a new cons cell into the first tail pointer. After this change, we can run change change propagation to update the output to `[a,b,c]`.

How fast would we expect change propagation be after inserting an element into the input? Intuitively, we only need to translate the new integer into a letter, which requires constant time, but we also need to find the right place to insert the element in the output—it is not clear how much time that would take.

Derivation Distance. We develop techniques for reasoning about the effectiveness of change propagation by using derivation distance.[2] The idea is to compare the evaluation derivations of a program with two different, typically similar, inputs and compute the "edit distance" between these derivations. But what should the distance between evaluations be? Comparing evaluation derivations directly yields coarse distances. To see this, consider comparing the derivation for the evaluation of `mapA` with inputs `[1,3]` and `[1,2,3]`. Since these inputs are represented in the store and since the store is threaded through the derivation, all of derivation steps will be different—stores won't match. Thus the distance between the derivations would be linear in the size of the input—far larger than the constant that we expect.

To realize the similarity between the derivations, we exclude the store from the derivations and include the store operations instead. (`P` stands for **put** (allocation); `G` stands for **get** (dereference).) Figure 2 shows the derivations of `mapA` with inputs `[1,3]` and `[1,2,3]`. The differences between the derivations are highlighted: the two derivations differ only at steps that operate on the element 2, which is what differs between two runs. Note that the difference would remain the same even if we add more elements to these lists, *e.g.*, `[...,0,1,3,4,...]` and `[...,0,1,2,3,4,...]`.

[2] In Section 3, we represent derivations with traces and formally define trace distance. Here, we use derivations because they are more intuitive.

$$\dfrac{\mathtt{G}^{\ell_1\to 1::\ell_3} \quad \dfrac{\mathtt{G}^{\ell_3\to 3::\ell_n}\quad \dfrac{\mathtt{G}^{\ell_n\to\mathtt{nil}}\quad \mathtt{P}^{\ell'_n\uparrow\mathtt{nil}}}{\mathtt{mapA}\,(\ell_n)\Downarrow\ell'_n}\quad \mathtt{P}^{\ell_3\uparrow \mathsf{c}::\ell'_n}}{\mathtt{mapA}\,(\ell_3)\Downarrow\ell_c} \quad \mathtt{P}^{\ell_a\uparrow \mathsf{a}::\ell_c}}{\mathtt{mapA}\,(\ell_1)\Downarrow\ell_a}$$

$$\dfrac{\mathtt{G}^{\ell_1\to 1::\ell_2} \quad \dfrac{\mathtt{G}^{\ell_2\to 2::\ell_3} \quad \dfrac{\mathtt{G}^{\ell_3\to 3::\ell_n}\quad \dfrac{\mathtt{G}^{\ell_n\to\mathtt{nil}}\quad \mathtt{P}^{\ell'_n\uparrow\mathtt{nil}}}{\mathtt{mapA}\,(\ell_n)\Downarrow\ell'_n}\quad \mathtt{P}^{\ell_c\uparrow \mathsf{c}::\ell'_n}}{\mathtt{mapA}\,(\ell_3)\Downarrow\ell_c} \quad \mathtt{P}^{\ell_b\uparrow \mathsf{b}::\ell_c}}{\mathtt{mapA}\,(\ell_2)\Downarrow\ell_b} \quad \mathtt{P}^{\ell_a\uparrow \mathsf{a}::\ell_b}}{\mathtt{mapA}\,(\ell_1)\Downarrow\ell_a}$$

Figure 2. The abstract derivations for executions of `mapA` with inputs `[1,3]` (top) and `[1,2,3]` (bottom).

Of course, it is possible to make the "distance" between derivations arbitrarily small when we suppress arbitrary parts of the derivation. We prove that this distance is in fact realistic by describing how source programs may be compiled (Section 6) to a target language (Section 5) with provable efficiency.

Derivation Contexts. To reason about the asymptotic complexity bounds for distance, we need to compute distance for all (appropriately changed) inputs. This is difficult with the distance described above, which requires two concrete executions. To facilitate asymptotic analysis, we use derivation contexts (Section 3). A *derivation context* is a context with one or more holes in it. We write a hole as $\bigtriangledown^{e\Downarrow v}$, where $e \Downarrow v$ denotes the evaluation for which the hole stands. We can obtain a derivation from a context by substituting a derivation for a hole. As an example, consider the derivation, shown below, of `mapA` applied to the list $[\alpha_1, \ldots, \alpha_m]@\Box$ where $\Box$ represents an unspecified list. In the derivation ℓ_i (resp. o_i) denotes the reference to the cons cell containing input α_i (resp. output for β_i), and β_i denotes the character to which α_i is mapped. Given this derivation context, we can substitute the list `[1,3]` for $\Box$ and obtain the derivation for that input by substituting the derivation of `[1,3]` (shown in Figure 2) in place of the hole.[3]

$$\dfrac{\mathtt{G}^{\ell_1\to\alpha_1::\ell_2}\quad \dfrac{\dfrac{\mathtt{G}^{\ell_m\to\alpha_m::\ell_\Box}\quad \bigtriangledown^{\mathtt{mapA}(\ell_\Box)\Downarrow o_\Box}\quad \mathtt{P}^{o_m\uparrow\beta_m::o_\Box}}{\vdots}}{\mathtt{mapA}\,(\ell_2)\Downarrow o_2}\quad \mathtt{P}^{o_1\uparrow\beta_1::o_2}}{\mathtt{mapA}\,(\ell_1)\Downarrow o_1}$$

Let $\mathscr{D}_1[\Box]$ and $\mathscr{D}_2[\Box]$ be derivation contexts and let D'_1 and D'_2 be derivations. We prove that the distance between $\mathscr{D}_1[D'_1]$ and $\mathscr{D}_2[D'_2]$ is the sum of the distances between $\mathscr{D}_1[\Box]$ and $\mathscr{D}_2[\Box]$ and between D'_1 and D'_2, for suitably-shaped contexts. This result enables generalizing concrete distances to arbitrary inputs. For example, the above two analyses can be generalized and combined to show that the distance between derivations of `mapA` with inputs that differ by one element is constant. This allows us to also derive asymptotic complexity bounds, which is generally difficult with concrete cost semantics (Section 4).

3. The Source Language (Src)

The Src language is a simply-typed, call-by-value λ-calculus with recursive functions and ML-style references. The language also includes natural numbers for didactic purposes and can easily be extended with products, sums, recursive types, etc., but we omit them

[3] Note that not all substitutions yield well-formed derivations. In particular, the choice of locations needs to be consistent.

$$\frac{}{\_;\sigma;v \Downarrow \sigma;v;\varepsilon;0}$$

$$\frac{\mathcal{E};\sigma;e_{\mathrm{z}} \Downarrow \sigma';v';T;c}{\mathcal{E};\sigma;\mathbf{caseN}\,\mathbf{zero}\,e_{\mathrm{z}}\,(x.e_{\mathrm{s}}) \Downarrow \sigma';v';T;c}$$

$$\frac{\mathcal{E};\sigma;\{v_{\mathrm{n}}\,/\,x\}\,e_{\mathrm{s}} \Downarrow \sigma';v';T;c}{\mathcal{E};\sigma;\mathbf{caseN}\,(\mathbf{succ}\,v_{\mathrm{n}})\,e_{\mathrm{z}}\,(x.e_{\mathrm{s}}) \Downarrow \sigma';v';T;c}$$

$$\frac{\begin{array}{l}\mathcal{E}[\Box\,e_{\mathrm{x}}];\sigma;e_{\mathrm{f}} \Downarrow \sigma_{\mathrm{f}};\mathbf{fun}\,f.x.e;T_{\mathrm{f}};c_{\mathrm{f}}\\ \mathcal{E}[(\mathbf{fun}\,f.x.e')\,\Box];\sigma_{\mathrm{f}};e_{\mathrm{x}} \Downarrow \sigma_{\mathrm{x}};v_{\mathrm{x}};T_{\mathrm{x}};c_{\mathrm{x}}\\ \mathcal{E};\sigma_{\mathrm{x}};\{v_{\mathrm{x}}\,/\,x\}\,\{\mathbf{fun}\,f.x.e\,/\,f\}\,e \Downarrow \sigma';v';T;c\end{array}}{\mathcal{E};\sigma;e_{\mathrm{f}}\,e_{\mathrm{x}} \Downarrow \sigma';v';T_{\mathrm{f}}\cdot T_{\mathrm{x}}\cdot(\mathtt{M}_{\mathcal{E}}^{(\mathbf{fun}\,f.x.e)\ v_{\mathrm{x}}\Downarrow v'}(T)\cdot\varepsilon);c_{\mathrm{f}}+c_{\mathrm{x}}+1+c}$$

$$\frac{\ell \notin \mathrm{dom}(\sigma) \quad \sigma' = \sigma \uplus \{\ell \mapsto v\}}{\mathcal{E};\sigma;\mathbf{put}\,v \Downarrow \sigma';\ell;\mathtt{P}_{\mathcal{E}}^{v\uparrow\ell}\cdot\varepsilon;1}$$

$$\frac{\ell \in \mathrm{dom}(\sigma) \quad \sigma(\ell) = v}{\mathcal{E};\sigma;\mathbf{get}\,\ell \Downarrow \sigma;v;\mathtt{G}_{\mathcal{E}}^{\ell\rightarrow v}\cdot\varepsilon;1}$$

$$\frac{\ell \in \mathrm{dom}(\sigma) \quad \sigma' = \sigma[\ell \mapsto v]}{\mathcal{E};\sigma;\mathbf{set}\,\ell\,v \Downarrow \sigma';\mathbf{zero};\mathtt{S}_{\mathcal{E}}^{\ell\leftarrow v}\cdot\varepsilon;1}$$

Figure 3. Src evaluation $\mathcal{E};\sigma;e \Downarrow \sigma';v';T';c$.

as they provide no additional insight. Although Src has no operational support for self-adjusting computation (*i.e.*, a mechanism for updating a computation under input changes), its dynamic semantics produces an *execution trace* that can be used to quantify similarities between runs as a *distance*. Src programs can be compiled into Tgt programs (see Sections 5 and 6), whose semantics include a *change propagation* judgement that realizes updates and asymptotically matches Src distances.

The syntax of Src is given below, which defines types τ, expressions e, and values v, using metavariables f and x for identifiers and ℓ for locations.

$$\begin{array}{l}\tau ::= \mathbf{nat} \mid \tau_x \rightarrow \tau \mid \tau\ \mathbf{ref}\\ e ::= v \mid \mathbf{caseN}\,v_n\,e_z\,(x.e_s) \mid e_f\,e_x \mid \mathbf{put}\,v \mid \mathbf{get}\,v_\ell \mid \mathbf{set}\,v_\ell\,v\\ v ::= x \mid \mathbf{zero} \mid \mathbf{succ}\,v \mid \mathbf{fun}\,f.x.e \mid \ell\end{array}$$

The dynamic semantics of *memoizing* functions $\mathbf{fun}\,f.x.e$ is instrumented to identify opportunities for computation reuse. The reference primitives and scrutinee of $\mathbf{caseN}$ are restricted to value forms for technical simplicity. This restriction can be avoided with syntactic sugar, for example the unrestricted dereference form $\mathbf{get}\,e_\ell$ can be defined as $(\mathbf{fun}\,f.x.\mathbf{get}\,x)\,e_\ell$.

3.1 Static, Dynamic, and Cost Semantics

The (standard, hence omitted) typing judgement $\Sigma;\Gamma \vdash e : \tau$ ascribes the type τ to the expression e in the store and variable typing contexts Σ and Γ. Figure 3 gives the dynamic and cost semantics of Src. The large-step evaluation relation $\mathcal{E};\sigma;e \Downarrow \sigma';v';T;c$ reduces expression e in store σ to value v' in updated store σ' and yields an execution trace T and a *cost* c. The trace internalizes the *shape* of an evaluation derivation and will be used to identify the similar computations. The cost internalizes the *size* of a trace and will be used to relate the constant slowdown due to compiling Src programs to Tgt programs. For the present time, we suggest that the reader ignore the highlighted evaluation context $\mathcal{E}$ component; it is crucial for relating Src and Tgt distances (see Section 6), but is not necessary for reasoning about Src distance.

We distinguish *active* computation as work that may be used to identify similarities and differences in computation. Evaluation of reference primitives and application of memoizing functions yield active computation. Case-analysis and (in the presence of sums, products, *etc.*) other forms of β-reduction are considered *passive* computation. An evaluation derivation internalizes its *size* in a *cost* c as a natural number that quantifies active work. We do not explicitly quantify passive work because it is always bounded by a constant multiple of active work. Intuitively, since a Src program can only perform a bounded amount of computation between function calls, memoizing function actions suffice to account for all passive work; including actions for passive work (*i.e.* case-analysis) would give a more accurate measure but isn't necessary for calculating asymptotic time complexity or distance. This property is formalized in the companion technical report (Ley-Wild et al. 2008a).

A *trace* T is an interleaving of *actions* that internalizes the *shape* of an evaluation derivation:

$$\begin{array}{l}A_{\mathrm{s}} ::= \mathtt{P}_{\mathcal{E}}^{v\uparrow\ell} \mid \mathtt{G}_{\mathcal{E}}^{\ell\rightarrow v} \mid \mathtt{S}_{\mathcal{E}}^{\ell\leftarrow v}\\ A ::= A_{\mathrm{s}} \mid \mathtt{M}_{\mathcal{E}}^{v_f\ v_x\Downarrow v}(T)\\ T ::= \varepsilon \mid A\cdot T\end{array}$$

Actions A serve as markers for active work and consists of store actions and memoizing function actions. *Store actions* A_{s} include allocation ($\mathtt{P}$), dereference ($\mathtt{G}$), and update ($\mathtt{S}$), which are labeled with the location ℓ and value v involved in each operation. A *memoizing function action* $\mathtt{M}_{\mathcal{E}}^{v_f\ v_x\Downarrow v}(T)$ is labeled with a function v_f, argument v_x, and result v; the delimited trace T identifies the body of the function application for reuse; as in the dynamic semantics, the highlighted evaluation context $\mathcal{E}$ can be ignored.

Traces facilitate identifying the similarities and differences between different runs of a program. More specifically, since store mutation is the only kind observable side effect in Src, reference primitives uniquely determine the control flow of a closed program. Thus, by recording them in the trace, we can identify where program runs differ. Since memoizing functions can be used to identify explicitly similar computations by matching arguments to function calls, they can be used to identify where program runs perform similar computations. Therefore actions in traces are necessary and sufficient to isolate the similarities and differences between program runs.

Returning to the dynamic semantics (Figure 3), evaluation extends the trace and increments the cost counter according to the kind of reduction. Cost grows in lock-step with the trace and could be defined as the "size" of the trace, but we keep it explicit to relate the intensional semantics of the Src and Tgt languages. A value reduces to itself, produces an empty trace and has no cost. A case-analysis reduces according to the branch prescribed by the scrutinee; the trace and cost are unchanged, since, as noted above, case-analysis incurs only passive work.

A function application reduces the function e_f and argument e_x to values and then evaluates the redex. An application concatenates the function, argument, and redex traces to represent the sequencing of work; the redex trace is delimited by the memoizing function action to identify the scope of the function call; the cost of the traces are added and incremented by a unit of work for the β-reduction.

A reference allocation extends the store with a fresh location that is initialized with the specified value and returns the location. A dereference returns the location's value. An update changes the location's contents and returns $\mathbf{zero}$. In each case, the trace is the singleton action corresponding to the primitive, and the work is 1.

3.2 Trace Distance

Consider running a single program under two different stores: intuitively, the executions will be identical up to the first differing store primitive (viz. the run of `mapA` on the prefix `...,0,1` from Section 2), after which the traces may correspond to different subprograms (*i.e.*, because an allocation produced different locations or a read found different values). In terms of traces, they will have a common prefix up to the first differing store action. Conservatively, the only similarity between two runs would be the common

$$\frac{}{\varepsilon \boxminus \varepsilon = \langle 0,0\rangle}\ \textbf{search/nil} \qquad \frac{T_1 \ominus T_2 = d \quad T_1' \boxminus T_2' = d'}{\mathtt{M}^{v_f\ v_x \Downarrow v_1}_{\mathcal{E}_1}(T_1)\cdot T_1' \boxminus \mathtt{M}^{v_f\ v_x \Downarrow v_2}_{\mathcal{E}_2}(T_2)\cdot T_2' = \langle 1,1\rangle + d + d'}\ \textbf{search/synch}$$

$$\frac{T_1\cdot T_1' \boxminus T_2 = d}{\mathtt{M}^{v_f\ v_x \Downarrow v}_{\mathcal{E}}(T_1)\cdot T_1' \boxminus T_2 = \langle 1,0\rangle + d}\ \textbf{search/memo/L} \qquad \frac{T_1 \boxminus T_2\cdot T_2' = d}{T_1 \boxminus \mathtt{M}^{v_f\ v_x \Downarrow v}_{\mathcal{E}}(T_2)\cdot T_2' = \langle 0,1\rangle + d}\ \textbf{search/memo/R}$$

$$\frac{T_1 \boxminus T_2 = d}{A_s\cdot T_1 \boxminus T_2 = \langle 1,0\rangle + d}\ \textbf{search/store/L} \qquad \frac{T_1 \boxminus T_2 = d}{T_1 \boxminus A_s\cdot T_2 = \langle 0,1\rangle + d}\ \textbf{search/store/R}$$

$$\frac{}{\varepsilon \ominus \varepsilon = \langle 0,0\rangle}\ \textbf{synch/nil} \qquad \frac{T_1 \ominus T_2 = d}{A_s\cdot T_1 \ominus A_s\cdot T_2 = d}\ \textbf{synch/store} \qquad \frac{T_1 \boxminus T_2 = d}{T_1 \ominus T_2 = d}\ \textbf{synch/search}$$

$$\frac{T_1 \ominus T_2 = d \quad T_1' \ominus T_2' = d'}{\mathtt{M}^{v_f\ v_x \Downarrow v}_{\mathcal{E}}(T_1)\cdot T_1' \ominus \mathtt{M}^{v_f\ v_x \Downarrow v}_{\mathcal{E}}(T_2)\cdot T_2' = d + d'}\ \textbf{synch/memo}$$

Figure 4. Src search distance $T_1 \boxminus T_2 = d$ (top) and synchronization distance $T_1 \ominus T_2 = d$ (bottom).

prefix. Memoizing functions, however, enable recognizing similar computations that occur *after* two runs have diverged (viz. the run of `mapA` on the postfix `3,4,...`) because they identify the trace of the same function applied to the same argument. Nevertheless, even if two computations result from the same function application, they may also have different traces and return different results due to interactions with the store. More generally, comparing two traces alternates between *searching* for a point where traces align (viz. memoizing function application) and *synchronizing* the two similar traces until they again differ (viz. store actions).

Distance is formally captured by the search distance $T_1 \boxminus T_2 = d$ and synchronization distance $T_1 \ominus T_2 = d$ judgements (given in Figure 4), defined by structural induction on the two traces. The search mode *can* switch to synchronization if it encounters similar program fragments (as identified by memoizing application actions), and the synchronization mode *must* switch to search mode if the trace actions differ at some point. Intuitively, the trace distance measures the symmetric difference between two traces, *i.e.*, the size of trace segments that don't occur in both traces. Concretely, we quantify distance $d = \langle c_1, c_2\rangle$ between traces T_1 and T_2 as a pair of costs, where c_1 is the amount of work in T_1 that isn't shared with T_2 and c_2 is the amount of work in T_2 that isn't shared with T_1. We let $d + d'$ denote pointwise addition for distance.

Since traces approximate the shape of an evaluation derivation, trace distance approximates a (higher-order) distance judgement on evaluation derivations that quantifies the dis/similarities between two runs (modulo the stores). The dynamic semantics of Tgt has nondeterministic allocation and memoization in order to avoid committing to an implementation; consequently, the definition of Src trace distance is a relation, but we will show that any distance derivable for Src programs is preserved in Tgt (Theorem 7).

The search distance $T_1 \boxminus T_2 = d$ accounts for traces that don't match, but switches to synchronization mode if it can align memoization actions. The search distance between empty traces is zero. Upon simultaneously encountering memoization actions $\mathtt{M}^{v_f\ v_x \Downarrow v_1}_{\mathcal{E}_1}(T_1)\cdot T_1'$ and $\mathtt{M}^{v_f\ v_x \Downarrow v_2}_{\mathcal{E}_2}(T_2)\cdot T_2'$ of the same function v_f and argument v_x (**search/synch** rule), the search distance can switch to synchronizing the bodies T_1 and T_2, while separately searching for further synchronization of the tails T_1' and T_2'. The cost of the synchronization and search are added to the cost of 1 for the memoization match in each trace.

Finally, skipping an action in search mode incurs a cost of 1 in addition to the distance between the tail of the trace (**search/memo** rules and **search/store** rules).

Turning to the synchronization distance, the $T_1 \ominus T_2 = d$ judgement attempts to structurally match the two traces. Identical work in both traces incurs no cost, but synchronization returns to search mode either nondeterministically or when work cannot be reused because traces don't match. Synchronization mode is only meant to be used on traces generated by the evaluation of the same expression under (possibly) different stores.

The synchronization distance between empty traces is zero. Encountering identical store actions allows distance to remain in synchronization mode without cost (**synch/store** rule). Synchronizing a memoization action (**synch/memo** rule) requires the function call (function v_f and argument v_x) and return (result v) to match; this allows the bodies as well as the tails to be synchronized separately and their distance compounded. Note that even if the bodies don't match completely and return to search mode, memoizing functions provide a degree of isolation because tails can be matched independently. Synchronization falls back to search mode (**synch/search** rule) nondeterministically or necessarily when the actions are distinct (*i.e.*, because store or memo actions don't match).

Observe that the definition of synchronization distance is quasi-symmetric: $T_1 \ominus T_2 = \langle c_1, c_2\rangle$ iff $T_2 \ominus T_1 = \langle c_2, c_1\rangle$; similarly for search distance. Furthermore, note that distance of Src programs is defined by induction on the two traces: both judgements traverse traces left-to-right either matching work or accounting for skipping it. This means that common work consists of a subsequence of actions that appears in both traces, which precludes reordering work. For example, comparing runs $\mathtt{M}^{f\ x \Downarrow a}(\_)\cdot \mathtt{M}^{g\ y \Downarrow b}(\_)\cdot\_$ and $\mathtt{M}^{g\ y \Downarrow b}(\_)\cdot \mathtt{M}^{f\ x \Downarrow a}(\_)\cdot\_$ can only synchronize one of the calls, the other call must be skipped. This restriction avoids having to search for permutations for matching computations and simplifies the implementation requirements of Tgt; however, the limitation obviously hinders the efficiency of self-adjusting computation for certain classes of computations.

3.3 Trace Contexts

In order to reason compositionally about distance, we define a *trace context* $\mathscr{T}$ to be a trace with a hole; the formalization to multi-holed contexts is straightforward and omitted for reasons of space.

$$\mathscr{T} ::= \square \mid \mathtt{M}^{v_f\ v_x \Downarrow v}_{\mathcal{E}}(\mathscr{T})\cdot T \mid \mathtt{M}^{v_f\ v_x \Downarrow v}_{\mathcal{E}}(T)\cdot \mathscr{T} \mid A_s\cdot \mathscr{T}$$

Trace context distances $\mathscr{T}_1 \boxminus \mathscr{T}_2 = d$ and $\mathscr{T}_1 \ominus \mathscr{T}_2 = d$ are obtained by lifting distance on traces to trace contexts, extended with the following rules for holes (in the multi-hole analogue, holes are uniquely labeled and labels must also coincide):

$$\frac{}{\square \boxminus \square = \langle 0,0\rangle} \qquad \frac{}{\square \ominus \square = \langle 0,0\rangle}$$

By requiring holes to coincide when comparing trace contexts, we can reason separately about context and trace distance, and then combine the results. Intuitively, the identity theorem means a common suffix subcomputation incurs no cost. The substitution theorem shows that a common prefix computation does not affect distance either: provided the hole in both trace contexts is immediately bounded by a memoization action of the same function and argument, context and trace distance can be combined additively. Both theorems are proved by induction on the subject trace T.

Theorem 1 (Identity for Traces)
For any trace T, $T \ominus T = \langle 0, 0\rangle$.

Theorem 2 (Identity for Trace Contexts)
For any trace context $\mathscr{T}$,
$\mathscr{T}[\mathtt{M}_{\mathcal{E}}^{v_f\ v_x \Downarrow v}(\square)\cdot T] \ominus \mathscr{T}[\mathtt{M}_{\mathcal{E}}^{v_f\ v_x \Downarrow v}(\square)\cdot T] = \langle 0, 0\rangle$.

Theorem 3 (Substitution)
If $\mathscr{T}_1[\mathtt{M}_{\mathcal{E}_1}^{v_f\ v_x \Downarrow v_1}(\square)\cdot T_1] \boxminus \mathscr{T}_2[\mathtt{M}_{\mathcal{E}_2}^{v_f\ v_x \Downarrow v_2}(\square)\cdot T_2] = d$
and $T'_1 \ominus T'_2 = d'$,
then $\mathscr{T}_1[\mathtt{M}_{\mathcal{E}_1}^{v_f\ v_x \Downarrow v_1}(T'_1)\cdot T_1] \boxminus \mathscr{T}_2[\mathtt{M}_{\mathcal{E}_2}^{v_f\ v_x \Downarrow v_2}(T'_2)\cdot T_2] = d + d'$.
If $\mathscr{T}_1[\mathtt{M}_{\mathcal{E}_1}^{v_f\ v_x \Downarrow v_1}(\square)\cdot T_1] \ominus \mathscr{T}_2[\mathtt{M}_{\mathcal{E}_2}^{v_f\ v_x \Downarrow v_2}(\square)\cdot T_2] = d$
and $T'_1 \ominus T'_2 = d'$,
then $\mathscr{T}_1[\mathtt{M}_{\mathcal{E}_1}^{v_f\ v_x \Downarrow v_1}(T'_1)\cdot T_1] \ominus \mathscr{T}_2[\mathtt{M}_{\mathcal{E}_2}^{v_f\ v_x \Downarrow v_2}(T'_2)\cdot T_2] = d + d'$.
Proof: By simultaneous induction on the first derivations. ■

3.4 Trace Distance, Revisited

The rules of Figure 4 are useful for high level reasoning, but aren't rich enough to demonstrate a correspondence with Tgt trace distance. We present an alternate rule system that subsumes the above system and serves as an intermediary for proving the preservation of distance under compilation.

Failure Actions. The **search/synch** rule separately synchronizes the bodies and searches the tails when it encounters matching memoizing actions. While this rule is useful, it precludes memoization between one body and another tail; for example, it doesn't allow splitting T_1 as $T_{11}\cdot T_{12}$ and synchronizing T_{11} with a prefix of T_2 and searching T_{12} against the rest of T_2. The naïve rule

$$\frac{T_1\cdot T'_1 \ominus T_2\cdot T'_2 = d}{\mathtt{M}_{\mathcal{E}_1}^{v_f\ v_x \Downarrow v_1}(T_1)\cdot T'_1 \boxminus \mathtt{M}_{\mathcal{E}_2}^{v_f\ v_x \Downarrow v_2}(T_2)\cdot T'_2 = \langle 1, 1\rangle + d}$$

would allow splitting both traces, but it is unsound because it may fully synchronize $T_1\cdot T'_1$ with $T_2\cdot T'_2$, even though the trace concatenation may *not* have been generated the same expression, violating the well-formedness of synchronization distance. We remedy this by introducing the failure action $\mathtt{F}_{\mathcal{E}}^{\Downarrow v}$ to explicitly force synchronization mode to switch back to search mode; it is labeled by an evaluation context $\mathcal{E}$ and result v, which are needed for technical reasons but can be ignored when reasoning about Src distance:

$$A ::= \cdots \mid \mathtt{F}_{\mathcal{E}}^{\Downarrow v}$$

The revised system is obtained by removing the **search/synch** and **search/memo** rules from Figure 4 and adding the following:

$$\frac{T_1\cdot \mathtt{F}_{\mathcal{E}}^{\Downarrow v}\cdot T'_1 \boxminus T_2 = d}{\mathtt{M}_{\mathcal{E}}^{v_f\ v_x \Downarrow v}(T_1)\cdot T'_1 \boxminus T_2 = \langle 1, 0\rangle + d}\ \textbf{search/memo'/L}$$

$$\frac{T_1 \boxminus T_2\cdot \mathtt{F}_{\mathcal{E}}^{\Downarrow v}\cdot T'_2 = d}{T_1 \boxminus \mathtt{M}_{\mathcal{E}}^{v_f\ v_x \Downarrow v}(T_2)\cdot T'_2 = \langle 0, 1\rangle + d}\ \textbf{search/memo'/R}$$

$$\frac{T_1 \boxminus T_2 = d}{\mathtt{F}_{\mathcal{E}}^{\Downarrow v}\cdot T_1 \boxminus T_2 = d}\ \textbf{search/fail/L}$$

$$\frac{T_1 \boxminus T_2 = d}{T_1 \boxminus \mathtt{F}_{\mathcal{E}}^{\Downarrow v}\cdot T_2 = d}\ \textbf{search/fail/R}$$

$$\frac{T_1\cdot \mathtt{F}_{\mathcal{E}_1}^{\Downarrow v_1}\cdot T'_1 \ominus T_2\cdot \mathtt{F}_{\mathcal{E}_2}^{\Downarrow v_2}\cdot T'_2 = d}{\mathtt{M}_{\mathcal{E}_1}^{v_f\ v_x \Downarrow v_1}(T_1)\cdot T'_1 \boxminus \mathtt{M}_{\mathcal{E}_2}^{v_f\ v_x \Downarrow v_2}(T_2)\cdot T'_2 = \langle 1, 1\rangle + d}\ \textbf{search/synch'}$$

The new **search/memo'** rules insert an explicit failure action between the body and tail of a memoization action, and still incur a cost of 1 for failing to match. The **search/fail** rules allow search to skip a failure action without cost. Observe that, in Figure 4, a trace is subjected to synchronization if it is delimited by a memoization action and failure actions never occur in the scope of a memoization action, so failure actions never appear in synchronization mode. Therefore the **search/memo'** and **search/fail** rules subsume the (replaced) **search/memo** rules: any distance derivable from the failure-free deductive system is also derivable from the system with explicit failure.

The **search/synch'** rule identifies matching function applications and switches to synchronizing the concatenation of the body, failure action, and tail. Since there are no new synchronization distance rules, leading failure actions force synchronization to switch to search (only the **synch/search** rule applies). Therefore the **search/synch'** rule enables synchronizing part of T_1 with T_2 and then searching the remainder of T_1 against T'_2 (after encountering the failure action between T_2 and T'_2). The **search/synch'** rule subsumes the (replaced) **search/sync** rule.

Evaluation Contexts. The *evaluation contexts* $\mathcal{E}$ in Src evaluation and traces are necessary for relating Src and Tgt traces in Section 6, but can be ignored when reasoning about Src evaluation and distance (in the deductive systems with and without failure). An evaluation context is built up throughout evaluation (Figure 3) to capture the shape of the surrounding evaluation derivation, up to the nearest memoizing function application:

$$\mathcal{E} ::= \square \mid \mathcal{E}\, e_x \mid v_f\, \mathcal{E}$$

The language restriction on the occurrence of expressions avoids explicit forms for case-analysis or reference manipulation. The evaluation of a memoizing function application extends the context for evaluating the function and argument expressions, but *resets* the context for evaluating the redex; passive β-reduction (*i.e.*, case-analysis) passes the context unchanged. The accumulated context is used to label the actions with the current context and is used by the ACPS trace translation to reify the continuation.

Intuitively, contexts help identify if computation *after* a memoizing function application can be reused. The **search/synch** rule ignores the contexts of each trace, the **search/memo** rules pass the context and result to the failure action. The **synch/store** and **synch/memo** rules formally require the contexts to be identical. Since synchronization begins at memoizing actions $\mathtt{M}_{\mathcal{E}_1}^{v_f\ v_x \Downarrow v_1}(T_1)$ and $\mathtt{M}_{\mathcal{E}_2}^{v_f\ v_x \Downarrow v_2}(T_2)$ (cf. **search/synch**), the bodies T_1 and T_2 result from the evaluation of the *same* expression in the *same* reset context (cf. application evaluation) but under (possibly) different stores. Synchronization distance inductively preserves the property that the two traces being compared result from the *same* expression in the *same* context. In particular, note that the evaluation contexts and results match in the **synch/memo** rule, so the property holds of the tails, which justifies why they can be synchronized independently of the bodies. Therefore, contexts in synchronization mode are necessarily equal, and can be ignored when reasoning about Src distance.

4. Examples

We consider several examples to show how we can use trace distance to analyze the sensitivity of programs to small changes

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{T_0 \ominus T_0 = 0 \qquad \mathtt{P}^{\ell_b\uparrow\mathtt{b}::\ell_c}\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_b} \boxminus \mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_b} = \langle 2,1\rangle}{\mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_b\uparrow\mathtt{b}::\ell_c}\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_b} \boxminus \mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_c} = \langle 3,2\rangle}}{\mathtt{G}^{\ell_2\to\mathtt{2}::\ell_3}\cdot A^{\mathtt{2}\Downarrow\mathtt{b}}\cdot\mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_b\uparrow\mathtt{b}::\ell_c}\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_b} \boxminus \mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_c} = \langle 5,2\rangle}}{\mathtt{M}^{\ell_2\Downarrow\ell_b}(\mathtt{G}^{\ell_2\to\mathtt{2}::\ell_3}\cdot A^{\mathtt{2}\Downarrow\mathtt{b}}\cdot\mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_b\uparrow\mathtt{b}::\ell_c})\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_b} \boxminus \mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_c} = \langle 7,3\rangle}}{\mathtt{G}^{\ell_1\to\mathtt{1}::\ell_2}\cdot A^{\mathtt{1}\Downarrow\mathtt{a}}\cdot\mathtt{M}^{\ell_2\Downarrow\ell_b}(\mathtt{G}^{\ell_2\to\mathtt{2}::\ell_3}\cdot A^{\mathtt{2}\Downarrow\mathtt{b}}\cdot\mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_b\uparrow\mathtt{b}::\ell_c})\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_b} \ominus \mathtt{G}^{\ell_1\to\mathtt{1}::\ell_3}\cdot A^{\mathtt{1}\Downarrow\mathtt{a}}\cdot\mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_c} = \langle 8,4\rangle}}{\mathtt{M}^{\ell_1\Downarrow\ell_a}(\mathtt{G}^{\ell_1\to\mathtt{1}::\ell_2}\cdot A^{\mathtt{1}\Downarrow\mathtt{a}}\cdot\mathtt{M}^{\ell_2\Downarrow\ell_b}(\mathtt{G}^{\ell_2\to\mathtt{2}::\ell_3}\cdot A^{\mathtt{2}\Downarrow\mathtt{b}}\cdot\mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_b\uparrow\mathtt{b}::\ell_c})\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_b}) \boxminus \mathtt{M}^{\ell_1\Downarrow\ell_a}(\mathtt{G}^{\ell_1\to\mathtt{1}::\ell_3}\cdot A^{\mathtt{1}\Downarrow\mathtt{a}}\cdot\mathtt{M}^{\ell_3\Downarrow\ell_c}(T_0)\cdot\mathtt{P}^{\ell_a\uparrow\mathtt{a}::\ell_c}) = \langle 9,5\rangle}
$$

Figure 5. Trace distance between `mapA [1,2,3]` and `mapA [1,3]`.

in their input. We say that a program is $O(f(n))$-*sensitive* or $O(f(n))$-*stable* for an input change if the distance between the traces of that program is bounded by $O(f(n))$ for inputs related by that change. In our analysis, we consider two kinds of changes: insertions/deletions that relate lists that differ by the existence of an element (*e.g.*, `[1,3]` and `[1,2,3]`) and replacements that relate inputs that differ by the value of one element (*e.g.*, `[1,2,3]` and `[1,7,3]`). We start with the `map` example that we considered informally (Section 2) and analyze its sensitivity to an insertion/deletion into/from the input by comparing its traces. When convenient, we visualize traces as derivations and analyze their relative distance under a replacement.

In our analysis, we consider two kinds of bounds: upper bounds and lower bounds. Our upper bounds state that the distance between the traces of a program with inputs related by some change can be asymptotically bounded by some function of the input size under the assumption that locations allocated in the computation (or mentioned in the trace) can be chosen to match nicely. Without the ability to match locations, it is not possible to prove interesting upper bounds, because two runs of the program can differ by as much as the size of the traces if their locations are chosen from disjoint sets. As we discuss in Section 7, an implementation can often match locations, sometimes with programmer guidance. Our lower bounds state that the distance between traces of a program with inputs related by some change cannot be asymptotically smaller than a function of input size regardless of how we choose locations. Such lower bounds suggest but do not prove a lower bound on the running time for change propagation (Section 7).

```
fun reduce f id l =
let fun red r l =
      case !l of
        nil => ref r
      | h::t => red (f(h,r)) t
in red id l end

fun reducePair f id l =
let fun comp l =
      case !l of
        nil => ref nil
      | a::t => case !t of
        nil => ref (a::ref nil)
        | b::u => ref (f(a,b)::(comp u))

    fun rec l =
      if !(lenLT (l,2)) then case !l of
                               nil => id
                             | h::_ => h
      else rec (comp l)
in rec l end

fun msort l =
  if !(lenLT (l,2)) then l
  else let (a,b) = partition l
           sa = msort a
           sb = msort b
       in merge (sa,sb) end

fun filter f l =
  case !l of
    nil => ref nil
    h::t => if (f h) then h::(filter f t)
            else filter f t
```

Figure 6. Code for the examples.

Our analyses fit into one of the following patterns. Sometimes, we start with two concrete inputs and show a bound on the distance between traces with these inputs. We then generalize this bound to arbitrary inputs using the identity and substitution theorems (Theorems 1 and 3). Other times, using the identity and the substitution theorems, we write a recursive formula for the distance between the traces of the program with inputs related by some change, and solve this formula to establish the bound. When analyzing our examples and using the identity and the substitution theorems, we ignore contexts, because, as noted in Section 3, they are not needed for analysis. We use the distance and the composition theorems in the informal style of traditional algorithmic analysis, because we have no meta-logical framework for reasoning about asymptotic properties of self-adjusting programs Section 7.

Figure 6 shows the code for list-reduction and merge-sort (see Section 2 for the code of `map`). The list-reduce and merge-sort implementations use several functions, whose code we omit for brevity. The `lenLT(l,i)` function returns (in a reference) `true` iff the length of the list `l` is less than the integer `i`. The `partition` function evenly splits a list into two and `merge` combines two sorted lists. All of these functions are $O(1)$-sensitive to replacements on average (for `merge`, we need to average over all permutations of the input to obtain this bound). To focus on the main ideas, we omit the analysis of these utility functions here, which are similar to that of the `map` function discussed below.

4.1 Map

Recall the `mapA` function from Section 2. We analyze the sensitivity of `mapA` to an insertion/deletion more precisely by using trace distance. Figure 5 shows the derivation of the trace distance for `mapA` with inputs `L = [1,2,3]` and `L' = [1,3]`. We consider derivations where the input locations are $\ell_1, \ell_2, \ell_3, \ell_4$ and the output locations are $\ell_a, \ell_b, \ell_c, \ell_n$. In the derivations we use the notation $\mathtt{M}^{\ell\Downarrow\ell'}(T)$ as a shorthand for the memoization action $\mathtt{M}^{\mathtt{mapA}\;\ell\Downarrow\ell'}(T)$. Similarly we write $A^{\mathtt{x}\Downarrow\mathtt{y}}$ as a shorthand for the memoization action $\mathtt{M}^{f\;x\Downarrow y}(\_)$ of the function f mapping integer x to letter y, whose subtrace (body) we leave unspecified and assume to be of length constant (it contributes one to the distance). We define the tail trace T_0 common to both executions as $\mathtt{G}^{\ell_3\to 3::\ell_4}\cdot A^{\mathtt{3}\Downarrow\mathtt{c}}\cdot\mathtt{M}^{\ell_4\Downarrow\ell_n}(\mathtt{G}^{\ell_4\to\mathtt{nil}}\cdot\mathtt{P}^{\ell_n\uparrow\mathtt{nil}})\cdot\mathtt{P}^{\ell_c\uparrow c::\ell_d}$. When deriving the distance, we combine consecutive applications of the same rule and use the fact that the synchronization distance between a trace and itself is zero, *i.e.*, $T \ominus T = 0$ (Theorem 1).

Having derived a constant bound for this example, we can generalize the result to obtain an asymptotic bound for a change in one element in the middle of an arbitrary list. Consider the traces T_1 and T_2 for `mapA(`L_1`)` and `mapA(`L_2`)` where L_1 = `[`x`]` and L_2 = `[]`. The the distance between them is trivially constant for any x. We will now use the substitution theorem to generalize this result to arbitrary lists by showing how to extend the inputs lists with identical prefixes and suffixes without affecting the constant bound.

We consider extending the input with the same suffix. We start by replacing each of the sub-traces of the form $\mathtt{M}^{\text{-}\Downarrow\text{-}}(\_)$ for the rightmost call to `mapA` in T_1 and T_2 with a hole to obtain the trace contexts $\mathscr{T}_1$ and $\mathscr{T}_2$. Let L_3 be any list and let T_3 be the trace for `mapA(`L_3`)`. Note that the traces $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$ are the traces for `mapA(`$L_1@L_3$`)` and `mapA(`$L_2@L_3$`)`. By the identity theorem, the distance between T_3 and T_3 is zero. Since T_3 starts with memoization action of the form $\mathtt{M}^{l_i \Downarrow l_\alpha}(\ldots)$, we can apply the substitution theorem, so the distance between $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$ is equal to the distance between $\mathscr{T}_1[\mathtt{M}^{l_i \Downarrow l_\alpha}(\square)]$ and $\mathscr{T}_3[\mathtt{M}^{l_i \Downarrow l_\alpha}(\square)]$, which is constant. Thus, we are able to append any suffix to L_1 and L_2 without increasing their distance.

Symmetrically, we can extend these lists with the same prefix. To see this, let L_0 be a list and consider its trace T_0 with `mapA`. Now define the trace context $\mathscr{T}_0$ as the context obtained by replacing the rightmost sub-trace in T_0 of the form $\mathtt{M}^{\text{-}\Downarrow\text{-}}(\_)$ with a hole. Now, substitute into this trace the traces $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$, i.e., $\mathscr{T}_0[\mathscr{T}_1[T_3]]$ and $\mathscr{T}_0[\mathscr{T}_2[T_3]]$. By the identity and the substitution theorems, the distance is equal to distance between of $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$, which is constant.

Thus, we can generalize the simple examples with lists L_1 and L_2 to other lists by prepending and appending arbitrary lists, essentially obtaining any two lists related by an insertion/deletion. Thus, we conclude that `mapA` is constant sensitive for an insertion into/deletion from its input.

4.2 Reduce

The list-reduce function reduces a list to a value by applying a given binary operator with a specified identity element to the elements of the list. The standard accumulator-based implementation, `reduce: ('a * 'a -> 'a) -> 'a -> 'a list -> 'a ref` shown in Figure 6, is not amenable to self-adjusting computation, because the distance can be as large as linear. To see this note that all intermediate updates of the accumulator depend on the previously-seen elements. Thus replacing the first element will prevent all derivation steps from matching, causing the distance to be linear in the size of the input (in the worst case).

Figure 6 shows another implementation for list-reduce, called `reducePair`. This implementation applies the function `comp` repeatedly until the list is reduced to contain at most one element. Function `comp` pairs the elements of the input list from left to right and applies `f` to each pair reducing the size of the input list by half. Thus, `comp` is called a logarithmic number of times. Using the shorthand $\mathtt{chk}(\ell) \Downarrow v$ for derivations of the form $\mathtt{lenLT}(\ell) \Downarrow b \;\; \mathtt{G}^{b\to v}$, the derivations for `reducePair` can be represented with the following derivation context.

$$\dfrac{\dfrac{\mathtt{chk}(\ell)\Downarrow \mathtt{F} \qquad \overset{\mathtt{comp}(\ell)\Downarrow\ell_1}{\bigtriangledown} \qquad \overset{\mathtt{rec}(\ell_1)\Downarrow r_1}{\bigtriangledown}}{\mathtt{rec}(\ell)\Downarrow r_1}}{\mathtt{reducePair}\,(f,id,\ell)\Downarrow r_1}$$

To analyze the sensitivity of `reducePair` for a replacement operation, consider evaluating `reducePair` with two lists related by a replacement. The recursive case for the derivations both fit the derivation context given above. Note that the derivations for `comp` are related by a replacement. Since a replacement in the input causes the output of `comp` to change by a replacement as well, the recursive calls to `rec` are related by a replacement as well. Furthermore, since the derivation for `comp` and `rec` both start with memoized functions, we can apply the substitution theorem assuming that the `comp` returns its output in the same location. More precisely, we can write the sensitivity of `rec` to a replacement for an input size of n as

$$\Delta_{\mathtt{rec}}(n) = \begin{cases} \Delta_{\mathtt{rec}}(n/2) + \Delta_{\mathtt{comp}}(n/2) & \text{if } n > 1 \\ 1 & \text{otherwise.} \end{cases}$$

Since `comp` uses an element of the input list to produce just one of the output elements, it is relatively easy to show that is is $O(1)$ sensitive to replacement when `f` is $O(1)$, *i.e.*, $\Delta_{\mathtt{comp}}(m) = O(1)$ for any m. By straightforward arithmetic, we conclude that $\Delta_{\mathtt{rec}}(n) = O(\log n)$. Since `reducePair` simply calls `rec` this implies that `reducePair` has logarithmic sensitivity to a replacement.

4.3 Merge sort

We analyze the sensitivity of the merge-sort algorithm to replacement operations. The recursive case for the derivations of `msort` with inputs that differ in one element, fit the following derivation context (function names are abbreviated).

$$\dfrac{\overset{\mathtt{len}(\ell)\Downarrow b}{\bigtriangledown} \qquad \mathtt{G}^{b\to\mathtt{F}} \qquad \overset{\mathtt{part}(\ell)\Downarrow(\ell_a,\ell_b)}{\bigtriangledown} \qquad \overset{\mathtt{ms}(\ell_a)\Downarrow\ell_c}{\bigtriangledown} \qquad \overset{\mathtt{ms}(\ell_b)\Downarrow\ell_d}{\bigtriangledown} \qquad \overset{\mathtt{mg}(\ell_c,\ell_d)\Downarrow\ell'}{\bigtriangledown}}{\mathtt{ms}\,(\ell)\Downarrow\ell'}$$

The derivation starts with a check for the length of the list being greater than one. In the recursive case, the list has more than one element so the `lenLT` function returns `false`. Thus, we `partition` the input lists into two lists ℓ_a and ℓ_b of half the length, sort them to obtain ℓ_c and ℓ_d, and merge the sorted lists. Since both evaluations can be derived from this context, the distance between the derivations is the distance between the derivations substituted for the holes in the context.

Consider the derivations substituted for each hole. Since `lenLT` and `part` are called with the input, the derivations for $\mathtt{lenLT}(\ell_1)$ (and $\mathtt{part}(\ell_1)$) are related by replacement. As a result, one of ℓ_a or ℓ_b are also related by replacement. Thus only one of the derivations $\mathtt{ms}(\ell_a)$ or $\mathtt{ms}(\ell_b)$ are related by replacement and the other pair is identical. Consequently $\mathtt{mg}(\ell_c,\ell_d)$ derivations are related by replacement. Since all contexts belong to memoized function calls, we can apply the substitution theorem by assuming that all related and identical functions calls in both evaluations return their results in the same locations. Thus, we can write the sensitivity of `msort` as $\Delta_{\mathtt{msort}}(n) = 2\Delta_{\mathtt{msort}}(n/2) + \Delta_{\mathtt{partition}}(n) + \Delta\mathtt{merge}(n)$. It is easy to show that `partition` and `lenLT` functions are $O(1)$ sensitive to replacements. Similarly, we can show that `merge` is $O(1)$ sensitive to replacements on average, if we take averages over all permutations of the input list. Thus, we obtain

$$\Delta_{\mathtt{msort}}(n) = \begin{cases} \Delta_{\mathtt{msort}}(n/2) + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

This recurrence trivially is bounded by $1 + 4c\log n$. Thus, we conclude that `msort` is $O(\log n)$-sensitive to replacement operations.

4.4 Filter

As an example of another program that is not naturally stable we consider a standard list filter function `filter`, whose code is shown in Figure 6, for which we prove that there are inputs whose traces are separated by a linear distance in the size of the inputs regardless of the choice of locations. In other words, we will prove a lower bound for the sensitivity of `filter`. The reason for why `filter` is not stable is similar to that of the conventional implementation of `reduce`, which we consider in Section 4.2, but more subtle, because it is primarily determined by choices of locations rather than the computations being performed.

To see why `filter` can be highly sensitive, it suffices to consider a specialization, which we call `filter0`, that filters out the elements that are zero. For example, with input lists L = `[0,0,0]`

and L' = `[0,0,1]`, `filter0` returns `[]` and `[1]`. Since we are interested in proving a lower bound only, we can summarize traces by including function calls and put operations only—the omitted parts of the trace will affect the bound by a constant factor assuming that the filtering functions takes constant time. In particular, using the shorthand $\mathtt{M}^{\ell \Downarrow \ell'}(T)$ for the memoization action $\mathtt{M}^{\mathtt{filter0}\, \ell \Downarrow \ell'}(T)$, the traces for filter with L and L' are respectively:

$$\mathtt{M}^{\ell_1 \Downarrow \ell_n}(\mathtt{M}^{\ell_2 \Downarrow \ell_n}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell_n}(\mathtt{P}^{\ell_n \uparrow \mathtt{nil}})))), \text{ and}$$

$$\mathtt{M}^{\ell_1 \Downarrow \ell_a}(\mathtt{M}^{\ell_2 \Downarrow \ell_a}(\mathtt{M}^{\ell_3 \Downarrow \ell_a}(\mathtt{M}^{\ell_4 \Downarrow \ell_n}(\mathtt{P}^{\ell_n \uparrow \mathtt{nil}}) \cdot \mathtt{P}^{\ell_a \uparrow \mathtt{1} :: \ell_n}))).$$

Note that the distance between these two traces is greater than 3—the length of the input—because in the second trace three memoization actions return the location ℓ_a holding `[1]`, whereas in the first trace ℓ_n is returned. Since these locations are different, the memoization actions do not match and contribute to the distance. This example does not lead to a lower bound, however, because we can give two traces for the considered inputs for which the distance is one, e.g.,

$$\mathtt{M}^{\ell_1 \Downarrow \ell_n}(\mathtt{M}^{\ell_2 \Downarrow \ell_n}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell_n}(\mathtt{P}^{\ell_n \uparrow \mathtt{nil}})))), \text{ and}$$

$$\mathtt{M}^{\ell_1 \Downarrow \ell_n}(\mathtt{M}^{\ell_2 \Downarrow \ell_n}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell'_n}(\mathtt{P}^{\ell'_n \uparrow \mathtt{nil}}) \cdot \mathtt{P}^{\ell_n \uparrow \mathtt{1} :: \ell'_n}))).$$

The idea is to choose the locations in such a way that the traces overlap maximally. It is not difficult to generalize this example for arbitrary lists of the form `[0,...,0,0]` and `[0,...,0,1]`.

We obtain the worst case inputs by modifying this example to prevent location choices from reducing the distance arbitrarily. Consider parameterized lists of the form $L_1(n)$ = `[(0)`^{n}`,0,(0)`^{n}`]` and $L_2(n)$ = `[(0)`^{n}`,1,(0)`^{n}`]`, where 0^n denotes n repeated 0's. We will show that the distance between traces for any two such inputs is at least $n+1$ and thus linear in the size of the input, $2n+1$. For example, the traces for $L_1(1)$= `[0,0,0]` and $L_2(1)$ = `[0,1,0]` have the form

$$\mathtt{M}^{\ell_1 \Downarrow \ell_n}(\mathtt{M}^{\ell_2 \Downarrow \ell_n}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell_n}(\mathtt{P}^{\ell_n \uparrow \mathtt{nil}})))), \text{ and}$$

$$\mathtt{M}^{\ell_1 \Downarrow \ell_a}(\mathtt{M}^{\ell_2 \Downarrow l_a}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell_n}(\mathtt{P}^{\ell_n \uparrow \mathtt{nil}})) \cdot \mathtt{P}^{l_a \uparrow \mathtt{1} :: l_n})).$$

These traces have distance greater than 2. Regardless of how we change the locations this distance will not decrease because the return locations of n memoization actions before and after the occurrence of `1` will have to differ. Thus, regardless of which location the other trace chooses to store the empty list, at least half the calls will have a differing location. We can generalize this example with $n = 3$ to arbitrary lists by using our identity and substitution theorems as we did for the `map` example. Since the approach is essentially the same as with `map`, we leave it out here. Thus, we conclude that `filter` is $\Omega(n)$-sensitive to a replacement.

This example implies that a self-adjusting computation can do poorly with this implementation of `filter`. As with `reduce`, however, we can give a stable implementation of `filter` by using a compress function similar to `comp` of `reducePair` that applies the filter function to half of the remaining unfiltered elements. We can show that this implementation of `filter` achieves $O(\log n)$ stability/sensitivity under particular choices of locations.

5. The Target Language (Tgt)

The Tgt language is a simply-typed, call-by-value λ-calculus with natural numbers and recursive functions, extended with *modifiable references* and a *memoization* primitive. The language is self-adjusting: its semantics includes evaluation and change-propagation judgements that can be used to reduce expressions to values and adapt computations to input changes. Tgt extends the read-only modifiables of (Ley-Wild et al. 2008b) with imperative update, a cost semantics for evaluation and change propagation, and a notion of trace distance.

The syntax of Tgt is given below, which defines types τ, expressions e, values v, and adaptive commands k, using metavariables f and x for identifiers and ℓ for locations.

$$\begin{array}{l}
\tau ::= \mathbf{nat} \mid \tau_x \to \tau \mid \tau\ \mathbf{mod} \mid \mathbf{res} \\
e ::= v \mid \mathbf{caseN}\, v_n\, e_z\, (x.e_s) \mid e_f\, v_x \\
v ::= x \mid k \mid \mathbf{zero} \mid \mathbf{succ}\, v \mid \mathbf{fun}\, f.x.e \mid \ell \\
k ::= \mathbf{putk}\, v\, v_k \mid \mathbf{getk}\, v_\ell\, v_k \mid \mathbf{setk}\, v_\ell\, v\, v_k \mid \mathbf{memo}\, e \mid \mathbf{halt}\, v \\
\lambda\, x.e \stackrel{\text{def}}{=} \mathbf{fun}\, f.x.e \qquad \text{with } f \notin \mathrm{FV}(e)
\end{array}$$

Tgt enforces a continuation-passing style (cps) discipline to help identify opportunities for reuse and computations for re-execution. Adaptive store commands have an explicit continuation v_k identifying the computation that follows the command. The cps discipline also restricts a function application $e_f\, v_x$ to have a value argument. Modifiables τ **mod** are mutable references with adaptive store commands **putk**, **getk**, and **setk** for allocation, dereference, and update. The type **res** is an opaque answer type, while **halt** is a continuation that injects a final value into the **res** type.

5.1 Static, Dynamic, and Cost Semantics

Figure 7 gives a fragment of the static semantics of Tgt. The typing judgement $\Sigma; \Gamma \vdash e : \tau$ ascribes the type τ to the expression e in the store and variable typing contexts Σ and Γ; the omitted rules are standard.

Figure 8 gives the dynamic semantics. Evaluation uses and produces a *trace* T as a sequence of adaptive (store and memo) actions A, ending in a halt action:

$$\begin{array}{l}
A_{\mathrm{s}} ::= \mathtt{P}^{v \uparrow \ell}_{v_k} \mid \mathtt{G}^{\ell \to v}_{v_k} \mid \mathtt{S}^{\ell \leftarrow v}_{v_k} \\
A ::= A_{\mathrm{s}} \mid \mathtt{M}^{e} \\
T ::= \mathtt{H}^{v} \mid A \cdot T \\
\dot{T} ::= \circ \mid T
\end{array}$$

The large-step evaluation relation $\dot{T}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'$ (resp. $\dot{T}; \sigma; k \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'$) reduces the expression e (resp. the adaptive command k) under the store σ, yielding the value v' and the updated store σ'; evaluation also takes an (optional) reuse trace $\dot{T}$ from a previous run, and produces an execution trace T' for the current run and a pair of costs $d' = \langle c, c' \rangle$ for work c discarded from the reuse trace and new work c' performed for the current run. The auxiliary evaluation relation $e \Downarrow v'$ reduces an expression e to a value v', independent of the store.

The **halt** v command yields a computation's final value, with a cost of 1 for the current run and a cost $c = |\dot{T}|$ for work discarded from the reuse trace $\dot{T}$, where the cost of an optional trace is:

$$|\circ| = 0 \quad |\mathtt{H}^v| = 1 \quad |A \cdot T| = 1 + |T|$$

An adaptive store command uses the store (**putk**, **getk**, and **setk** rules) and delivers the result to the continuation; the trace is extended with the corresponding store action labeled by the location, value, and continuation involved, and incurs a cost of

$$\frac{\Sigma; \Gamma \vdash v : \tau \qquad \Sigma; \Gamma \vdash v_{\mathrm{k}} : \tau\ \mathbf{mod} \to \mathbf{res}}{\Sigma; \Gamma \vdash \mathbf{putk}\, v\, v_{\mathrm{k}} : \mathbf{res}} \qquad \frac{\Sigma; \Gamma \vdash v_{\mathrm{l}} : \tau\ \mathbf{mod} \qquad \Sigma; \Gamma \vdash v_{\mathrm{k}} : \tau \to \mathbf{res}}{\Sigma; \Gamma \vdash \mathbf{getk}\, v_{\mathrm{l}}\, v_{\mathrm{k}} : \mathbf{res}}$$

$$\frac{\Sigma; \Gamma \vdash v_{\mathrm{l}} : \tau\ \mathbf{mod} \qquad \Sigma; \Gamma \vdash v : \tau \qquad \Sigma; \Gamma \vdash v_{\mathrm{k}} : \mathbf{nat} \to \mathbf{res}}{\Sigma; \Gamma \vdash \mathbf{setk}\, v_{\mathrm{l}}\, v\, v_{\mathrm{k}} : \mathbf{res}}$$

$$\frac{\Sigma; \Gamma \vdash e : \mathbf{res}}{\Sigma; \Gamma \vdash \mathbf{memo}\, e : \mathbf{res}} \qquad \frac{\Sigma; \Gamma \vdash v : \tau}{\Sigma; \Gamma \vdash \mathbf{halt}\, v : \mathbf{res}}$$

Figure 7. Tgt typing $\Sigma; \Gamma \vdash e : \tau$ (fragment).

$$\frac{}{v \Downarrow v} \qquad \frac{e_{\mathrm{z}} \Downarrow v}{\mathbf{caseN}\,\mathbf{zero}\,e_{\mathrm{z}}\,(x.e_{\mathrm{s}}) \Downarrow v} \qquad \frac{\{v_{\mathrm{n}} \,/\, x\}\, e_{\mathrm{s}} \Downarrow v}{\mathbf{caseN}\,(\mathbf{succ}\,v_{\mathrm{n}})\,e_{\mathrm{z}}\,(x.e_{\mathrm{s}}) \Downarrow v} \qquad \frac{e_{\mathrm{f}} \Downarrow \mathbf{fun}\,f.x.e \quad \{v_{\mathrm{x}} \,/\, x\}\,\{\mathbf{fun}\,f.x.e \,/\, f\}\, e \Downarrow v}{e_{\mathrm{f}}\, v_{\mathrm{x}} \Downarrow v}$$

$$\frac{e \Downarrow \kappa \quad \dot{T}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'}{\dot{T}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'} \qquad \frac{|\dot{T}| = c}{\dot{T}; \sigma; \mathbf{halt}\, v \Downarrow_{\mathrm{K}} \mathtt{H}^{v}; \sigma; v; \langle c, 1\rangle} \qquad \frac{\ell \notin \mathrm{dom}(\sigma) \quad \sigma_{\mathrm{l}} = \sigma \uplus \{\ell \mapsto v\} \quad \dot{T}; \sigma_{\mathrm{l}}; v_{\mathrm{k}}\, \ell \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'}{\dot{T}; \sigma; \mathbf{putk}\, v\, v_{\mathrm{k}} \Downarrow_{\mathrm{K}} \mathtt{P}_{v_{\mathrm{k}}}^{v\uparrow\ell}\cdot T'; \sigma'; v'; \langle 0, 1\rangle + d'}$$

$$\frac{\ell \in \mathrm{dom}(\sigma) \quad \sigma(\ell) = v \quad \dot{T}; \sigma; v_{\mathrm{k}}\, v \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'}{\dot{T}; \sigma; \mathbf{getk}\, \ell\, v_{\mathrm{k}} \Downarrow_{\mathrm{K}} \mathtt{G}_{v_{\mathrm{k}}}^{\ell\rightarrow v}\cdot T'; \sigma'; v'; \langle 0, 1\rangle + d'} \qquad \frac{\ell \in \mathrm{dom}(\sigma) \quad \sigma_{\mathrm{l}} = \sigma[\ell \mapsto v] \quad \dot{T}; \sigma_{\mathrm{l}}; v_{\mathrm{k}}\, \mathbf{zero} \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'}{\dot{T}; \sigma; \mathbf{setk}\, \ell\, v\, v_{\mathrm{k}} \Downarrow_{\mathrm{K}} \mathtt{S}_{v_{\mathrm{k}}}^{\ell\leftarrow v}\cdot T'; \sigma'; v'; \langle 0, 1\rangle + d'}$$

$$\frac{\dot{T}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'}{\dot{T}; \sigma; \mathbf{memo}\, e \Downarrow_{\mathrm{K}} \mathtt{M}^{e}\cdot T'; \sigma'; v'; \langle 0, 1\rangle + d'}\ \textbf{memo/miss} \qquad \frac{T; e \overset{\mathrm{m}}{\rightsquigarrow} T_{\mathrm{e}}; c \quad T_{\mathrm{e}}; \sigma \curvearrowright T'; \sigma'; v'; d'}{T; \sigma; \mathbf{memo}\, e \Downarrow_{\mathrm{K}} \mathtt{M}^{e}\cdot T'; \sigma'; v'; \langle c, 1\rangle + d'}\ \textbf{memo/hit}$$

Figure 8. Tgt reduction $e \Downarrow v$ and evaluation $\dot{T}; \sigma; k \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'$ and $\dot{T}; \sigma; k \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'$.

1 for the current run. A memoized expression $\mathbf{memo}\, e$ in Tgt has no special behavior when evaluated from scratch (**memo/miss** rule): it evaluates the body e and extends the trace with a memo action $\mathtt{M}^{e}$, incurring a cost of 1 for the current run. The **memo/hit** rule exploits the reuse trace and switches to change propagation. The memoization judgement $T; e \overset{\mathrm{m}}{\rightsquigarrow} T_e; c$ finds a trace T_e that corresponds to a previous run of e (under a (possibly) different store), incurring a cost c for discarding the prefix of T up to T_e:

$$\frac{}{\mathtt{M}^{e}\cdot T; e \overset{\mathrm{m}}{\rightsquigarrow} T; 1} \qquad \frac{T; e \overset{\mathrm{m}}{\rightsquigarrow} T_{\mathrm{e}}; c}{A\cdot T; e \overset{\mathrm{m}}{\rightsquigarrow} T_{\mathrm{e}}; 1 + c}$$

The change propagation relation $T; \sigma \curvearrowright T'; \sigma'; v'; d'$ (given in Figure 9) replays the execution trace T under the store σ, yielding the value v' and the updated store σ', with an updated execution trace T' and a pair of costs $d' = \langle c, c'\rangle$ for work c discarded from T and new work c' performed for T'. A halt action can be replayed without cost to obtain the (unchanged) final value. An adaptive action can be replayed without cost if the action is consistent with the current store (**reuse** rules), the tail of the trace can be recursively change propagated and then extended with the same action. However, if a store action is inconsistent with the store (*e.g.* a specific location can't be allocated), then change propagation must switch back to evaluation. Since adaptive actions capture their continuation, a trace T can be *reified* back into an adaptive command $\lceil T \rceil$ that represents the rest of the computation:

$$\begin{array}{ll} \lceil \mathtt{P}_{v_k}^{v\uparrow\ell}\cdot T \rceil = \mathbf{putk}\, v\, v_k & \lceil \mathtt{M}^{e}\cdot T \rceil = \mathbf{memo}\, e \\ \lceil \mathtt{G}_{v_k}^{\ell\rightarrow v}\cdot T \rceil = \mathbf{getk}\, \ell\, v_k & \lceil \mathtt{H}^{v} \rceil = \mathbf{halt}\, v \\ \lceil \mathtt{S}_{v_k}^{\ell\leftarrow v}\cdot T \rceil = \mathbf{setk}\, \ell\, v\, v_k & \end{array}$$

Thus, change propagation can reify and re-evaluate an inconsistent trace T (**change** rule), while keeping the trace T for possible reuse later. Note that the reified $\mathbf{putk}$ (resp. $\mathbf{getk}$) forgets the (stale) location (resp. value). The **change** rule does *not*, however, require the action to be inconsistent; this nondeterminism intentionally avoids committing to particular allocation and memoization policies.

5.2 Consistency of Change Propagation

Suppose we have a Tgt program e such that $\Sigma; \cdot \vdash e : \mathbf{res}$ and an initial store σ_1 such that $\vdash \sigma_1 : \Sigma \uplus \Sigma_1$. We can evaluate e under the store σ_1 and no reuse trace, yielding the initial result v_1' and a trace T_1': $\circ; \sigma_1; e \Downarrow_{\mathrm{E}} T_1'; \sigma_1'; v_1'; d_1'$. After this initial evaluation, we can consider another store σ_2 such that $\vdash \sigma_2 : \Sigma \uplus \Sigma_2$ and update the output of the evaluation with respect to this store by applying change propagation to T_1' under the store σ_2: $T_1'; \sigma_2 \curvearrowright T_2'; \sigma_2'; v_2'; d_2'$. The consistency of change propagation asserts that the result and trace obtained by change propagation are identical to those obtained by evaluation (recall the bottom left square of Figure 1). We prove this consistency property for Tgt by giving a simple structural proof.

Theorem 4 (Consistency of Change propagation)
If $\circ; \sigma_1; e \Downarrow_{\mathrm{E}} T_1'; \sigma_1'; v_1'; \_$ *and* $T_1'; \sigma_2 \curvearrowright T_2'; \sigma_2'; v_2'; \_$,
then $\circ; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$.
If $\circ; \_; \_ \Downarrow_{\mathrm{E}} T_1'; \_; \_; \_$ *and* $T_1'; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$,
then $\circ; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$.

Proof: The theorem must be strengthened with analogous statements for the $\Downarrow_{\mathrm{K}}$ relation. By simultaneous induction on the second derivation of each statement. Proved in Twelf. ■

Recent work gave a similar consistency theorem, but with a different language (Acar et al. 2008a). Compared to that work, our proof is dramatically simpler. We achieve this by casting change propagation as a full replay mechanism that can re-allocate locations. In previous work, it was not possible to express change propagation as a full replay mechanism—change propagation could not re-allocate locations allocated in a previous run. This required arguing that the results obtained by change propagation and evaluation are *isomorphic* by using step-indexed logical relations.

5.3 Trace Distance

Reasoning about computation reuse achieved by change propagation is difficult. In this section, we introduce a notion of trace distance and show that the cost of change propagation may be bounded by the distance between the input and the result traces. The definition of distance is similar to the source at a high level. Indeed, in Section 6 we show that they are asymptotically the same.

As in Src, we define a *search distance* $T_1 \boxminus T_2 = d$ that accounts for differences between traces until it finds matching memoization actions, at which point it can use the *synchronization distance* $T_1 \ominus T_2 = d$ that accounts for reuse between traces until they differ, at which point it must return to the search distance. The distance $d = \langle c_1, c_2\rangle$ quantifies the cost c_1 of work in T_1 that isn't shared with T_2 and the cost c_2 of work in T_2 that isn't shared with T_1.

The search distance (given in Figure 10) between halt actions is 1 for each action, irrespective of the value returned. Two identical memo actions incur a cost of 1 each, but afford the possibility of switching from search to synchronization mode. Skipping an action incurs a cost of 1 for the corresponding trace and forces distance to remain in search mode. Note that these last two rules allow memo actions to remain in search mode; identical memo actions are never forced to synchronize.

Synchronization distance, as in Src, is only meant to be used on traces generated by the evaluation of the same expression under

$$\frac{\ell \notin \mathrm{dom}(\sigma) \quad \sigma_1 = \sigma \uplus \{\ell \mapsto v\} \qquad T; \sigma_1 \curvearrowright T'; \sigma'; v'; d'}{\mathtt{P}^{v\uparrow\ell}_{v_k}{\cdot}T; \sigma \curvearrowright \mathtt{P}^{v\uparrow\ell}_{v_k}{\cdot}T'; \sigma'; v'; d'}\ \textbf{put/reuse} \qquad \frac{\ell \in \mathrm{dom}(\sigma) \quad \sigma(\ell) = v \qquad T; \sigma \curvearrowright T'; \sigma'; v'; d'}{\mathtt{G}^{\ell\to v}_{v_k}{\cdot}T; \sigma \curvearrowright \mathtt{G}^{\ell\to v}_{v_k}{\cdot}T'; \sigma'; v'; d'}\ \textbf{get/reuse} \qquad \frac{\ell \in \mathrm{dom}(\sigma) \quad \sigma_1 = \sigma[\ell \mapsto v] \qquad T; \sigma_1 \curvearrowright T'; \sigma'; v'; d'}{\mathtt{S}^{\ell\leftarrow v}_{v_k}{\cdot}T; \sigma \curvearrowright \mathtt{S}^{\ell\leftarrow v}_{v_k}{\cdot}T'; \sigma'; v'; d'}\ \textbf{set/reuse}$$

$$\frac{T; \sigma \curvearrowright T'; \sigma'; v'; d'}{\mathtt{M}^e{\cdot}T; \sigma \curvearrowright \mathtt{M}^e{\cdot}T'; \sigma'; v'; d'}\ \textbf{memo/reuse} \qquad \frac{}{\mathtt{H}^v; \sigma \curvearrowright \mathtt{H}^v; \sigma; v; \langle 0,0\rangle} \qquad \frac{\lceil T \rceil = \kappa \qquad T; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'}{T; \sigma \curvearrowright T'; \sigma'; v'; d'}\ \textbf{change}$$

Figure 9. Tgt change propagation $\dot{T}; \sigma \curvearrowright \sigma'; v'; T; d'$.

$$\frac{}{\mathtt{H}^{v_1} \boxminus \mathtt{H}^{v_2} = \langle 1,1\rangle} \qquad \frac{T_1 \ominus T_2 = d}{\mathtt{M}^e{\cdot}T_1 \boxminus \mathtt{M}^e{\cdot}T_2 = \langle 1,1\rangle + d} \qquad \frac{T_1 \boxminus T_2 = d}{A{\cdot}T_1 \boxminus T_2 = \langle 1,0\rangle + d} \qquad \frac{T_1 \boxminus T_2 = d}{T_1 \boxminus A{\cdot}T_2 = \langle 0,1\rangle + d}$$

$$\frac{}{\mathtt{H}^v \ominus \mathtt{H}^v = \langle 0,0\rangle} \qquad \frac{T_1 \ominus T_2 = d}{A{\cdot}T_1 \ominus A{\cdot}T_2 = d} \qquad \frac{T_1 \boxminus T_2 = d}{T_1 \ominus T_2 = d}$$

Figure 10. Tgt trace search distance $T_1 \boxminus T_2 = d$ and synchronization distance $T_1 \ominus T_2 = d$.

(possibly) different stores (though, there exists a synchronization distance between any two traces). The synchronization distance between halt actions is $\langle 0,0\rangle$, and assumes both actions return the same value. Identical adaptive actions match without cost and allow distance to continue synchronizing the tail. Synchronization may return to search mode, either nondeterministically or because adaptive actions don't match. Just as Src distance, Tgt distance judgements are quasi-symmetric and induce a ternary relation due to the nondeterminism of memo matching.

In light of the dynamic semantics, trace distance can be given an asymmetrical operational interpretation: the distance is the amount of work that must be discarded from one run and executed to produce the other run. (Intuitively, the asymmetry arises from the fact that discarding work, while not free, is cheaper than performing work.) In particular, search distance has an operational analogue realized by evaluation, while synchronization distance is realized by change propagation. A distance $\langle c_1, c_2\rangle$ between traces T_1 and T_2 intuitively means there is cost c_1 for discarding unusable work from the reuse trace T_1 and cost c_2 for performing new work for T_2, but any common work that can be reused is free. This relation between distance and the dynamic semantics is formally captured by the following theorem (recall the bottom right diagram of Figure 1).

Theorem 5 (Dynamic semantics coincides with distance)
If $\circ; \sigma_1; e_1 \Downarrow_{\mathrm{E}} T'_1; \sigma'_1; v'_1; \_$ *and* $\circ; \sigma_2; e_2 \Downarrow_{\mathrm{E}} T'_2; \sigma'_2; v'_2; \_$,
then $T'_1 \boxminus T'_2 = d$ *iff* $T'_1; \sigma_2; e_2 \Downarrow_{\mathrm{E}} T'_2; \sigma'_2; v'_2; d$.
If $\circ; \sigma_1; e \Downarrow_{\mathrm{E}} T'_1; \sigma'_1; v'_1; \_$ *and* $\circ; \sigma_2; e \Downarrow_{\mathrm{E}} T'_2; \sigma'_2; v'_2; \_$,
then $T'_1 \ominus T'_2 = d$ *iff* $T'_1; \sigma_2 \curvearrowright T'_2; \sigma'_2; v'_2; d$.

Proof: The theorem must be strengthened with analogous statements for the $\Downarrow_{\mathrm{K}}$ judgement. By simultaneous induction on the second derivation of each statement. Proved in Twelf. ■

6. Translation

Program Translation. The adaptive primitives of Src programs are used to guide an *adaptive continuation-passing style* (ACPS) transformation into equivalent Tgt programs (given in Figure 11). The type translation $[\![\tau^{\mathsf{s}}]\!] = \tau^{\mathsf{t}}$ preserves the $\mathbf{nat}$ type, converts the function type to take a continuation argument, and converts the reference type to a modifiable type. The expression and value translations $[\![e^{\mathsf{s}}]\!]\ v^{\mathsf{t}}_k = e^{\mathsf{t}}$ and $[\![v^{\mathsf{s}}]\!] = v^{\mathsf{t}}$ (the former using the Tgt value v^{t}_k as an explicit continuation) are standard cps conversions for natural numbers, while reference primitives are translated into Tgt adaptive store commands with an explicit continuation v_k. The value translations (except for functions) are straightforward. The $\mathbf{halt}$ expression is not in the image of the translation, but it can be used as an initial identity continuation $\mathbf{id} = \lambda\, x.\mathbf{halt}\, x$ for evaluating a cps-converted program. The metavariable y is used to distinguish identifiers introduced by the translation. The type translation is extended pointwise to Src store and variable typing contexts Σ and Γ; the value translation is extended pointwise to Src stores σ.

The cps discipline in Tgt facilitates identifying the scope of an adaptive store action as the rest of the computation, so change propagating an inconsistent store action will re-execute the tail of the trace. Memoizing a function, however, in the presence of (possibly different) continuations and store mutation is subtle and crucially relies on two ideas: threading continuations through the store, and using explicit $\mathbf{memo}$ operations before and after the function body. First, the translation treats the continuation as changeable data by threading it through the store during the function call (*cf.* $\mathbf{putk}$ in the function body and $\mathbf{getk}$ in the continuation). This effectively shifts the continuation to the store, so the function call can memo match on its argument even if its continuation differs (provided the same location is used to store the continuation as in the previous run). After the function body is change propagated without cost, the (new) continuation will be resumed by reading it from the store and passing it the memoized result. Second, the translation inserts memo commands at the function call *and* return points in an attempt to isolate reuse of the function body separately from reuse of the rest of the computation. Thus the continuation can memo match if the result is the same, even if the function body had to re-execute due to an inconsistent store interaction.

The correctness and efficiency of the translation is captured by the fact that well-typed Src programs are compiled into (statically and dynamically) equivalent well-typed Tgt programs with the same asymptotic complexity for initial runs (*i.e.*, Tgt evaluation with an empty reuse trace). Type preservation is standard and elided for reasons of space. More importantly, the evaluation and asymptotic cost of from-scratch runs of Src programs is preserved by compilation (recall the top right diagram of Figure 1).

Theorem 6 (Translation preserves extensional/intensional)
If $\mathcal{E}; \sigma_0; e_0 \Downarrow \sigma_1; v_1; T; c_0$,
and $\circ; [\![\sigma_1]\!] \uplus \sigma_k; v_k\ [\![v_1]\!] \Downarrow_{\mathrm{E}} \sigma_2; v_2; T_k; \langle \_, c_1\rangle$,
then $\circ; [\![\sigma_0]\!] \uplus \sigma_k; [\![e_0]\!]\ v_k \Downarrow_{\mathrm{E}} \sigma_2 \uplus \sigma_e; v_2; T'; \langle \_, c_2\rangle$
and $c_0 + c_1 \le c_2 \le 4c_0 + c_1$ *whence* $c_2 \in \Theta(c_0 + c_1)$.

Proof: By induction on the first derivation. ■

The store σ_k accounts for locations free in the continuation v_k, while the store σ_e accounts for locations allocated for (the continuations of) memoizing functions. Instantiating this theorem

$$
\begin{aligned}
\llbracket \mathbf{nat} \rrbracket &= \mathbf{nat} \\
\llbracket \tau_{\mathrm{x}} \to \tau \rrbracket &= \llbracket \tau_{\mathrm{x}} \rrbracket \to (\llbracket \tau \rrbracket \to \mathbf{res}) \to \mathbf{res} \\
\llbracket \tau\ \mathbf{ref} \rrbracket &= \llbracket \tau \rrbracket\ \mathbf{mod}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket v \rrbracket\ v_{\mathrm{k}} &= v_{\mathrm{k}}\ \llbracket v \rrbracket \\
\llbracket \mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, (x.e_{\mathrm{s}}) \rrbracket\ v_{\mathrm{k}} &= \mathbf{caseN}\ \llbracket v_{\mathrm{n}} \rrbracket\ (\llbracket e_{\mathrm{z}} \rrbracket\ v_{\mathrm{k}})\, (x.\, \llbracket e_{\mathrm{s}} \rrbracket\ v_{\mathrm{k}}) \\
\llbracket e_{\mathrm{f}}\, e_{\mathrm{x}} \rrbracket\ v_{\mathrm{k}} &= \llbracket e_{\mathrm{f}} \rrbracket\ (\lambda\, y_{\mathrm{f}}.\, \llbracket e_{\mathrm{x}} \rrbracket\ (\lambda\, y_{\mathrm{x}}.(y_{\mathrm{f}}\ y_{\mathrm{x}})\, v_{\mathrm{k}})) \\
\llbracket \mathbf{put}\, v \rrbracket\ v_{\mathrm{k}} &= \mathbf{putk}\, \llbracket v \rrbracket\, v_{\mathrm{k}} \\
\llbracket \mathbf{get}\, v_{\mathrm{l}} \rrbracket\ v_{\mathrm{k}} &= \mathbf{getk}\, \llbracket v_{\mathrm{l}} \rrbracket\, v_{\mathrm{k}} \\
\llbracket \mathbf{set}\, v_{\mathrm{l}}\ v \rrbracket\ v_{\mathrm{k}} &= \mathbf{setk}\, \llbracket v_{\mathrm{l}} \rrbracket\, \llbracket v \rrbracket\, v_{\mathrm{k}} \\
\llbracket x \rrbracket &= x \\
\llbracket \mathbf{zero} \rrbracket &= \mathbf{zero} \\
\llbracket \mathbf{succ}\, v \rrbracket &= \mathbf{succ}\ \llbracket v \rrbracket \\
\llbracket \ell \rrbracket &= \ell \\
\llbracket \mathbf{fun}\, f.x.e \rrbracket &= \\
\mathbf{fun}\, f.x.\lambda\, y_{\mathrm{k}}. & \\
\quad \mathbf{putk}\, (\lambda\, y_{\mathrm{r}}.\mathbf{memo}\, (y_{\mathrm{k}}\ y_{\mathrm{r}})) & \\
\quad\quad (\lambda\, y_{\mathrm{l}}.\mathbf{memo}\, (\llbracket e \rrbracket\ (\lambda\, y_{\mathrm{r}}.\mathbf{getk}\, y_{\mathrm{l}}\, (\lambda\, y_{\mathrm{k}}.y_{\mathrm{k}}\ y_{\mathrm{r}})))) &
\end{aligned}
$$

Figure 11. Type translation $\llbracket \tau^{\mathsf{s}} \rrbracket = \tau^{\mathsf{t}}$ (top) and term translations $\llbracket e^{\mathsf{s}} \rrbracket\ v_k^{\mathsf{t}} = e^{\mathsf{t}}$ and $\llbracket v^{\mathsf{s}} \rrbracket = v^{\mathsf{t}}$ (bottom).

with the identity continuation $v_k = \mathbf{id}$, we have that evaluation of a Src program coincides with (from-scratch) Tgt evaluation of its ACPS-translation. Furthermore, the adaptive work $c_2 \in \Theta(c_0)$ in Tgt is proportional to the active work c_0 in Src, because the work of the identity continuation is constant. This means that the translation preserves the asymptotic complexity of from-scratch runs.

Trace Translation. The Tgt trace of an ACPS-compiled Src program is richer than its Src counterpart because Tgt traces have explicit continuations and the ACPS translation introduces administrative redices, threads continuations through the store, and inserts memoization at function call and return points. The Src dynamic semantics and distance, however, are sufficiently instrumented to translate Src traces into equivalent Tgt traces. An explicit Src evaluation context $\mathcal{E}$ is sufficient to reify the current continuation $\llbracket \mathcal{E} \rrbracket\ v_k$ relative to an initial Tgt continuation v_k:

$$
\begin{aligned}
\llbracket \square \rrbracket\ v_{\mathrm{k}} &= v_{\mathrm{k}} \\
\llbracket \mathcal{E}\, e_{\mathrm{x}} \rrbracket\ v_{\mathrm{k}} &= \llbracket \mathcal{E} \rrbracket\ (\lambda\, y_{\mathrm{f}}.\, \llbracket e_{\mathrm{x}} \rrbracket\ (\lambda\, y_{\mathrm{x}}.(y_{\mathrm{f}}\ y_{\mathrm{x}})\, v_{\mathrm{k}})) \\
\llbracket v_{\mathrm{f}}\, \mathcal{E} \rrbracket\ v_{\mathrm{k}} &= \llbracket \mathcal{E} \rrbracket\ (\lambda\, y_{\mathrm{x}}.(\llbracket v_{\mathrm{f}} \rrbracket\ y_{\mathrm{x}})\, v_{\mathrm{k}})
\end{aligned}
$$

Moreover, since active Src actions are instrumented with their local evaluation context, we can give a *trace translation* $\llbracket T^{\mathsf{s}} \rrbracket\ v_k^{\mathsf{t}}\ T_k^{\mathsf{t}}$ of Src trace T^{s} using the v_k^{t} as an initial continuation (to extend the local context $\mathcal{E}$ of actions) and suffix T_k^{t}. The translation of the empty trace and store actions is straightforward:

$$
\begin{aligned}
\llbracket \varepsilon \rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}} &= T_{\mathrm{k}} \\
\left\llbracket \mathtt{P}_{\mathcal{E}}^{v\uparrow\ell}\cdot T \right\rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}} &= \mathtt{P}_{\llbracket \mathcal{E} \rrbracket\, v_{\mathrm{k}}}^{\llbracket v \rrbracket\uparrow\ell}\cdot(\llbracket T \rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}}) \\
\llbracket \mathtt{G}_{\mathcal{E}}^{\ell\to v}\cdot T \rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}} &= \mathtt{G}_{\llbracket \mathcal{E} \rrbracket\, v_{\mathrm{k}}}^{\ell\to\llbracket v \rrbracket}\cdot(\llbracket T \rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}}) \\
\llbracket \mathtt{S}_{\mathcal{E}}^{\ell\leftarrow v}\cdot T \rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}} &= \mathtt{S}_{\llbracket \mathcal{E} \rrbracket\, v_{\mathrm{k}}}^{\ell\leftarrow\llbracket v \rrbracket}\cdot(\llbracket T \rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}})
\end{aligned}
$$

Since a failure action is inserted at a function's return point, it is translated to the trace that follows the evaluation of a function body (*cf.* ACPS function translation):

$$
\begin{aligned}
\left\llbracket \mathtt{F}_{\mathcal{E}}^{\Downarrow v}\cdot T' \right\rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}} &= \mathtt{G}_{k_{\mathrm{a}}}^{\ell\to k_{\mathrm{w}}}\cdot\mathtt{M}^{((\llbracket \mathcal{E} \rrbracket\, v_{\mathrm{k}})\, \llbracket v \rrbracket)}\cdot(\llbracket T' \rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}}) \\
\text{where}\ \ k_{\mathrm{w}} &= \lambda\, y_{\mathrm{r}}.\mathbf{memo}\, ((\llbracket \mathcal{E} \rrbracket\ v_{\mathrm{k}})\, y_{\mathrm{r}}) \\
k_{\mathrm{a}} &= \lambda\, y_{\mathrm{k}}.y_{\mathrm{k}}\ \llbracket v \rrbracket
\end{aligned}
$$

Note that k_{w} is the memoizing version of the original continuation that was written to the store before the evaluation of the body and k_{a} is the continuation of the $\mathbf{getk}$ command that fetches the memoizing version of original continuation.

The translation of a memoizing function action must account for writing the memoizing version of the original continuation to the store before memoizing on the evaluation of the body:

$$
\begin{aligned}
\left\llbracket \mathtt{M}_{\mathcal{E}}^{(\mathbf{fun}\, f.x.e)\ v_{\mathrm{x}}\Downarrow v}(T)\cdot T' \right\rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}} &= \mathtt{P}_{k_{\mathrm{m}}}^{k_{\mathrm{w}}\uparrow\ell}\cdot\mathtt{M}^{(\llbracket e' \rrbracket\ k_{\mathrm{r}})}\cdot(\llbracket T \rrbracket\ k_{\mathrm{r}}\, T_{\mathrm{r}}) \\
\text{where}\ \ k_{\mathrm{w}} &= \lambda\, y_{\mathrm{r}}.\mathbf{memo}\, ((\llbracket \mathcal{E} \rrbracket\ v_{\mathrm{k}})\, y_{\mathrm{r}}) \\
k_{\mathrm{m}} &= \lambda\, y_{\mathrm{l}}.\mathbf{memo}\, (\llbracket e' \rrbracket\ (\lambda\, y_{\mathrm{r}}.\mathbf{getk}\, y_{\mathrm{l}}\, (\lambda\, y_{\mathrm{k}}.y_{\mathrm{k}}\ y_{\mathrm{r}}))) \\
e' &= \{\mathbf{fun}\, f.x.e\, /\, f\}\, \{v_{\mathrm{x}}\, /\, x\}\, e \\
k_{\mathrm{r}} &= \lambda\, y_{\mathrm{r}}.\mathbf{getk}\, \ell\, (\lambda\, y_{\mathrm{k}}.y_{\mathrm{k}}\ y_{\mathrm{r}}) \\
T_{\mathrm{r}} &= \left\llbracket \mathtt{F}_{\mathcal{E}}^{\Downarrow v}\cdot T' \right\rrbracket\ v_{\mathrm{k}}\, T_{\mathrm{k}}
\end{aligned}
$$

Note that k_{r} is the continuation that fetches and invokes the memoizing version of the original continuation; k_{r} is the continuation that is passed to the body. The body of the memoizing function action is translated with respect to k_{r} and T_{r}, which is the translation of a failure action. Trace translation is syntax-directed, except for the choice of locations for continuations of memoizing functions; below we specify how these locations are chosen.

Given the trace translation, Theorem 6 can be strengthened to show that the if the continuation v_k is of the form $\llbracket \mathcal{E} \rrbracket\ v_k'$, then the Tgt evaluation trace T' is $\llbracket T \rrbracket\ v_k\ T_k$. Finally, Src distance may be related to Tgt distance by trace translation (recall top right diagram of Figure 1).

Theorem 7 (Src/Tgt **distance soundness)**
Assume $T_{k1}^{\mathsf{t}} \ominus T_{k2}^{\mathsf{t}} = \langle \_, c_1' \rangle$, $T_{k1}^{\mathsf{t}} \boxminus T_{k2}^{\mathsf{t}} = \langle \_, c_2' \rangle$.
If $T_1 \boxminus T_2 = \langle \_, c \rangle$,
then $(\llbracket T_1 \rrbracket\ v_k^{\mathsf{t}}\, T_{k1}^{\mathsf{t}}) \boxminus (\llbracket T_2 \rrbracket\ v_k^{\mathsf{t}}\, T_{k2}^{\mathsf{t}}) = \langle \_, c'' \rangle$
and $c \leq c'' \leq 4c + \max\{c_1', c_2'\}$.
If $T_1 \ominus T_2 = \langle \_, c \rangle$,
then $(\llbracket T_1 \rrbracket\ v_{k1}^{\mathsf{t}}\, T_{k1}^{\mathsf{t}}) \ominus (\llbracket T_2 \rrbracket\ v_{k2}^{\mathsf{t}}\, T_{k2}^{\mathsf{t}}) = \langle \_, c'' \rangle$
and $c \leq c'' \leq 4c + \max\{c_1', c_2'\}$.

Proof (sketch): We define a variant of Src's distance relation with precise accounting for memoization at function call and return points. We show that the original Src distance bounds the precise Src distance by a constant factor (the original version uses amortization to avoid precise accounting and to simplify reasoning). Next, we preprocess the precise Src distance derivation by assigning matching fresh locations to memoization actions that synchronize, these are used by the trace translation for continuations (this is always possible because stores and traces are finite). Finally, we proceed by induction on the (instrumented) precise Src distance derivation, using the trace translation to build an equivalent Tgt distance derivation. ∎

Corollary 8 (Src/Tgt **distance soundness)**
Let $T_{\mathbf{id}i}^{\mathsf{t}}$ *be the identity continuation trace for* T_i $(i \in \{1, 2\})$.
If $T_1 \boxminus T_2 = \langle \_, c \rangle$,
then $(\llbracket T_1 \rrbracket\ \mathbf{id}\, T_{\mathbf{id}1}^{\mathsf{t}}) \boxminus (\llbracket T_2 \rrbracket\ \mathbf{id}\, T_{\mathbf{id}2}^{\mathsf{t}}) = \langle \_, c'' \rangle$ *and* $c'' \in \Theta(c)$.
If $T_1 \ominus T_2 = \langle \_, c \rangle$,
then $(\llbracket T_1 \rrbracket\ \mathbf{id}\, T_{\mathbf{id}1}^{\mathsf{t}}) \ominus (\llbracket T_2 \rrbracket\ \mathbf{id}\, T_{\mathbf{id}2}^{\mathsf{t}}) = \langle \_, c'' \rangle$ *and* $c'' \in \Theta(c)$.

Proof: The search distance $T_{\mathbf{id}1}^{\mathsf{t}} \boxminus T_{\mathbf{id}2}^{\mathsf{t}}$ and synchronization distance $T_{\mathbf{id}1}^{\mathsf{t}} \ominus T_{\mathbf{id}2}^{\mathsf{t}}$ between the identity continuation traces are constant, therefore the asymptotic bound $c'' \in \Theta(c)$ follows by Theorem 7. ∎

Note that since Src and Tgt distance are quasi-symmetric, an analogous theorem and corollary hold of the left component of distance. This means that change propagation has the same asymptotic time-complexity as trace distance. The converse of the theorem does not hold: a distance may be derivable of Tgt traces which does not correspond to any derivable Src distance. This incompleteness is to be expected because memoization of a function call and return in Tgt need not match in lock-step, whereas the **synch/memo** (resp. **synch/search**) Src rule requires both (resp. neither) to match in lock-step.

7. Discussion

We briefly remark on some limitations of our approach.

Incompleteness. Soundness of the translation guarantees that any distance derivable in Src is also (up to a constant factor) derivable in Tgt. However, the Tgt proof system exhibits more possible distances: in Src, memoization requires matching both the function call and return points, while the ACPS translation into Tgt distinguishes memoization at the call and the return. Therefore, there are more opportunities for switching between search and synchronization in Tgt and there may be more distance values derivable in Tgt than in Src. For example, in Tgt a function call memoization can miss (*i.e.*, remain in search mode) while the return can match (*i.e.*, switch from search to synchronization mode), which is not possible in Src. Consequently, any upper bound found using Src distance is preserved by compilation, but lower bound arguments on a Src program are not necessarily lower bounds on the Tgt distance.

Nondeterminism. The dynamic semantics and distance of Src and Tgt programs are nondeterministic due to the freedom in choosing locations as well as deciding when memoization matches. This avoids having to commit to a particular implementation, but also means that any upper bound derived using the nondeterministic semantics may not be realized in a particular implementation. In order for an implementation to realize an upper bound on distance, allocation and memoization policies used in deriving the distance must coincide those of the implementation. In previous work (Ley-Wild et al. 2008b), we proposed both user-specified and compiler-generated mechanisms for defining allocation and memoization policies, which suffices for realizing the bounds derived in our examples. Ultimately, it would be useful to develop compilation and run-time techniques to minimize automatically the distance between the computations by considering all possible policies.

Meta-logic. The proof system for distance applies to concrete traces, while in our examples we use it to reason schematically over classes of contexts and input changes. To fully formalize the examples, we would need a meta-logic that permits quantification over contexts and classes of input changes, and can express asymptotic bounds. Such a meta-logic could be extended with theorem-proving capabilities which could automate finding bounds on distance.

8. Related Work

We briefly review previous work on incremental computation and cost semantics.

Incremental and Self-Adjusting Computation Incremental computation has been studied extensively since the early 80's. We briefly mention a few approaches here and refer the reader to the survey by Ramalingam and Reps (1993) and some recent papers (*e.g.*, (Ley-Wild et al. 2008b)) for a more detailed set of references. Effective early approaches to incremental computation either use dependence graphs (Demers et al. 1981; Reps 1982; Yellin and Strom 1991) or memoization (*e.g.*, Pugh and Teitelbaum 1989; Abadi et al. 1996; Heydon et al. 2000). Self-adjusting computation generalized dependence graphs techniques by introducing dynamic dependence graphs (Acar et al. 2006b), which enables a change propagation algorithm update the structure of the computation based on data modifications, and combining them with memoization (Acar et al. 2006a). Recent work showed that the approach can be generalized to support imperative updates (side effects to memory) (Acar et al. 2008a). Ley-Wild et al 2008b described how to incorporate a version of the compilation technique used in this paper for a pure source language into an existing compiler (MLton). That paper did not consider mutable references and provided no cost semantics or effectiveness guarantees.

Researchers proposed several implementations of self-adjusting computation. Carlsson (2002) present a Haskell implementation of the initial proposal to self-adjusting computation (Acar et al. 2006b). Shankar and Bodik 2007 use a variant of self-adjusting computation techniques for the purpose of incremental invariant checking. Cooper and Krishnamurthi (Cooper and Krishnamurthi 2006) adapt the initial proposal for self-adjusting computation (Acar et al. 2006b) to support Functional Reactive Programming (Elliott and Hudak 1997)). All of these implementations of self-adjusting computation assume purely functional programming (except for the mutator) and often require support from a higher-order language, e.g. ML, Haskell. Recent work made some progress on giving an implementation of self-adjusting computation in the C language (Hammer and Acar 2008).

Self-adjusting computation has been applied, in several incarnations, to a number of problems from a reasonably broad set of application domains such as motion simulation (Acar et al. 2006c, 2008b), machine learning (Acar et al. 2008c), and other algorithmic problems (Acar et al. 2004, 2005, 2006a). It is possible to analyze the performance of change propagation for a particular problem by using algorithmic analysis techniques. For example, earlier work (Acar et al. 2004) analyzed the performance of change propagation for tree contraction problem. Most applications of self-adjusting computation, however, evaluated the effectiveness of the approach experimentally e.g., (Acar et al. 2006a). The examples that we consider in this paper confirm these experimental findings.

Cost Semantics This work builds on previous work on profiling or cost semantics for reasoning about resource requirements of programs. The idea of instrumenting evaluations to generate cost information goes back to the early 90s (Sands 1990a; Rosendahl 1989). The approach has been shown to be particularly important in high-level languages such as lazy (*e.g.*, Sands 1990a,b; Sansom and Jones 1995) and parallel languages (*e.g.*, Blelloch and Greiner 1995, 1996; Spoonhower et al. 2008) where it is particularly difficult to relate execution time to the source code. The idea of having a cost semantics construct a trace resembles the techniques used for evaluation of parallel programs (Blelloch and Greiner 1996; Spoonhower et al. 2008). The structure and use of our traces, however, differs significantly from those used in parallel languages: we record store actions and compute distances, whereas they work in a pure setting and use traces to reason about parallelism. In the context of incremental computation, we know of no other work that offers a source-level cost semantics for reasoning about effectiveness of incremental update mechanisms.

9. Conclusion

Due to its complex semantics and the nature of previously proposed linguistic facilities, reasoning about the effectiveness of self-adjusting programs has been difficult, forcing previous work to resort to experimental validation.

This paper gives a high-level cost semantics for self-adjusting computation. The approach enables programming in a familiar setting, λ-calculus with first-class references, and compiling such programs into self-adjusting programs. The user can determine the responsiveness of compiled self-adjusting programs by computing a kind of "edit distance" between traces of source programs. These traces consists of function calls and individual store operations. The user need not reason about evaluation contexts or global state. These results are made possible by 1) a compilation mechanism that can translate ordinary code into self-adjusting code while preserving its efficiency, and 2) by techniques for matching evaluation contexts appropriately without exposing them to the user for source-level reasoning.

A common limitation of cost semantics-based approaches to performance analysis is that they often apply only to concrete evaluations. We show that this need not be the case by providing techniques for generalizing trace distances of concrete evaluations to arbitrary inputs, composing trace distances, and by reasoning with trace contexts. For illustrative purposes, we derive asymptotic bounds for several examples. We expect these results to lead to a more formal and precise reasoning of effectiveness of self-adjusting programs as well as profiling tools that can infer concrete and perhaps asymptotic complexity bounds.

References

Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and Caching of Dependencies. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 83–91, 1996.

Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2005.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006a.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2006b.

Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic Algorithms via Self-Adjusting Computation. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pages 636–647, September 2006c.

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2008a.

Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*, September 2008b.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Inference on General Graphical Models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008c.

Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 226–237, 1995. ISBN 0-89791-719-7.

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 213–225. ACM, 1996.

Magnus Carlsson. Monads for Incremental Computing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming (ICFP)*, pages 26–35. ACM Press, 2002.

Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*, 2006.

Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental Evaluation of Attribute Grammars with Application to Syntax-directed Editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

Conal Elliott and Paul Hudak. Functional Reactive Animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273. ACM, 1997.

David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

Leonidas J. Guibas. Kinetic data structures: a state of the art report. In *WAFR '98: Proceedings of the third workshop on the algorithmic foundations of robotics*, pages 191–209, 1998.

Matthew Hammer and Umut A. Acar. Memory Management for Self-Adjusting Computation. In *The 2008 International Symposium on Memory Management*, 2008.

Allan Heydon, Roy Levin, and Yuan Yu. Caching Function Calls Using Precise Dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 311–320, 2000.

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A Cost Semantics for Self-Adjusting Computation. Technical Report CMU-CS-08-141, Department of Computer Science, Carnegie Mellon University, July 2008a.

Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2008b.

William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.

G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 502–510, 1993.

Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages (POPL)*, pages 169–176, 1982.

Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156. ACM, 1989.

David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, September 1990a.

David Sands. Complexity analysis for a lazy higher-order language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*, pages 361–376. Springer-Verlag, 1990b.

Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–366, 1995.

Ajeet Shankar and Rastislav Bodik. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming language Design and Implementation (PLDI)*, 2007.

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2008.

Philip Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proc. of Functional programming languages and computer architecture*, pages 385–407. Springer-Verlag, 1987.

D. M. Yellin and R. E. Strom. INC: A Language for Incremental Computations. *ACM Transactions on Programming Languages and Systems*, 13 (2):211–236, April 1991.