Non-monotonic Self-Adjusting Computation

Ruy Ley-Wild[1], Umut A. Acar[2], and Guy Blelloch[3]

[1] IMDEA Software Institute
[2] Max-Planck Institute for Software Systems
[3] Carnegie Mellon University

Abstract. Self-adjusting computation is a language-based approach to writing programs that respond dynamically to input changes by maintaining a *trace* of the computation consistent with the input, thus also updating the output. For *monotonic* programs, i.e. where localized input changes cause localized changes in the computation, the trace can be repaired efficiently by insertions and deletions. However, non-local input changes can cause major reordering of the trace. In such cases, updating the trace can be asymptotically equal to running from scratch.

In this paper, we eliminate the monotonicity restriction by generalizing the update mechanism to use *trace slices*, which are partial fragments of the computation that can be reordered with some bookkeeping. We provide a high-level source language for pure programs, equipped with a notion of *trace distance* for comparing two runs of a program modulo reordering. The source language is translated into a low-level target language with intrinsic support for *non-monotonic* update (i.e., with reordering). We show that the translation asymptotically preserves the semantics and trace distance, that the cost of update coincides with trace distance, and that updating produces the same answer as a from-scratch run. We describe a concrete algorithm for implementing change-propagation with asymptotic bounds on running time. The concrete algorithm achieves running time bounds which are within $O(\log n)$ of the trace distance, where n is the trace length.

1 Introduction

In many applications, small changes to the input data cause proportionally small changes to the computation and output data. The broad goal of *incremental computation* is to exploit this correlation by efficiently updating the output when the input changes. *Dynamic* algorithms and data structures can be designed to take advantage of the particular problem structure [7,9].The manual approach often yields updates that are asymptotically faster than recomputing from scratch, but carries inherent complexity and non-compositionality that makes the algorithms difficult to design, analyze, and use.

Programming languages for incremental computation provide compile- and run-time support to (semi-)automatically derive incremental programs from static programs [8,16,17]. In particular, *self-adjusting computation* (SAC) is a language-based approach that provides a general-purpose *change-propagation* mechanism to update the output [1]. Previous work shows that SAC can be effective in a reasonably broad range of domains, such as computational geometry [3], invariant checking [18], and machine learning [5]. In many cases, self-adjusting programs closely match or improve the asymptotic complexity achieved by algorithmic techniques, and have even helped solve challenging open problems by providing high-level reasoning for complex computations [4].

Self-adjusting programs construct and maintain a *trace* that records data and control dependencies of the computation. The trace is initially built during a run from scratch, recording the operations (e.g., that depend on the input or identify possibility of reuse) in execution order. Change-propagation edits the trace of the first run into the trace of the second run: input changes identify parts of the computation affected that must be rebuilt, while unaffected parts can be reused. This update takes time proportional to performing the new work for the updated run and discarding stale work from the previous run; there is no cost for work that is reused between runs.

Previous semantics and implementation techniques for SAC critically relied on reusing subcomputations *monotonically*, i.e., in the same order that they appear in a trace. For input changes that reorder subcomputations, however, existing change-propagation mechanisms can be grossly inefficient. As an abstract example, consider a computation that initially performs $f(x); g(y)$. After a small input change, the execution order might swap, yielding $g(y); f(x)$ instead. Under monotonic change-propagation, we could only reuse one of these functions: we can reuse $g(y)$ but would have to re-run $f(x)$, or vice versa. If both calls are expensive, neither choice will have an efficient update. In Section 2, we discuss a concrete example where non-local input changes cause computation reordering, and compare monotonic and non-monotonic change-propagation.

All previous work on SAC critically relies on monotonicity of change-propagation to ensure correctness and efficiency. Relaxing this constraint would make the technique effective for a broader class of computations, but requires overcoming three key challenges: (1) Can change-propagation be generalized to correctly support reordering? (2) How can we reason about the complexity of non-monotonic change-propagation at the program level? (3) How can non-monotonic change-propagation be realized efficiently? In this paper, we generalize SAC to support *non-monotonic reuse* where subcomputations may be reused out of order and provide complete solutions to the three challenges.

We give a high-level, direct-style source language for pure programs (Src) (Section 3) with tree-shaped traces of their execution. A formal notion of *trace distance* quantifies dissimilarity between two runs modulo reordering and abstractly measures change-propagation time. Under monotonic reuse, *local* trace distance compares two runs head-to-head in execution order to account for their differences; intuitively this is edit distance under insertions and deletions. Under non-monotonic reuse, trace distance is supplemented by a *global* trace distance that decomposes each run into a set of *trace slices* (traces with holes), pairs subcomputations from each run, and adds their local trace distance; intuitively this is local trace distance modulo reordering, akin to set difference.

We translate the source languages into a low-level, continuation-passing target language (`Tgt` in Section 4) with intrinsic support for non-monotonic change-propagation. Since continuations capture the rest of the computation, a list-shaped trace overapproximates the scope of operations that must be re-run due to inconsistencies with input changes. Since a hole in a trace slice indicates computation that has been reused out of order and the hole is labeled with its continuation, the computation can resume by running the continuation. Therefore trace slices are essential for change-propagation to support non-monotonic (i.e., out-of-order) reuse while maintaining correctness. We prove the key consistency theorem that non-monotonic change-propagation always yields results that are consistent with a from scratch run. Moreover, we show that target-level global trace distance coincides with the cost of non-monotonic change-propagation. Finally, we also prove that greedy non-monotonic reuse yields asymptotically-optimal change-propagation for a particular class of programs.

We relate the source and target languages by translation and prove that the translation preserves the semantics and trace distance (Section 5).

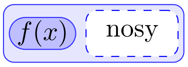
Finally, we describe how to efficiently support non-monotonic SAC (Section 6). Specifically, we give algorithms and data structures to implement trace slices and non-monotonic change-propagation, such that the source-level trace distance can be realized with a logarithmic factor overhead in the size of the trace. We defer experimental evaluation to future work. Further discussion and technical details are in the first author's dissertation [11].

2 Overview

We illustrate how non-local input changes can cause computation reordering with a pure, self-adjusting `map` program on lists:

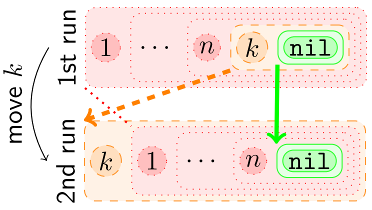
```
datatype 'a cell = nil | :: of 'a * 'a list
withtype 'a list = 'a cell ref

fun map (f : 'a -> 'b) (l : 'a list) : 'b list =
  case get l of nil  => put nil
              | h::t => put ((f h) :: (map f t))
```

We use write-once *modifiable references* (with `put` and `get` operations) for the tail to identify where input changes require new computation, and memoizing functions (declared by `fun`) to identify possible reuse across runs (Section 3). Here we explain its change-propagation and trace distance under monotonic and non-monotonic reuse. We revisit its formal trace distance in Section 3 and its performance under the change-propagation algorithm in Section 6.

A *trace* is a syntactic representation of a computation, which we depict with hierarchical box diagrams of the form: 

where the oval names the computation (e.g., $f(x)$), the inner rectangle is a hole to be filled with subtraces (e.g., recursive calls) capturing the call order, and the outer rectangle represents the local computation (i.e., between subcalls).

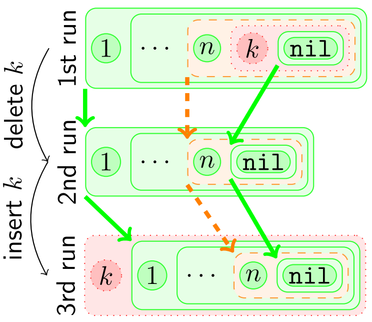
Monotonic SAC. Suppose we first map a function f on the list $[1, \ldots, n, k]$. Next, a meta-level mutator can change the input to $[k, 1, \ldots, n]$ by moving k to the front, and *change-propagate* the first run to be consistent with the new input.



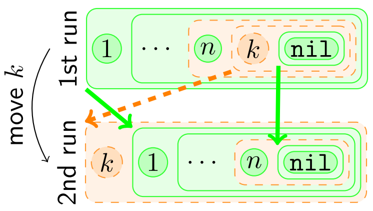
The obvious way to change the input is to splice k out of the list, reinsert it at the front, and change-propagate. The first run (top) is a from-scratch execution that constructs the trace: each rectangle represents the local work (dereference the location, apply the function to the element, place the result in a new reference) with its nested recursive call. After changing the input list, change-propagation (Subsection 4.1) uses the input change(s) to edit the trace of the first run into the trace of the second run (bottom). Since the list's head element is k instead of 1, change-propagation greedily steals the corresponding subtrace from the first run; this is a form of *partial reuse* (indicated by dashed/orange) between runs because it's the same local work but has different subcomputation. Assuming a monotonic reuse, change-propagation must discard the prefix trace $(1 \cdots n)$ from the first run in order to reuse the k subtrace, thus the work for $(1 \cdots n)$ must be done afresh for the second run; this work is *obstructed from (i.e., not available for) reuse* (indicated by dotted/red) between successive runs. Finally, the work for `nil` can be *fully reused* (indicated by solid/green) between runs. Thus change-propagation takes $O(n)$ time to update the computation, which is no more efficient than running from scratch.

Moving the last element to the front is a *non-local* change that swaps the relative order of execution between the computations for k and $(1 \cdots n)$. This is incompatible with monotonicity because work may only be reused if it occurs in the same order in both runs. Geometrically, the reuse arrows between the two traces cannot intersect.

Due to the complex semantics of change-propagation for the low-level Tgt language, we prefer to reason with an abstract *trace distance* [12] for the Src language, which quantifies the dissimilarities between runs. In Sections 4.3 and 5, we show that trace distance asymptotically coincides with the time for change-propagation. For monotonic reuse, *local* trace distance corresponds to an edit distance between traces. Intuitively, the distance between two traces is proportional to the partially reusable and discarded/fresh computation.



To improve change-propagation, we could employ a different reuse policy that instead performs the work for k afresh and reuses the work for $(1 \cdots n)$. Alternatively, we can factor the move into: (1) delete k from the list and change-propagate, then (2) reinsert it at the front and change-propagate again. Thus the bulk of the computation can be reused and change-propagation only requires $O(1)$ time to splice k out for the second run and perform the work afresh for the third run. Note that the work for $(1 \cdots n)$ and `nil` are reused monotonically. However, these solutions aren't robust enough to handle other changes such as swapping the first and second halves of the list.



Non-Monotonic SAC. In the non-monotonic setting, reusing a subtrace doesn't discard its prefix and thus change-propagation can reuse work out of order. Geometrically, non-monotonicity allows the reuse arrows to intersect, so obstructed reuse lines from the monotonic illustration become full or partial reuse arrows.

Change-propagation can greedily steal the work for k without sacrificing the prefix trace $(1 \cdots n)$, again this is partial reuse because the element has a different tail computation ($(1 \cdots n)$ instead of `nil`). Next, the subtrace for $(1 \cdots n)$ from the first run can be (almost) fully reused, except for its differing tail list (`nil` instead of k). Finally, the trace for `nil` can also be fully reused. In this example reuse is maximized, thus change-propagation takes $O(1)$ time to update the computation, an asymptotic speedup over running from scratch. Unlike the alternatives suggested above, non-monotonicity make change-propagation robust enough to handle swapping larger list segments.

For non-monotonic reuse, trace distance is a hybrid of set difference and edit distance. In particular, *global* trace distance (Subsection 3.2) allows decomposing the trace of each run into *trace slices* (traces with holes) which are then compared pairwise with local trace distance. In Section 3, we revisit this example's formal trace distance derivation. Briefly, each trace can be decomposed into separate slices for $(1 \cdots n)$, k, and `nil`. The similar slices of each run have $O(1)$ local distance because the $(1 \cdots n)$ and k slices have to account for their differing tails between runs but are otherwise identical. Thus the global distance between runs is also $O(1)$. Finally, the algorithmic overhead of non-monotonic change-propagation (Section 6) is logarithmic in the size of the trace, so an implementation would require $O(\log n)$ time to update.

3 The Src Language

The Src language serves to write pure direct-style programs that depend on input data that differs across runs, and can be compiled into equivalent self-adjusting Tgt programs (see Sections 4 and 5). The dynamic and cost semantics of Src produces an *execution trace* that can be used to determine a *trace distance* that quantifies differences between runs modulo reordering, which is asymptotically matched by the change-propagation mechanism of Tgt.

The Src language is a pure call-by-value λ-calculus with ML-style references (without update) to represent data that may change across runs.[1] The following grammar gives the syntax of types τ, expressions e, and values v, using metavariables f and x for identifiers and ℓ for locations.

$$\tau ::= \mathbf{nat} \mid \tau_{\mathrm{x}} \to \tau \mid \tau\ \mathbf{ref} \quad e ::= v \mid \mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, x.e_{\mathrm{s}} \mid e_{\mathrm{f}}\ \$\ e_{\mathrm{x}} \mid \mathbf{put}\, v \mid \mathbf{get}\, v_{\mathrm{l}}$$
$$v ::= x \mid \mathbf{zero} \mid \mathbf{succ}\, v \mid \mathbf{fun}\, f.x.e \mid \ell$$

[1] Src (and Tgt of Section 4) includes natural numbers for didactic purposes and can easily be extended with products, sums, recursive types, etc..

Function application has the usual β-reduction semantics and is additionally recorded in the execution trace to help identify similarities between runs. The τ **ref** type classifies references: $\mathbf{put}\, v$ creates a reference; $\mathbf{get}\, v_l$ dereferences and identifies the need for re-computation by recording data dependencies.

3.1 Static, Dynamic, and Cost Semantics

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. For brevity, we only give the types of the reference and suspension primitives: $\mathbf{put} : \tau \to \tau\ \mathbf{ref}$ and $\mathbf{get} : \tau\ \mathbf{ref} \to \tau$.

The dynamic and cost semantics of Src are defined by the large-step evaluation relation $\sigma; e \Downarrow T'; \sigma'; v'; c'$ to reduce expression e in store σ to value v' in updated store σ' and yields an execution *trace* T' and a *cost* c'. A store σ is a finite map from locations to values. The trace internalizes the *shape* of an evaluation derivation and will be used to identify the similarity of computations. The cost internalizes the *size* of a trace and will be used to relate the constant slowdown due to implementing suspensions with references and compiling Src programs to Tgt programs.

A *trace* T is a ε-terminated interleaving of *actions* A:

$$T ::= \varepsilon \mid A{\cdot}T \quad A ::= L \mid M(T) \quad L ::= \mathtt{put}^{v\uparrow\ell} \mid \mathtt{get}^{\ell\to v} \quad M ::= \mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v}$$

Local actions L identify where input changes cause two runs to differ because the operation yields a different result, while *memoizing actions* M delimit the trace T of an operation and identify where two runs perform similar computations. Therefore traces are necessary and sufficient to isolate the similarities and differences between program runs, without having to capture pure computation (e.g., case-analysis) because it is determined by the rest of the trace. Reference actions include allocation ($\mathtt{put}$) and dereference ($\mathtt{get}$) labeled with the location ℓ and value v involved in the operation. The function application action ($\mathtt{app}$) is labeled with a function v_f, argument v_x, and result v.

For brevity, we only show the dynamic semantics of functions and references.

$$\frac{\sigma; e_\mathrm{f} \Downarrow T_\mathrm{f}; \sigma_\mathrm{f}; \mathbf{fun}\, f.x.e; c_\mathrm{f} \quad \sigma_\mathrm{f}; e_\mathrm{x} \Downarrow T_\mathrm{x}; \sigma_\mathrm{x}; v_\mathrm{x}; c_\mathrm{x} \quad \sigma_\mathrm{x}; [\mathbf{fun}\, f.x.e/f][v_\mathrm{x}/x]e \Downarrow T'; \sigma'; v'; c'}{\sigma; e_\mathrm{f}\,\$\,e_\mathrm{x} \Downarrow T_\mathrm{f}{\cdot}T_\mathrm{x}{\cdot}(\!\mathtt{app}^{(\mathbf{fun}\, f.x.e)\$v_\mathrm{x}\Downarrow v'}(T'){\cdot}\varepsilon\!); \sigma'; v'; c_\mathrm{f} + c_\mathrm{x} + 1 + c'}$$

$$\frac{\ell \notin \operatorname{dom}\sigma \qquad \sigma' = \sigma[\ell \mapsto v]}{\sigma; \mathbf{put}\, v \Downarrow \mathtt{put}^{v\uparrow\ell}{\cdot}\varepsilon; \sigma'; \ell; 1} \qquad \frac{\ell \in \operatorname{dom}\sigma \qquad \sigma(\ell) = v}{\sigma; \mathbf{get}\, \ell \Downarrow \mathtt{get}^{\ell\to v}{\cdot}\varepsilon; \sigma; v; 1}$$

Evaluation extends the trace and increments the cost counter according to the kind of reduction. 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 it is pure computation.

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 1 for the β-reduction.

Allocation extends the store with a fresh location that is initialized with the specified value and returns the location. Dereference returns the location's value. In each case, the trace is the singleton action of the primitive, and the work is 1.

3.2 Trace Distance

To reason about the effectiveness of *monotonic* self-adjusting computation, previous work developed a notion of *trace distance* to quantify the difference between two runs [12]. 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). Under monotonic reuse, the traces produced by the dynamic semantics are compared in execution order and thus trace distance intuitively captures their edit distance.

Under non-monotonic reuse, trace distance must be generalized to account for reordering and thus trace distance is a hybrid of set difference and edit distance. Intuitively, the difference between two runs can be obtained by globally decomposing each run into a set of subcomputations and locally comparing subcomputations pairwise under some matching. More specifically, the global decomposition of a computation slices a trace into a set of traces with holes, and the local comparison of two traces alternates between *searching* for a point where traces align (i.e., at memoizing actions) and *synchronizing* the two similar traces until they again differ (i.e., at local actions).

Action slices B and *trace slices* S represent (possibly) partial computations, analogous to how actions and traces represent full computations. Thus, memoizing action slices delimit an *optional trace slice* $\dot{S}$, which can be a present subcomputation or an absent subcomputation that was reordered.

$$B ::= L \mid M(\dot{S}) \quad S ::= \varepsilon \mid B{\cdot}S \quad \dot{S} ::= \square \mid S$$

Note that a trace is also a trace slice with no holes. The notation $\overline{S}$ denotes a list of slices and the metavariable U denotes a non-empty list of traces. A memoizing action $M(T)$ can be decomposed into a (skeleton) action slice with a hole $M(\square)$ and an extracted trace T. The slicing judgement $S \gg S', \overline{S}'$ (alternatively, $S \gg U'$) extends this operation to structurally traverse the slice S and decompose it into a (skeleton) slice S' with (nondeterministically) extracted slices $\overline{S}'$:

$$\frac{}{L \gg L, \bullet} \qquad \frac{S \gg S', \overline{S}'}{M(S) \gg M(S'), \overline{S}'} \qquad \frac{S \gg S', \overline{S}'}{M(S) \gg M(\square), (M(S'){\cdot}\varepsilon, \overline{S}')}$$

$$\frac{}{M(\square) \gg M(\square), \bullet} \qquad \frac{}{\varepsilon \gg \varepsilon, \bullet} \qquad \frac{B \gg B', \overline{S}'_1 \quad S \gg S', \overline{S}'_2}{B{\cdot}S \gg B'{\cdot}S', (\overline{S}'_1, \overline{S}'_2)}$$

Intuitively, if $S \gg S', \overline{S}'$, then S' contains holes of the form $M_i(\square)$ and $\overline{S}'$ consists of trace slices $M_i(S_i){\cdot}\varepsilon$ representing the subcomputations of M_i extracted from S. Thus, replacing the corresponding holes in S' with S_i would reconsistute S.

Consider a trace slice $S[M(T)]$ that contains a deeply-nested trace $M(T)$ that could be stolen by non-monotonic memoization for out-of-order reuse. Intuitively, $S[M(T)]$ can be sliced into the trace $M(T)$ and a residual slice $S[M(\Box)]$, where the $M(\Box)$ indicates what computation was stolen. Formally, this is captured by the judgement $S[M(T)] \gg S[M(\Box)], M(T){\cdot}\varepsilon$, which can be derived by using the first two rules to structurally traverse $S[M(T)]$ until reaching the trace $M(T)$, then using the third rule to extract the trace $M(T)$. Moreover, the premise of the third rule allows further decomposing the trace T into sub-slices $\overline{S}$.

The *global distance* $S_1 \boxminus^{\gg} S_2 = d$ between two slices S_1 and S_2 is obtained by decomposing each slice into the same number of sub-slices (e.g., the $M_i(S_si)$ above), matching sub-slices from each set (the notation $i \sim j$ is a bijective pairing of indices), and adding up the local distance between each pair of sub-slices:

$$\frac{S_1 \gg \overline{S'_{1i}} \quad S_2 \gg \overline{S'_{2j}} \quad i \sim j \quad S'_{1i} \boxminus S'_{2j} = d_{\text{ij}} \quad d = \sum_{i\sim j} d_{\text{ij}}}{S_1 \boxminus^{\gg} S_2 = d}$$

Local distance is formally captured by the search distance $S_1 \boxminus S_2 = d$ and synchronization distance $S_1 \ominus S_2 = d$ judgements:

search/l/L

$$\frac{}{\varepsilon \boxminus \varepsilon = \langle 0,0\rangle} \qquad \frac{S_1 \boxminus S_2 = d}{L{\cdot}S_1 \boxminus S_2 = \langle 1,0\rangle + d}$$

search/m/L

$$\frac{S_1{\cdot}S'_1 \boxminus S_2 = d}{M(S_1){\cdot}S'_1 \boxminus S_2 = \langle 1,0\rangle + d}$$

search/none/L

$$\frac{S'_1 \boxminus S_2 = d}{M(\Box){\cdot}S'_1 \boxminus S_2 = \langle 1,0\rangle + d}$$

search/synch

$$\frac{M_1 \approx M_2 \qquad S_1 \ominus S_2 = d \qquad S'_1 \boxminus S'_2 = d'}{M_1(S_1){\cdot}S'_1 \boxminus M_2(S_2){\cdot}S'_2 = \langle 1,1\rangle + d + d'}$$

synch/l

$$\frac{}{\varepsilon \ominus \varepsilon = \langle 0,0\rangle} \qquad \frac{S_1 \ominus S_2 = d}{L{\cdot}S_1 \ominus L{\cdot}S_2 = d}$$

synch/m

$$\frac{S_1 \ominus S_2 = d \qquad S'_1 \ominus S'_2 = d'}{M(S_1){\cdot}S'_1 \ominus M(S_2){\cdot}S'_2 = d + d'}$$

synch/search

$$\frac{S_1 \boxminus S_2 = d}{S_1 \ominus S_2 = d}$$

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 S_1 and S_2 as a pair of costs, where c_1 is the amount of work in S_1 that isn't shared with S_2 and c_2 is the amount of work in S_2 that isn't shared with S_1. We let $d + d'$ denote pointwise addition for distance.

The search distance $S_1 \boxminus S_2 = d$ 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. Skipping an action in search mode incurs a cost of 1 in addition to the distance between the tail of the trace (**search/*/L** rules, the right rules are omitted). Upon simultaneously encountering similar memoizing actions $M_1(S_1){\cdot}S'_1$ and $M_2(S_2){\cdot}S'_2$ (**search/synch** rule), the search distance can switch to synchronizing the bodies S_1 and S_2, while separately searching for further synchronization of the tails S'_1 and S'_2. Two memoizing actions are *similar* $M_1 \approx M_2$ if they are both applications of the same function and argument ($M_i = \mathtt{app}^{v_f \$ v_x \Downarrow v_i}$); note that the return values need not coincide. The cost of the synchronization and search are added to the cost of 1 for the memoization match in each trace.

Turning to the synchronization distance, the $S_1 \ominus S_2 = d$ 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 local actions allows distance to remain in synchronization mode without cost (**synch/l** rule). Synchronizing memoizing actions (**synch/m** rule) requires the actions to be identical; 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 actions 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 differ (e.g., because actions don't match).

The definition of Src trace distance is a relation because of nondeterminism in how global distance slices the traces and when local distance alternates between search and synchronization mode. While it is desirable to minimize the distance between runs (and thus the update time), the dynamic semantics of Tgt has nondeterministic allocation and memoization in order to avoid committing to an implementation. We show that any distance derivable for Src programs is preserved in Tgt (Corollary 1).

Example. Returning to the map example (Section 2), if ℓ contains $h::t$, the trace slice of $\mathtt{map}(\ell)$ has the form: $\mathtt{app}^{\mathtt{map}\$\ell\Downarrow\ell'}(\mathtt{get}^{\ell\to h::t}\cdot\mathtt{app}^{\mathtt{f}\$h\Downarrow h'}(T^{f(h)})\cdot\Box\cdot\mathtt{put}^{h'::t'\uparrow\ell'})$ where the trace $T^{f(h)}$ of $f(h)$ is assumed to have $O(1)$ size, and $\Box$ is a hole for the recursive call $\mathtt{map}(t) = t'$; we abbreviate such a slice as $\mathtt{m}^{h::t}(\Box)$. Thus the traces for the two runs from the example are, (abusing notation by confusing a location with its contents): $\mathtt{m}^{1..n::k}(\mathtt{m}^{k::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}}))$ and $\mathtt{m}^{k::1}(\mathtt{m}^{1..n::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}}))$, where $\mathtt{m}^{1..n::h}(\Box)$ abbreviates $\mathtt{m}^{1::2}(\cdots\mathtt{m}^{n::h}(\Box)\cdots)$.

Under monotonic reuse, change-propagation can only do as well as the local trace distance. We assume trace distance has a bias towards synchronizing the right-hand trace (which corresponds to greedy reuse). This derivation shows that trace distance is $O(n)$, with the relevant portions underlined with the same notation as in Section 2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{\mathtt{m}}^{\mathtt{nil}} \ominus \underline{\mathtt{m}}^{\mathtt{nil}} = \langle 0, 0\rangle}\ \textbf{synch}}{\mathtt{m}^{\mathtt{nil}} \boxminus \mathtt{m}^{\mathtt{nil}} = \langle O(1), O(1)\rangle}\ \textbf{search/synch}}{\mathtt{m}^{\mathtt{nil}} \boxminus \underline{\mathtt{m}^{1..n::\mathtt{nil}}}(\mathtt{m}^{\mathtt{nil}}) = \langle O(1), O(n)\rangle}\ \textbf{search/*/R}}{\mathtt{m}^{\mathtt{nil}} \ominus \mathtt{m}^{1..n::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}}) = \langle O(1), O(n)\rangle}\ \textbf{synch/search}}{\underline{\mathtt{m}^{k::\mathtt{nil}}}(\mathtt{m}^{\mathtt{nil}}) \ominus \underline{\mathtt{m}^{k::1}}(\mathtt{m}^{1..n::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) = \langle O(1), O(n)\rangle}\ \textbf{synch}}{\mathtt{m}^{k::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}}) \boxminus \mathtt{m}^{k::1}(\mathtt{m}^{1..n::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) = \langle O(1), O(n)\rangle}\ \textbf{search/synch}}{\underline{\mathtt{m}^{1..n::k}}(\mathtt{m}^{k::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) \boxminus \mathtt{m}^{k::1}(\mathtt{m}^{1..n::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) = \langle O(n), O(n)\rangle}\ \textbf{search/*/L}$$

Read bottom up: (1) search discards $\mathtt{m}^{1..n::k}$ with $O(n)$ cost on the left; (2) $\mathtt{m}^{k::\mathtt{nil}}$ and $\mathtt{m}^{k::1}$ match with $O(1)$ cost, the synchronization is partial because the tails differ; (3) search discards $\mathtt{m}^{1..n::\mathtt{nil}}$ with $O(n)$ cost on the right; (4) and finally $\mathtt{m}^{\mathtt{nil}}$ synchronizes with $O(1)$ cost. Note that the memoizing action for the application $\mathtt{map\$}k$ appears at the head of both $\mathtt{m}^{k::\mathtt{m}^{\mathtt{nil}}}$ and $\mathtt{m}^{k::1}$, which enables switching from search to synchronization mode (*cf.* rule **memo/match** in the evaluation semantics of Tgt, Subsection 4.1). On the other hand, the local action that fetches k from the store finds differing tails ($\mathtt{m}^{\mathtt{nil}}$ and 1), which require switching back to search mode (*cf.* rule **change** in the change-propagation semantics of Tgt, Subsection 4.1).

Under non-monotonic reuse, change-propagation can do as well as the global trace distance. This derivation decomposes each run into separate trace slices for $1..n$, k, and `nil`. Since the slices are nearly identical, their distance is $O(1)$ to account for the initial synchronization and the return to search mode for the differing tails. Adding the local distances yields a global distance of $O(1)$.

$$\begin{array}{c}
\mathtt{m}^{1..n::k}(\mathtt{m}^{k::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) \gg \mathtt{m}^{1..n::k}(\square), \mathtt{m}^{k::\mathtt{nil}}(\square), \mathtt{m}^{\mathtt{nil}} \\
\mathtt{m}^{k::1}(\mathtt{m}^{1..n::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) \gg \mathtt{m}^{k::1}(\square), \mathtt{m}^{1..n::\mathtt{nil}}(\square), \mathtt{m}^{\mathtt{nil}} \\
\underline{\mathtt{m}^{\mathtt{nil}}} \boxminus \underline{\mathtt{m}^{\mathtt{nil}}} = \langle O(1), O(1) \rangle \\
\underline{\mathtt{m}^{1..n::k}}(\square) \boxminus \underline{\mathtt{m}^{1..n::\mathtt{nil}}}(\square) = \langle O(1), O(1) \rangle \\
\mathtt{m}^{\underline{k::\mathtt{nil}}}(\square) \boxminus \mathtt{m}^{\underline{k::1}}(\square) = \langle O(1), O(1) \rangle \\
\hline
\mathtt{m}^{1..n::k}(\mathtt{m}^{k::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) \boxminus^{\gg} \mathtt{m}^{k::1}(\mathtt{m}^{1..n::\mathtt{nil}}(\mathtt{m}^{\mathtt{nil}})) = \langle O(1), O(1) \rangle
\end{array}$$

4 The Tgt Language

The Tgt language is a call-by-value λ-calculus that enforces a continuation-passing style (CPS) discipline to help identify opportunities for reuse and computations for re-execution. The language includes *modifiable references* to track data dependencies and a *memoization* primitive to identify opportunities for computation reuse across runs.[2] The language is self-adjusting: its semantics includes evaluation to reduce expressions to values, and change-propagation to adapt computations to input changes. To support non-monotonic computation reuse, the dynamic semantics receives a trace of a previous run that can be sliced into subcomputations for reuse with reordering. Section 5 shows how Src programs are CPS-compiled into equivalent self-adjusting Tgt programs.

The following grammar gives the syntax of types τ, expressions e, values v, and adaptive commands κ.

$$\begin{array}{ll}
\tau ::= \mathbf{res} \mid \mathbf{nat} \mid \tau_{\mathrm{x}} \to \tau \mid \tau\,\mathbf{mod} & e ::= v \mid \mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, (x.e_{\mathrm{s}}) \mid e_{\mathrm{f}}\, v_{\mathrm{x}} \\
v ::= x \mid \mathbf{zero} \mid \mathbf{succ}\, v \mid \mathbf{fun}\, f.x.e \mid \ell \mid \kappa & \kappa ::= \mathbf{halt}\, v \mid \mathbf{memo}\, e \mid \mathbf{put}\, v\, v_{\mathrm{k}} \mid \mathbf{get}\, v_{\mathrm{l}}\, v_{\mathrm{k}}
\end{array}$$

Reference commands have an explicit continuation v_k identifying the computation that follows the command. The CPS discipline restricts a function application $e_f\, v_x$ to have a value argument. Modifiables $\tau\,\mathbf{mod}$ are mutable references

[2] Memoization in self-adjusting computation reuses computation between runs, whereas classical memoization [15] reuses results within a single run.

with commands **put** and **get** for allocation and dereference. The type **res** is an opaque answer type, while **halt** is a continuation that injects a final value into the **res** type. The dynamic semantics identifies opportunities for computation reuse at **memo** commands, which enable replaying the trace of a previous run.

4.1 Static, Dynamic, and Cost Semantics

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. For brevity, we only give the types of the adaptive commands:

$$\begin{array}{ll} \mathbf{halt} : \tau \to \mathbf{res} & \mathbf{memo} : \mathbf{res} \to \mathbf{res} \\ \mathbf{put} : \tau \to (\tau\, \mathbf{mod} \to \mathbf{res}) \to \mathbf{res} & \mathbf{get} : \tau\, \mathbf{mod} \to (\tau \to \mathbf{res}) \to \mathbf{res} \end{array}$$

The following rules give the dynamic and cost semantics of evaluation $\overline{S}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'$(left) and change-propagation $\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'$(right).

$$\frac{e \Downarrow \kappa \quad \overline{S}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'}{\overline{S}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'} \qquad \frac{\lceil S \rceil = \kappa \quad S, \overline{S}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'}{\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'}\ \textbf{change}$$

$$\frac{|\overline{S}| = c}{\overline{S}; \sigma; \mathbf{halt}\, v \Downarrow_{\mathrm{K}} \mathtt{halt}^{v}; \sigma; v; \langle c, 1\rangle} \qquad \frac{|\overline{S}| = c}{\overline{S}; \mathtt{halt}^{v}; \sigma \curvearrowright \mathtt{halt}^{v}; \sigma; v; \langle c, 0\rangle}$$

$$\textbf{memo/miss}\ \frac{\overline{S}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'}{\overline{S}; \sigma; \mathbf{memo}\, e \Downarrow_{\mathrm{K}} \mathtt{memo}^{e} \cdot T'; \sigma'; v'; \langle 0, 1\rangle + d'} \qquad \frac{\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'}{\overline{S}; \mathtt{memo}^{e} \cdot S; \sigma \curvearrowright \mathtt{memo}^{e} \cdot T'; \sigma'; v'; d'}$$

$$\textbf{memo/hit}\ \frac{\overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S_{\mathrm{e}} \quad \overline{S}'; S_{\mathrm{e}}; \sigma \curvearrowright T'; \sigma'; v'; d'}{\overline{S}; \sigma; \mathbf{memo}\, e \Downarrow_{\mathrm{K}} \mathtt{memo}^{e} \cdot T'; \sigma'; v'; \langle 1, 1\rangle + d'}$$

$$\frac{\ell \notin \mathrm{dom}\, \sigma \quad \sigma_1 = \sigma[\ell \mapsto v] \quad \overline{S}; \sigma_1; v_{\mathrm{k}}\, \ell \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'}{\overline{S}; \sigma; \mathbf{put}\, v\, v_{\mathrm{k}} \Downarrow_{\mathrm{K}} \mathtt{put}_{v_{\mathrm{k}}}^{v \uparrow \ell} \cdot T'; \sigma'; v'; \langle 0, 1\rangle + d'} \qquad \frac{\ell \notin \mathrm{dom}\, \sigma \quad \sigma_1 = \sigma[\ell \mapsto v] \quad \overline{S}; S; \sigma_1 \curvearrowright T'; \sigma'; v'; d'}{\overline{S}; \mathtt{put}_{v_{\mathrm{k}}}^{v \uparrow \ell} \cdot S; \sigma \curvearrowright \mathtt{put}_{v_{\mathrm{k}}}^{v \uparrow \ell} \cdot T'; \sigma'; v'; d'}$$

$$\frac{\ell \in \mathrm{dom}\, \sigma \quad \sigma(\ell) = v \quad \overline{S}; \sigma; v_{\mathrm{k}}\, v \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'}{\overline{S}; \sigma; \mathbf{get}\, \ell\, v_{\mathrm{k}} \Downarrow_{\mathrm{K}} \mathtt{get}_{v_{\mathrm{k}}}^{\ell \to v} \cdot T'; \sigma'; v'; \langle 0, 1\rangle + d'} \qquad \frac{\ell \in \mathrm{dom}\, \sigma \quad \sigma(\ell) = v \quad \overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'}{\overline{S}; \mathtt{get}_{v_{\mathrm{k}}}^{\ell \to v} \cdot S; \sigma \curvearrowright \mathtt{get}_{v_{\mathrm{k}}}^{\ell \to v} \cdot T'; \sigma'; v'; d'}$$

The large-step evaluation relation $\overline{S}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'$ (resp. $\overline{S}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'$) reduces the expression e (resp. the adaptive command κ) under the store σ, yielding the value v' and the updated store σ'. Evaluation also takes a list of trace slices $\overline{S}$ from a previous run which are available for reuse, and produces an execution trace T' of the current run and a pair of costs $d' = \langle c, c'\rangle$ for work c discarded from the reuse trace slices and new work c' performed for the current run. The auxiliary evaluation relation $e \Downarrow v'$ reduces an expression e to a value v' by the standard (and thus, elided) function and case-analysis β-reductions; such evaluation is pure and independent of the store.

A Tgt *trace* T is a sequence of reference and memo actions A, ending in a halt action. A *trace slice* S is a trace segment, possibly ending in a $\mathtt{hole}^{e}$ marker that indicates the rest of the trace (corresponding to the run of e) was stolen for out-of-order reuse. Note that a trace is also a trace slice without holes. $\overline{S}$ and U range over lists and non-empty lists of trace slices; concatenation extends to the first slice: $A \cdot (S, \overline{S}) = (A \cdot S, \overline{S})$.

$$A_s ::= \mathtt{put}^{v\uparrow\ell}_{v_k} \mid \mathtt{get}^{\ell\to v}_{v_k} \qquad A ::= A_s \mid \mathtt{memo}^e \quad T ::= \mathtt{halt}^v \mid A{\cdot}T$$
$$H ::= \mathtt{halt}^v \mid \mathtt{hole}^e \quad S ::= H \mid A{\cdot}S \quad \overline{S} ::= \bullet \mid S, \overline{S} \qquad U ::= S, \overline{S}$$

The **halt** v command yields a computation's final value, with a cost of 1 for the current run and a cost $c = |\overline{S}|$ summing the work discarded from the reuse trace slices $\overline{S}$, where the cost of a trace slice is the number of actions (except holes, which don't represent previous work) in the trace:

$$|\mathtt{hole}^e| = 0 \quad |\mathtt{halt}^v| = 1 \quad |A{\cdot}S| = 1 + |S|$$

An adaptive reference command uses the store (**put** and **get** rules) and passes the result to the continuation; the trace is extended with the corresponding action labeled by the location, value, and continuation, and incurs a cost of 1 for the current run. Note that it is acceptable (and, indeed, often desirable) for the location ℓ chosen by **put** to appear in the reuse trace slices because it can enable subsequent **memo**-matching on work from the previous run involving ℓ .

A memoized expression **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 $\mathtt{memo}^e$, incurring a cost of 1 for the current run. The **memo/hit** rule exploits the reuse trace from the previous evaluation and switches to change-propagation if the same expression was memoized and evaluated in the previous run.

The memoization judgement $S; e \overset{\mathrm{m}}{\rightsquigarrow} S'_1; S'_{\mathrm{e}}$ splits the reuse trace S into a suffix trace slice S'_e that corresponds to a (partial) previous run of e (under a (possibly) different store), and a prefix trace S'_1 of the work preceding S'_e with an explicit $\mathtt{hole}^e$ end marker to indicate the stolen tail.

$$\frac{S; e \overset{\mathrm{m}}{\rightsquigarrow} S'; S'_{\mathrm{e}}}{A{\cdot}S; e \overset{\mathrm{m}}{\rightsquigarrow} A{\cdot}S'; S'_{\mathrm{e}}} \qquad \textbf{hit}\ \frac{}{\mathtt{memo}^e{\cdot}S_{\mathrm{e}}; e \overset{\mathrm{m}}{\rightsquigarrow} \mathtt{hole}^e; S_{\mathrm{e}}} \qquad \frac{\overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S'_{\mathrm{e}}}{S, \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} S, \overline{S}'; S'_{\mathrm{e}}} \qquad \frac{S; e \overset{\mathrm{m}}{\rightsquigarrow} S'; S'_{\mathrm{e}}}{S, \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} S', \overline{S}; S'_{\mathrm{e}}}$$

Under monotonic memoization the prefix S'_1 would be discarded incurring a cost of $|S'_1|$, but under non-monotonicity it remains available for later reuse. Memoization extends to trace lists $\overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S'_{\mathrm{e}}$ by memo-matching with one trace from the list.

The change-propagation relation $\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'$ replays the partial execution trace S 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 $S, \overline{S}$ (viz. the dotted/red work from the previous run's trace) and new work c' performed for T' (viz. the dotted/red and dashed/orange work for the new run's trace); the additional reuse traces $\overline{S}$ are other computations from the previous run that may be reused if change-propagation returns to evaluation. Any work that can be replayed from the previous run is free (viz. the solid/green work common to both traces). A halt action can be replayed to obtain the (unchanged) final value, incurring the cost of discarding the additional reuse traces. An adaptive action can be replayed without cost if the action is consistent with the current store, the tail of the trace can be recursively change-propagated and then extended with the same action. However, if a reference action is inconsistent with the store (e.g., a specific location can't be allocated or a dereference fetches a different value), then change-propagation must switch back to evaluation. A trace slice S can be *reified* back into an adaptive command $\kappa = \lceil S \rceil$, the tail trace slice S' (if any) can be ignored because adaptive actions capture the rest of the computation in the continuation:

$$\lceil \mathtt{halt}^{v} \rceil = \mathbf{halt}\, v \qquad \lceil \mathtt{hole}^{e} \rceil = \mathbf{memo}\, e \quad \lceil \mathtt{memo}^{e} {\cdot} S' \rceil = \mathbf{memo}\, e$$
$$\lceil \mathtt{put}^{v \uparrow \ell}_{v_{\mathrm{k}}} {\cdot} S' \rceil = \mathbf{put}\, v\, v_{\mathrm{k}} \quad \lceil \mathtt{get}^{\ell \to v}_{v_{\mathrm{k}}} {\cdot} S' \rceil = \mathbf{get}\, \ell\, v_{\mathrm{k}}$$

Thus, change-propagation can reify an inconsistent trace slice S and re-evaluate the command, while keeping the trace S for possible reuse later (**change** rule). Note that the reified **put** (resp. **get**) 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.

4.2 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 of type $\Sigma \uplus \Sigma_1$. We can evaluate e under the store σ_1 and no reuse traces, yielding the initial result v'_1 and a trace T'_1: $\bullet; \sigma_1; e \Downarrow_{\mathrm{E}} \sigma'_1; v'_1; T'_1; d'_1$. After this initial evaluation, we can consider another store σ_2 of type $\Sigma \uplus \Sigma_2$ and update the output of the evaluation with respect to this store by applying change-propagation to T'_1 under the store σ_2: $\bullet; T'_1; \sigma_2 \curvearrowright T'_2; \sigma'_2; v'_2; d'_2$. The consistency of change-propagation asserts that the result and trace obtained by change-propagation are identical to those obtained by from-scratch evaluation (i.e., without any reuse traces). In the presence of non-monotonic memoization the reuse trace may be sliced, so consistency must be generalized to deal with trace slices and employs the auxiliary judgements S wfwrt e to mean S results from slicing a from-scratch execution of e ($\bullet; \_; e \Downarrow_{\mathrm{E}} T'; \_; \_; \_$ and $T'; e \overset{\mathrm{m}}{\rightsquigarrow} S; S'_{\mathrm{e}}$), and S wf to mean S wfwrt e for some e. Consistency is a corollary of the following theorem by instantiating $\overline{S}$ as the empty list and S'_1 as T'_1.

Theorem 1 (Consistency of Change-Propagation). *If* $\overline{S\ \mathsf{wf}}$, S'_1 wfwrt e, *and* $\overline{S}; S'_1; \sigma_2 \curvearrowright T'_2; \sigma'_2; v'_2; \_$, *then* $\bullet; \sigma_2; e \Downarrow_{\mathrm{E}} T'_2; \sigma'_2; v'_2; \_$.
If $\overline{S\ \mathsf{wf}}$ *and* $\overline{S}; \sigma_2; e \Downarrow_{\mathrm{E}} T'_2; \sigma'_2; v'_2; \_$, *then* $\bullet; \sigma_2; e \Downarrow_{\mathrm{E}} T'_2; \sigma'_2; v'_2; \_$.

4.3 Trace Distance

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 Src, in Section 5 we show that they are asymptotically the same.

The $S \gg U'$ judgement splits a Tgt trace slice S into a non-empty list of slices U' by (non-deterministically) replacing memo actions with holes.

$$\frac{}{H \gg H; \bullet} \qquad \frac{S \gg S'; \overline{S}'}{A{\cdot}S \gg A{\cdot}S'; \overline{S}'} \qquad \frac{S \gg S'; \overline{S}'}{\mathtt{memo}^{e}{\cdot}S \gg \mathtt{hole}^{e}; \mathtt{memo}^{e}{\cdot}S', \overline{S}'}$$

The judgement extends to decomposing lists of slices $U \gg U'$ by appending the decomposition of each slice in the list. The judgement $U \overset{\pi}{\rightsquigarrow} U'$ means U' is a permutation of U.

The *global (search) distance* $U_1 \boxminus^{\gg} U_2 = d$ of two slice lists U_1 and U_2 results from slicing and permuting each list, and taking their local search distance.

$$\frac{U_1 \gg U_1' \qquad U_1' \overset{\pi}{\rightsquigarrow} U_1'' \qquad U_2 \gg U_2' \qquad U_2' \overset{\pi}{\rightsquigarrow} U_2'' \qquad U_1'' \boxminus U_2'' = d}{U_1 \boxminus^{\gg} U_2 = d}$$

Since global distance accounts for computation reordering, the *local search distance* $U_1 \boxminus U_2 = d$ accounts for differences between traces in order until it finds matching memoization actions, then it can use the *local synchronization distance* $U_1 \ominus U_2 = d$ to account for reuse between traces until they differ, at which point it must return to search distance. The distance $d = \langle c_1, c_2 \rangle$ quantifies the cost c_1 of work in U_1 that isn't shared with U_2 and the cost c_2 of work in U_2 that isn't shared with U_1. Analogous to the dynamic semantics of Tgt, search distance accounts for discarding old work on the left and performing new work on the right, while synchronization distance reuses work between runs.

$$\frac{|H_1| = c_1 \qquad |H_2| = c_2}{H_1;\bullet \boxminus H_2;\bullet = \langle c_1, c_2 \rangle}$$

h/L

$$\frac{|H_1| = c_1 \qquad S_1;\overline{S}_1 \boxminus U_2 = d}{H_1; S_1, \overline{S}_1 \boxminus U_2 = \langle c_1, 0 \rangle + d}$$

$$\frac{}{\mathtt{halt}^v;\bullet \ominus \mathtt{halt}^v;\bullet = \langle 0, 0 \rangle}$$

$$\frac{S_1;\overline{S}_1 \boxminus U_2 = d}{A{\cdot}S_1;\overline{S}_1 \boxminus U_2 = \langle 1, 0 \rangle + d}\ \textbf{a/L}$$

$$\frac{S_1;\overline{S}_1 \ominus S_2;\overline{S}_2 = d}{A{\cdot}S_1;\overline{S}_1 \ominus A{\cdot}S_2;\overline{S}_2 = d}$$

$$\frac{S_1;\overline{S}_1 \ominus S_2;\overline{S}_2 = d}{\mathtt{memo}^e{\cdot}S_1;\overline{S}_1 \boxminus \mathtt{memo}^e{\cdot}S_2;\overline{S}_2 = \langle 1, 1 \rangle + d}\ \textbf{memo/hit}$$

$$\frac{U_1 \boxminus U_2 = d}{U_1 \ominus U_2 = d}$$

The search distance between halt or hole actions is the length of each action. Skipping an action incurs a cost of the length of the action for the corresponding trace and forces distance to remain in search mode (***/L** rules, the right rules are omitted). Two identical memo actions incur a cost of 1 each and enable switching from search to synchronization mode.

Synchronization distance, as in Src, is only meant to be used on traces generated by the evaluation of the same expression under (possibly) different stores (though synchronization distance exists 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.

The following shows that the distance between a program's trace T and some traces $\overline{S}$ coincides with the cost of evaluating the program with reuse traces $\overline{S}$.

Theorem 2 (Dynamic Semantics Coincides with Distance). *If* $\overline{S}$ wf*, and* $\bullet;\sigma;e \Downarrow_{\mathrm{E}} T';\sigma';v';\_$*, then* $\overline{S} \boxminus^{\gg} T' = d$ *iff* $\overline{S};\sigma;e \Downarrow_{\mathrm{E}} T';\sigma';v';d$*.*

The following result shows that for pure computations with unique function calls, greedy non-monotonic reuse is optimal in the sense that it achieves minimal trace distance. The uniqueness condition means that an application $e_f \$ e_x$ with a given function e_f and argument e_x occurs at most once during the execution. This assumption is necessary because in the presence of duplicate calls and nondeterministic allocation, greedily stealing a computation may unnecessarily cause computation to become inconsistent. The purity assumption is necessary because effects can introduce dependencies between computations that incur an additional cost to reorder (see Section 6).

Theorem 3 (Optimality of Greediness). *Given two pure computations with unique function calls, greedy memo-matching is an optimal memoization policy that change-propagates with asymptotically minimal distance.*

Proof. By the uniqueness assumption, greedy memo-matching achieves maximal reuse of the computation, whence the Tgt-level distance is minimized and in turn the Src-level distance is minimized, up to a constant factor.

5 Translation

In this section, we describe a semantics- and trace distance-preserving translation from Src to Tgt

To translate from Src to Tgt, we use an *adaptive continuation-passing style* transformation. The explicit continuation helps identify the scope of inconsistent store actions that need to be re-executed as well as identical memoized computations that can be reused. That translation was previously used for monotonic self-adjusting computation with traces and local trace distance [12]; we exploit its robustness to extend it to the non-monotonic setting by generalizing to trace slices and global trace distance.

Program Translation. To establish the semantic connection, we define translation for types $[\![\tau^{\mathsf{src}}]\!] = \tau^{\mathsf{tgt}}$, expressions $[\![e^{\mathsf{src}}]\!]\ v_k^{\mathsf{tgt}} = e^{\mathsf{tgt}}$ with an explicit Tgt-level continuation v_k^{tgt}, values $[\![v^{\mathsf{src}}]\!] = v^{\mathsf{tgt}}$. The translation is a standard CPS conversion except that store primitives are translated into Tgt store commands with an explicit continuation v_k, and the function translation threads the continuation through the store and uses explicit **memo** operations before and after the function body to isolate the function call from the rest of the computation.

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), which are straightforward adaptations of the proofs for the monotonic variant of Tgt.

Theorem 4 (Static and Dynamic Preservation). *If* $\Sigma; \Gamma \vdash e : \tau$, *and* $[\![\Sigma]\!]; [\![\Gamma]\!], \Gamma' \vdash v_{\mathrm{k}} : [\![\tau]\!] \to \mathbf{res}$, *then* $[\![\Sigma]\!]; [\![\Gamma]\!], \Gamma' \vdash [\![e]\!]\ v_{\mathrm{k}} : \mathbf{res}$.
If $\sigma_0; e_0 \Downarrow T; \sigma_1; v_1; c_0$, *and* $\bullet; [\![\sigma_1]\!] \uplus \sigma_{\mathrm{k}}; v_{\mathrm{k}}\ [\![v_1]\!] \Downarrow_{\mathrm{E}} T_{\mathrm{k}}; \sigma_2; v_2; \langle \_, c_1 \rangle$, *then* $\bullet; [\![\sigma_0]\!] \uplus \sigma_{\mathrm{k}}; [\![e_0]\!]\ v_{\mathrm{k}} \Downarrow_{\mathrm{E}} T'; \sigma_2 \uplus \sigma_{\mathrm{e}}; v_2; \langle \_, \Theta(c_0 + c_1) \rangle$.

Trace Translation. To establish the trace distance connection, we define a *trace translation* $[\![S^{\mathsf{src}}]\!]\ v_k^{\mathsf{tgt}}\ U_k^{\mathsf{tgt}} = U^{\mathsf{tgt}}$ of a Src trace slice S^{src} using v_k^{tgt} as an initial continuation and suffix slice list U_k^{tgt} to produce a Tgt slice list U^{tgt} corresponding to the original computation (with explicit holes). The proof of global trace distance preservation requires establishing the preservation of local trace distance, which in turn requires auxiliary translations for a trace slice S^{src} extracted from a larger computation and for non-empty Src slice list U^{src}.

Corollary 1 ($\mathsf{Src}/\mathsf{Tgt}$ **Distance Soundness).** *If* $S_1^{\mathsf{imp}} \boxminus\!\!\gg S_2^{\mathsf{imp}} = \langle \_, c\rangle$, *then* $\left[\!\!\left[S_1^{\mathsf{imp}}\right]\!\!\right]\ \mathbf{id}_1\ U_{\mathbf{id}1}^{\mathsf{tgt}} \boxminus\!\!\gg \left[\!\!\left[S_2^{\mathsf{imp}}\right]\!\!\right]\ \mathbf{id}_2\ U_{\mathbf{id}2}^{\mathsf{tgt}} = \langle \_, \Theta(c)\rangle$, *where* $U_{\mathbf{id}i}^{\mathsf{tgt}}$ *is the identity trace.*

Note that since Src and Tgt distance are quasi-symmetric, analogous results hold of the left component of distance. This means that change-propagation has the same asymptotic time-complexity as trace distance.

6 The Change Propagation Algorithm

Here we describe a concrete algorithm and associated data structures for efficiently supporting the reordering of the trace. This goes into a level more detail than the target semantics in Section 4 allowing an analysis of running time. We use CPA to refer to the change propagation algorithm in contrast to the abstract change propagation mechanism of Section 4. We use TDS to refer to the concrete data structure used for traces generated during the run of the program and updated by the CPA.

The main idea of the CPA is to traverse the trace in execution order while identifying the parts of the trace that need to be rerun (the $\Downarrow_{\mathrm{E}}$ and $\Downarrow_{\mathrm{K}}$ relations in Subsection 4.1) and the parts that can be reused (the $\curvearrowright$ relation in Subsection 4.1). In particular it is important to skip over the part that can be reused without incurring any cost. An important aspect is therefore to identify after a memo hit the next place in the trace that does not match the previous trace—i.e., the next inconsistency. Once this is identified the CPA also needs to splice the part between the match and the inconsistency out of the previous TDS and append it to the current TDS.

The TDS is based on a totally ordered timeline with a timestamp for each action in the trace—i.e., all memo and reference actions. This timeline therefore has a one-to-one correspondence to the trace in the target semantics. The TDS also maintains for each modifiable reference the timestamps for all actions on the reference, and for each **get** action it keeps the continuation that needs to be rerun if the value of the reference is changed. To support reordering this timeline needs to allow extraction and insertion of chunks of trace. As discussed below, this can be implemented reasonably efficiently. Finally the TDS needs to maintain a memo table mapping all memoized function calls and associated arguments to the timestamp at which the call is made. Here we assume that if there are multiple identical calls, only one is stored.

```
Algorithm CPA ($S$, $T$, $Q$, $t_s$)
    let $t_i$ = find the next element in $Q$ greater than $t_s$
    in if $t_i$ is the end then $T$ ++ $S[t_s$,end]
       else let $T_r = S[t_s,t_i)$
                $S' = S - T_r$
                $(t_m, Q', T_n)$ = run continuation of $t_i$ until memo match in $S'$
                   $t_m$ is the timestamp of the memo match
                   $Q'$ is $Q$ extended such that every put($\ell$) during
                      the run adds all associated get($\ell$)s to the queue
                   $T_n$ is the new trace
                $T' = T$ ++ $T_r$ ++ $T_n$
            in if $t_m$ is the end then $T'$
               else CPA ($S'$, $T'$, $Q'$, $t_m$)
```

Fig. 1. The non-monotonic change propagation algorithm

Figure 1 describes the non-monotonic CPA. The algorithm starts with an input trace S (i.e., the list of trace slices $\overline{S}$ in the Tgt semantics, but the separation into pieces is implicit) and generates an output trace T for the updated run. The algorithm maintains a queue Q of the timestamps of inconsistent reads (**get** actions for which the value of the corresponding reference has changed), ordered by time. The queue is initialized to include all the **get** actions on any input references that have changed. The time t_s represents a finger (position) in S which is the start of a piece of trace that is being reused. Initially, t_s is at the start of S; at each step (recursive call), the algorithm finds the next inconsistent read past t_s. If there is none, then there are no more inconsistencies and the algorithm is done by appending the trace in S past the finger onto the end of T. If the next inconsistent read is at time t_i, CPA extracts the part of the trace between t_s (inclusive) and t_i (exclusive) because it hasn't changed since the last run and can be reused by simply appending it to the output trace (skipping the $\curvearrowright$ replay transitions). This chunk is also removed from the input trace since we don't want to use the same part of the input trace more than once.

Since the read at t_i is inconsistent (reads a different value from before) the algorithm needs to rerun the continuation for that read. While the continuation runs it looks for a memo match in S and stops when it finds one. This match could be anywhere in S, and in particular out of order with respect to matches found in previous steps. While running, whenever a change is made to a reference that existed in the previous run (a write with a new value), the timestamps for all the reads associated with that reference are added to Q'. Thus when the rerun is completed, all inconsistent reads caused by the run are properly marked in Q' and all memoized function calls are placed in the memo table for future reference. The rerun returns the timestamp t_m of the memo match, as well as the modified queue Q' and the new trace segment T_n for the computation that has just run. Now CPA can extend the original output trace T with the reusable trace T_r and the new trace T_n. Thus on every step (except perhaps the last), the algorithm adds one reused chunk of trace and one new chunk of trace to the output trace. Only the new chunks require work.

This algorithm implements the change propagation scheme described in Section 4 and is therefore correct as long as it properly identifies the **change** rule from the Tgt dynamic semantics—i.e., it properly identifies the next difference in the trace. This identification is correct since the only way a **get** of a pure reference from the source language can become inconsistent (read a different value) is if the original **put** has changed. These reference updates are all included in Q. The important property is that any reordering among the reads does not affect the values read since the write happens before all reads. Also the order of a read and write cannot swap since that would be an invalid program and would not be generated by any trace. This is not true for imperative source references, where there can be interleaving between writes and reads and a reordering of traces can swap the ordering of a read and write.

Now let's consider the running time of CPA. Certainly all new computation needs to be run but this is accounted for in the trace distance. The other costs of the algorithm include the time for extracting and appending chunks of the trace, the cost for the queue operations, and the cost for memo lookup and associated insertion into the memo table. We use $T_{splice}(n)$ to indicate the time to append or extract a chunk of trace for a trace of size n. Using balanced trees this can easily be implemented in $O(\log n)$ time, and with some work comparisons between timestamps in the trace can be made to work in $O(1)$ time. We use $T_{queue}(n)$ to indicate the time to insert or delete in the queue of size n. This is easy to implement in $O(\log n)$ time per operation as long as the comparison of time stamps is $O(1)$ time. We assume the memo lookup uses standard hash tables and therefore takes constant expected time per operation (either lookup or insertion). Consider a computation in which the total new computation is c, the total number of recursive calls of the CPA is l, the total trace distance just counting reads is r, and the maximum of the sizes of the input and output traces is n. The running time is then $O(c + lT_{splice}(n) + (r+l)T_{queue}(n))$. Relating this to the trace distance measured by the semantics, change propagation for two traces S_1 and S_2 such that $S_1 \ominus^{\gg} S_2 = \langle c_1, c_2 \rangle$ will run in time $O((c_1 + c_2)(1 + T_{splice}(n) + T_{queue}(n))) = O((c_1 + c_2)\log n)$.

Example. The Tgt trace of map has the form (abbreviations given below):

$$\underbrace{\mathtt{call}^{\mathtt{map}\$\ell\Downarrow\ell'}}_{\overset{h}{\llcorner}}\cdot\underbrace{\mathtt{get}^{\ell\to h::t}}_{\mathtt{g}}\cdot\underbrace{\mathtt{call}^{f\$h\Downarrow h'}\cdot T^{f(h)}\cdot\mathtt{ret}^{f\$h\Uparrow h'}}_{F^h}\cdot\underbrace{\square}_{\square}\cdot\underbrace{\mathtt{put}^{h'::t'\uparrow\ell'}}_{\mathtt{p}}\cdot\underbrace{\mathtt{ret}^{\mathtt{map}\$\ell\Uparrow\ell'}}_{\overset{h}{\lrcorner}}$$

where $T^{f(h)}$ is the body of $f(h)$ and $\square$ is a hole for the recursive call $\mathtt{map}(t) = t'$.[3] The trace segments $\mathtt{call}^{g\$x\Downarrow a}$ and $\mathtt{ret}^{g\$x\Uparrow a}$ represent the *memoized* function call and return that result from translating a Src trace $\mathtt{app}^{g\$x\Downarrow a}(\_)$; they (1) enable reusing the subsequent trace up to the next inconsistent action and (2) identify an inconsistency (i.e., need to re-execute at the return) if the function is being reused in a different calling context (i.e., returning to a different continuation).

[3] For brevity, we omit the Tgt continuations on actions (e.g., a call has a continuation argument, a return passes the result to the continuation).

Next, we consider the CPA updating `map`. The table below shows the iterations of CPA with the reuse trace S and the trace T of the new run as it is built.

iteration	first run (S)	second run (T)
1	$\overset{1}{\llcorner}\mathtt{g}F^1\cdots\overset{n}{\llcorner}\underset{.}{\mathtt{g}}F^n\underline{\overset{k}{\llcorner}}\underset{.}{\mathtt{g}}F^k\overset{\mathtt{nil}}{\llcorner}\mathtt{gp}\overset{\mathtt{nil}}{\lrcorner}\mathtt{p}\underset{.}{\overset{k}{\lrcorner}}\mathtt{p}\overset{n}{\lrcorner}\cdots\mathtt{p}\overset{1}{\lrcorner}$	$\underline{\overset{k}{\llcorner}}\underset{.}{\mathtt{g}}$
2	$\overset{1}{\llcorner}\mathtt{g}F^1\cdots\overset{n}{\llcorner}\underset{.}{\mathtt{g}}F^n\quad\underset{--}{F^k}\overset{\mathtt{nil}}{\llcorner}\mathtt{gp}\overset{\mathtt{nil}}{\lrcorner}\mathtt{p}\underset{.}{\overset{k}{\lrcorner}}\mathtt{p}\overset{n}{\lrcorner}\cdots\mathtt{p}\overset{1}{\lrcorner}$	$\underset{--}{F^k}$
3	$\underline{\overset{1}{\llcorner}\mathtt{g}F^1\cdots\overset{n}{\llcorner}}\underset{.}{\mathtt{g}}F^n\quad\overset{\mathtt{nil}}{\llcorner}\mathtt{gp}\overset{\mathtt{nil}}{\lrcorner}\mathtt{p}\underset{.}{\overset{k}{\lrcorner}}\mathtt{p}\overset{n}{\lrcorner}\cdots\mathtt{p}\underset{.}{\overset{1}{\lrcorner}}$	$\underline{\overset{1}{\llcorner}\mathtt{g}F^1\cdots\overset{n}{\llcorner}}\underset{.}{\mathtt{g}}$
4	$\underset{--}{F^n}\quad\overset{\mathtt{nil}}{\llcorner}\mathtt{gp}\overset{\mathtt{nil}}{\lrcorner}\mathtt{p}\underset{.}{\overset{k}{\lrcorner}}\mathtt{p}\overset{n}{\lrcorner}\cdots\mathtt{p}\underset{.}{\overset{1}{\lrcorner}}$	$\underset{--}{F^n}$
5	$\underline{\overset{\mathtt{nil}}{\llcorner}\mathtt{gp}}\underset{..}{\overset{\mathtt{nil}}{\lrcorner}}\mathtt{p}\underset{.}{\overset{k}{\lrcorner}}\mathtt{p}\overset{n}{\lrcorner}\cdots\mathtt{p}\underset{.}{\overset{1}{\lrcorner}}$	$\underline{\overset{\mathtt{nil}}{\llcorner}\mathtt{gp}}\underset{....}{\overset{\mathtt{nil}}{\lrcorner}\mathtt{p}}$
6	$\mathtt{p}\underset{.}{\overset{k}{\lrcorner}}\underline{\mathtt{p}\overset{n}{\lrcorner}\cdots\mathtt{p}}\underset{.}{\overset{1}{\lrcorner}}$	$\underline{\overset{n}{\lrcorner}\cdots\mathtt{p}}\underset{...}{\overset{1}{\lrcorner}\mathtt{p}\overset{k}{\lrcorner}}$

The queue Q consists of inconsistent reads (e.g., `g`) due to input changes and inconsistent returns (e.g., $\overset{n}{\lrcorner}$ and the return at the end of F_n) because the calling context (i.e., caller) has changed. We use $\underset{\cdots\cdots}{\text{dotted/red}}$ in S for inconsistent actions and in T for new work (viz. T_n), $\underset{------}{\text{dashed/orange}}$ in S for a partially inconsistent trace and in T for partially reused work (viz. $T_r +\!\!+\ T_n$), and $\underline{\text{solid/green}}$ in S and T for the reused trace (viz. T_r).

The initial `map` on $[1, \ldots, n, k]$ produces the first trace S. Moving k to the front changes the input to $[k, 1, \ldots, n]$, and Q is initialized with the now-inconsistent `get` actions for k and n. In the first CPA iteration, $\overset{k}{\llcorner}$ is reused and the following `g` is re-run because it's inconsistent and immediately followed by a memo-match in F^k; in S, the return $\overset{k}{\lrcorner}$ is marked inconsistent because of the new caller (originally called from $\overset{n}{\llcorner}$, but now from the top-level) and the consumed trace segments are removed (indicated by blanks in the next iteration). In the second iteration, $F^k = \underline{\mathtt{call}^{f\$h\Downarrow h'}\cdot T^{f(h)}}\cdot\underset{\cdots\cdots}{\mathtt{ret}^{f\$h\Uparrow h'}}$ reuses the call and body, but re-runs the tail because of the different tail computation (`map`$\$[1, \ldots]$ instead of `map$nil`). The third and fourth iterations likewise reuse the `map` and f calls for $1..n$ and mark $\overset{1}{\lrcorner}$ inconsistent because of the different caller. The fifth iteration reuses the call and body for `nil`, but has to re-execute the return $\overset{\mathtt{nil}}{\lrcorner}$ and `p` of n because of the new caller. Finally, in the sixth iteration, the `map` returns of $n..2$ are reused, and the returns $\overset{1}{\lrcorner}$ and $\overset{k}{\lrcorner}$ are re-run because they have new callers. The reuse trace S is left over with unused remnants $\mathtt{p}\underset{.}{\overset{k}{\lrcorner}}$ and $\underset{.}{\overset{1}{\lrcorner}}$ which must be discarded.

7 Related Work

Self-adjusting computation has been realized through several formal languages and implementations. The first was a pure higher-order language with a modal type system that was implemented both as a Standard ML library with a monad and explicit destination-passing [2] and a Haskell library using several monads to enforce the modal constraints [6]. Subsequent proposals included a direct-style higher-order language compiled into a continuation-passing style (CPS) higher-order language implemented in the MLton Standard ML compiler [13], and a low-level imperative language implemented as a compiler for C [10]. All of these designs focus on *strict* languages with *call-by-value* (CBV) functions that *eagerly* evaluate function arguments[4] and none of them supported efficient reordering. Approaches based on pure memoization (function caching) alone [16,14] allow for incrementality with reordering; since they lack the fine-grained dependence tracking of modifiable references, they can only provide coarse-grained reuse and are inefficient for deeply-nested changes (e.g., changing the last element of a list). Previous work introduced a cost semantics for self-adjusting computation with updatable references and monotonic reuse, and showed analogous correctness properties of change-propagation and compilation [12].

8 Conclusion and Future Work

Self-adjusting computation (SAC) combines dynamic dependence tracking and memoization to effectively update a computation in response to input changes. However, since previous approaches are based on updating a timeline of the computation in monotonic (i.e., time-increasing) order and a greedy approach to memo matching, they perform inefficiently when subcomputations are reordered.

We generalize SAC with non-monotonic reuse to support input changes that affect the order of subcomputations. We give a high-level source language for expressing pure self-adjusting programs equipped with a notion of trace distance to quantify the dissimilarity of computations under an input change. We give a semantics- and trace distance-preserving translation to a low-level target language and show that trace distance coincides asymptotically with change-propagation (i.e., update). We also provide and analyze a new algorithm that realizes the semantics of change-propagation with reordering, which incurs a logarithmic overhead. In future work, we will evaluate the algorithm and extend non-monotonicity to other programming paradigms (e.g., updatable references and laziness).

References

1. Acar, U.A., Blelloch, G.E., Blume, M., Tangwongsan, K.: An experimental analysis of self-adjusting computation. In: PLDI (2006)
2. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM TOPLAS 28(6), 990–1034 (2006)
3. Acar, U.A., Blelloch, G.E., Tangwongsan, K., Türkoğlu, D.: Robust kinetic convex hulls in 3D. In: European Symposium on Algorithms (September 2008)
4. Acar, U.A., Cotter, A., Hudson, B., Türkoğlu, D.: Dynamic well-spaced point sets. In: Symposium on Computational Geometry (2010)
5. Acar, U.A., Ihler, A., Mettu, R., Sümer, Ö.: Adaptive Bayesian inference. In: Neural Information Processing Systems, NIPS (2007)

[4] Haskell is lazy, but the use of monads gives SAC primitives eager evaluation.

6. Carlsson, M.: Monads for incremental computing. In: ICFP (2002)
7. Chiang, Y.-J., Tamassia, R.: Dynamic algorithms in computational geometry. Proceedings of the IEEE 80(9), 1412–1434 (1992)
8. Demers, A., Reps, T., Teitelbaum, T.: Incremental evaluation of attribute grammars with application to syntax-directed editors. In: POPL (1981)
9. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Atallah, M.J. (ed.) Algorithms and Theory of Computation Handbook, ch.8, CRC Press (1999)
10. Hammer, M.A., Acar, U.A., Chen, Y.: CEAL: a C-based language for self-adjusting computation. In: PLDI (2009)
11. Ley-Wild, R.: Programmable Self-Adjusting Computation. PhD thesis, CSD, CMU (2010)
12. Ley-Wild, R., Acar, U.A., Fluet, M.: A cost semantics for self-adjusting computation. In: POPL (2009)
13. Ley-Wild, R., Fluet, M., Acar, U.A.: Compiling self-adjusting programs with continuations. In: ICFP (2008)
14. Liu, Y.A., Stoller, S., Teitelbaum, T.: Static caching for incremental computation. ACM TOPLAS 20(3), 546–585 (1998)
15. Michie, D.: "Memo" functions and machine learning. Nature 218, 19–22 (1968)
16. Pugh, W., Teitelbaum, T.: Incremental computation via function caching. In: POPL (1989)
17. Ramalingam, G., Reps, T.: A categorized bibliography on incremental computation. In: POPL (1993)
18. Shankar, A., Bodik, R.: DITTO: Automatic incrementalization of data structure invariant checks (in Java). In: PLDI (2007)