Imperative Self-Adjusting Computation

Umut A. Acar Amal Ahmed Matthias Blume

Toyota Technological Institute at Chicago

{umut,amal,blume}@tti-c.org

Abstract

Recent work on self-adjusting computation showed how to systematically write programs that respond efficiently to incremental changes in their inputs. The idea is to represent changeable data using *modifiable references*, i.e., a special data structure that keeps track of dependencies between *read* and *write*-operations, and to let computations construct *traces* that later, after changes have occurred, can drive a *change propagation* algorithm. The approach has been shown to be effective for a variety of algorithmic problems, including some for which ad-hoc solutions had previously remained elusive.

All previous work on self-adjusting computation, however, relied on a *purely functional* programming model. In this paper, we show that it is possible to remove this limitation and support modifiable references that can be written multiple times. We formalize this using a language AIL for which we define evaluation and change-propagation semantics. AIL closely resembles a traditional higher-order imperative programming language. For AIL we state and prove *consistency*, i.e., the property that although the semantics is inherently non-deterministic, different evaluation paths will still give observationally equivalent results. In the imperative setting where pointer graphs in the store can form cycles, our previous proof techniques do not apply. Instead, we make use of a novel form of a step-indexed logical relation that handles modifiable references.

We show that AIL can be realized efficiently by describing implementation strategies whose overhead is provably constant-time per primitive. When the number of *reads* and *writes* per modifiable is bounded by a constant, we can show that change propagation becomes as efficient as it was in the pure case. The general case incurs a slowdown that is logarithmic in the maximum number of such operations. We use DFS and related algorithms on graphs as our running examples and prove that they respond to insertions and deletions of edges efficiently.

1. Introduction

Incremental computation concerns the problem of updating the output of a computation while its input undergoes changes. Recent work on self-adjusting computation showed that a combination of dynamic dependence graphs and a particular form of memoization can be combined to achieve efficient update times for a reasonably broad range of applications including those that involve continuously moving objects [3, 4]. These results demonstrate that the approach can yield orders of magnitude of speedup over recomputing from scratch when the computation data changes slowly over time. Other implementations of self-adjusting-computation have been given by Carlsson [13] in the Haskell language and more recently by Shankar and Bodik [25] in Java. The purpose of Shankar and Bodik's work was incremental invariant checking. It proved to be effective in delivering orders of magnitude speedups compared to non-incremental approaches [25]. There are several examples where, by taking advantage of the high-level abstract way of designing, analyzing, and implementing incremental programs afforded by the self-adjusting computation framework, it was possible to give solutions to problems that previously had resisted ad-hoc algorithmic approaches [5, 24, 1, 7].

All previous work on self-adjusting computation, however, has one crucial limitation: it applies only to programs that are "purely functional" because locations cannot be written more than once.[1] Although purely functional programming is fully general (i.e., Turing complete), it is asymptotically slower than the Random-Access Model (RAM) of computation [21]. For some applications, for example those that utilize irregular data structures such as graphs, imperative programming is more natural.

The requirement of purity is inherent to the algorithmic approach: As the computation proceeds, the run-time system that supports incremental updates builds a *trace* representing the dependencies between parts of that computation and the data it manipulates. When the input is changed, a *change-propagation* algorithm traverses the trace, identifies the sub-computations affected by the change, and re-executes them. Self-adjusting computation uses memoization to enable, during change-propagation, the reuse of unchanged computations that are part of re-executed computations. The runtime techniques used to implement tracing and memoization critically depend on purity: side effects conflict with tracing by allowing programs to destroy their own trace. Memoization traditionally assumes purity. It seemed unlikely that these difficulties could be addressed.

In this paper, we show that it is possible to generalize self-adjusting computation to support imperative programming. As before, our new approach is also based on the idea of *modifiable references* (*modifiables* for short). However, while previously each modifiable had to be written exactly once during the run of a computation—which led to the modal distinction between *stable* and *changeable* computations to enforce this policy— we now allow modifiables to be written multiple times and drop the distinction between stable and changeable computations. To guarantee that all dependencies are tracked, we enforce that values that depend on the contents of modifiables are themselves communicated via modifiables.

[1] We note that the implementation of Shankar and Bodik [25], although it is given in the context of an imperative language, requires self-adjusting programs to be pure.

The resulting language, which we call AIL (Adaptive Imperative Language), is similar to an imperative language with references and provides a reasonably natural programming model. Thus, an ordinary program can be transformed into a self-adjusting program by replacing ordinary references with modifiables while restructuring the code to ensure that **read**-operations write their results into other modifiables. As an example, we show how to write a depth-first-search program on a graph and describe how this can be used to compute a topological sort. At the surface, the resulting program is algorithmically identical to the standard approach to DFS and topological sort, but due to built-in change propagation it responds to changes much faster than what would otherwise have to be done via from-scratch re-computation. In fact, the response times are similar to best bounds obtained by ad-hoc algorithmic approaches.

To support change propagation, we still trace all read and write operations. When the contents of a modifiable are changed, change propagation identifies and re-evaluates all the read operations that are affected by this change, i.e., operations that read the same modifiable between the time of the change and the time of the next update of the same location. Re-evaluation can change the contents of other modifiables, affecting other reads. This process continues until no more affected reads remain. While this looks quite similar to the way change propagation worked in the pure setting, realizing this algorithm efficiently requires a very different approach. The key difference is that when part of a trace is discarded, its write operations disappear along with it—which will affect subsequent reads.

We prove that the semantics of AIL is *consistent*, i.e., that the non-determinism in the operational semantics—due to memoization and non-deterministic allocation—is harmless, by showing that any two evaluations of the same program in the same store yield observationally (or contextually) equivalent results. In our prior work [6], we proved a similar consistency property for self-adjusting computation with only *write-once* modifiables. But, our earlier proof method fundamentally relied on the *absence of cycles* in the store. Thus, it is inapplicable in the current setting where the ability to update modifiables leads to the possibility of cyclic stores.

Reasoning about equivalence of programs in the presence of mutable state has long been recognized as a difficult problem (even for Algol-like languages [19, 26, 22]), which gets significantly harder in the presence of dynamic allocation [23, 27, 11], and harder still in the presence of cyclic stores. We know of only two recent results for proving equivalence of programs with mutable references and cyclic stores, a proof method based on bisimulation [18] and another on (denotational) logical relations [12] (neither of which is immediately applicable to proving the consistency of AIL).

In this paper, we prove equivalence of imperative self-adjusting programs using syntactic logical relations (that is, relations based on the operational semantics) where relations are indexed *not by types*, but by a natural number that, intuitively, records the number of steps available for future evaluation [10, 8]. This stratification is essential for modeling the recursive functions (available via encoding **fix**) and cyclic stores present in the language.

Another difficulty is realizing change propagation in an algorithmically efficient way. A direct implementation of the operational semantics of change propagation is not efficient because it takes time proportional to the size of the trace. Thus, in the general case, it would be no more efficient than recomputing from scratch. Instead, we represent the trace as a *persistent* graph structure. For each modifiable we keep all the *versions* that it takes over time. Each version corresponds to a write operation. For example, if a modifiable m is written the values 1 and 2 during an execution, it has two versions that corresponds to these values. We then keep track of which read operations depend on which versions. We keep the versions and the reads of each modifiable in its own *version-set* and *reader-set*, which are represented as searchable time-ordered sets. These sets support various operations such as *find*, *insert*, and *delete*, all in time logarithmic in the size of the set. We show that they can be used to support change propagation efficiently. In particular, for computations that write to each modifiable a constant number of times and read each location a constant number of times, we show that change propagation is asymptotically as efficient as in the pure case. When the number of writes is not bounded by a constant, then our approach incurs a logarithmic overhead in the maximum number of writes and reads to any modifiable.

```
signature ADAPTIVE =
sig
  type 'a mod
  type 'a eq = 'a * 'a -> bool

  val newMod: 'a eq -> 'a mod
  val mod: 'a eq -> ('a mod -> unit) -> 'a mod
  val read: 'a mod * ('a -> unit) -> unit
  val write: 'a mod -> 'a -> unit
  val hashMod:'a mod -> int
  val memo: unit -> (int list) -> (unit -> 'a) -> 'a

  val init: unit -> unit
  val deref:  'a mod -> 'a
  val change: 'a mod * 'a -> unit
  val propagate: unit -> unit
end
```

Figure 1. Signature of the library.

2. Programming with "Mutable Modifiables"

We give an overview of our framework based on our ML library and a self-adjusting version of depth-first-search trees on graphs.

The ML library. Figure 1 shows the signature of our library for self-adjusting computation with mutable modifiable references. The library provides functions to create (newMod, mod), read (read), and write (write) modifiable references, for memoizing functions (memo), and for hashing modifiables into unique integers (hashMod). We define a *self-adjusting program* as a Standard ML program that uses these functions to operate on modifiables. In addition, the library provides *meta functions* for initializing the library (init), inspecting and changing modifiable references (deref, change) and propagating changes (propagate). Meta functions cannot be used by a self-adjusting program.

A modifiable reference can be created either by the mod or newMod functions, both of which require a conservative equality test on the contents of the modifiable. A conservative comparison function returns false when the values are different but may return true or false when the values are the same. This equality function is used to stop change propagation when there is no change. The newMod function returns an uninitialized modifiable whose contents are empty, while the mod function returns a modifiable whose contents are initialized by the function provided as its second argument, which is responsible for writing to the modifiable. Both functions generate a unique integer tag for every modifiable allocated. The hashMod function returns the tag of its argument, which is used for memoizing functions. A read takes a modifiable to be read and a *reader* function. It fetches the contents of the modifiable, applies the reader to it, and returns unit. By returning unit, the reader function ensures that no ordinary (ML-) variable can depend on the contents of the modifiable read; readers can of course communicate by writing to modifiable references.

The memo function creates a memo table and returns a memoized function associated with that memo table. As usual, the memoized function takes a list of arguments and the function body,

```
datatype node =
   EMPTY
 | NODE of int * bool mod *
            NODE mod * node mod

fun depthFirstSearch f root =
let
  val mfun = memo ()
  fun dfs nr = mod (fn res => read nr (fn n =>
  case n of
    EMPTY => write res (f(NONE,NONE))
   | NODE (id,rf,rn1,rn2) => read rf (fn f =>
      mfun [id,# rf, # rn1, # rn2] (fn () =>
      if f then
      let
          val () = write rf true
          val res1 = dfs nr1
          val res2 = dfs nr2
      in
        read res1 (fn v1 => read res2 (fn v2 =>
        write res (f(SOME id, SOME(v1, v2)))))
      end))
      else
        write res NONE))
in
  dfs root
end
```

```
datatype node =
   empty
 | node of int * bool ref *
            node ref * node ref

fun depthFirstSearch f root =
let

  fun dfs n =
  case !n of
    empty => f (NONE,NONE)
   | node (id,rf,rn1,rn2) =>

      if !rf then
      let
          val () = rf := true
          val res1 = dfs rn1
          val res2 = dfs rn2
      in

        f (SOME id, SOME(res1, res2))
      end
      else
        NONE
in
    dfs root
end
```

Figure 2. The complete code for self-adjusting (left) and ordinary (right) depth-first search programs on a graph.

looks up the memo table based on the arguments, and computes and stores the result in the memo table if the result is not already found in it. To facilitate efficient memo lookups, memoized functions must hash their arguments to integers uniquely. This can be achieved by using standard boxing or tagging techniques. *Correct usage* requires that all the free-variables (or the arguments) of the memoized function body be listed as an argument to the memoized function. We do not enforce the correct-usage requirement in our implementation.

Writing Self-Adjusting Programs. Imperative self-adjusting programs use modifiable references in a way that is very similar to how ordinary programs use ordinary mutable references. Figure 2 shows the complete code for a depth-first-search (DFS) function on a graph. The graph is defined to be either empty or a node consisting of an integer identifier, a boolean flag, and two "pointers" for its neighbors. The boolean flag, and the neighbors are all placed in modifiable references.

The `depthFirstSearch` function takes a *visit function* `f` and the `root` of a graph as its arguments and performs a depth-first-search (`dfs`) starting at the root by visiting each node that has not been previously visited. The visit function can be chosen to compute various properties of graphs (e.g., topological sort, connected components). The `dfs` function visits a node by allocating a modifiable that will hold the result, and reading the node pointed to by its argument. If the node is empty, then `f` is applied with arguments that indicate that the node is empty. If the node is not empty, then the visited flag is read. If the flag is set to true, then the node was visited before and the function writes NONE to its result. If the flag is false, then the node has not been visited and the flag is first set to true, the neighboring nodes are visited recursively, and the results of the neighbors along with the identity of the visited node is passed to the visit function (`f`) to compute and write the result.

In contrast with ordinary references, the contents of modifiables are accessible only locally within the body of the reader. This requires the programmer to identify the boundaries of the code that uses the value being read. For the purpose of comparison, Figure 2 shows the code for depth-first-search with ordinary references. As can be observed, the two programs are structurally very similar. In fact, the imperative program can be transformed into the self-adjusting program by simply replacing each reference with a modifiable while inserting reads and writes appropriately.

```
structure Graph =
struct
  fun fromFile s = ...
  fun node i = ...
  fun neigbor (i,j) = ...
  fun topologicalSort = ...
end

fun test (s,i,j) =
let
  val _ = init ()
  val (root,graph,n) = fromFile s
  val r = depthFirstSearch topologicalSort (root)

  val nr = Graph.neighbor (i,0)
  val () = change nr (Graph.node j)
  val () = propagate ();
in
  r
end
```

Figure 3. Example of changing input and change propagation.

The `depthFirstSearch` function makes it possible to compute different properties of graphs by passing different visit functions. One common application of DFS is topological sort. We can compute the topological sort of a graph using a visitor that concatenates its argument lists (if any), and then returns the result with the node being visited inserted at the head. This visitor follows the standard description [14]. As an example, consider the graphs in Figure 4. Each node is annotated with two time stamps: the first and the last time they were visited. For node `A` these are 1 and 16, respectively. The topological sort of a graph can be determined by sorting the nodes according to their last-visit time. (This is effectively what our visitor does.) In Figure 4, the left graph is sorted as `A,B,C,D,E,F,G,H` and the right graph is sorted as `A,B,F,G,H,C,D,E`.

Propagation. A self-adjusting computation lets programmers change input values "after the fact" and updates the result accordingly. This process can be repeated. Figure 3 shows an example that uses `depthFirstSearch` to perform a topological sort. For

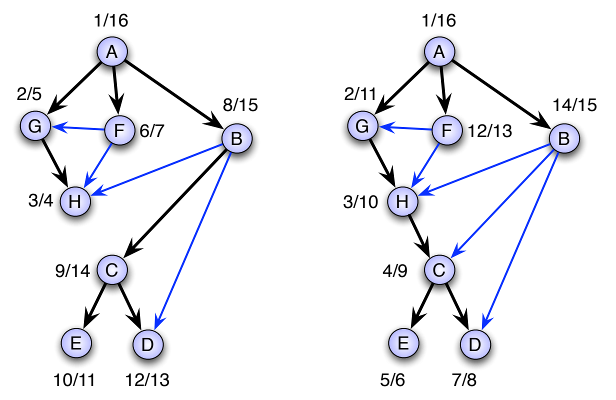

Figure 4. A DAG before and after insertion of the edge (H,C).

brevity, we omit the details of a graph library that supports various graph operations such as constructing graphs from a file, locating a node or finding a neighbor.

The `test` function first constructs a graph from a file and then computes its topological sort using `depthFirstSearch`. The `topologicalSort` function, whose code we omit, is a typical visitor that can be used with `depthFirstSearch` as described above.

After the initial run is complete, the `test` function inserts an edge from node i to node j as specified by its arguments. To insert the new edge, the function first finds the modifiable that points to the node i and then changes it to point to node j. The function then initiates a change-propagation to update the result. This change-and-propagate step can be repeated as many times as desired, with different nodes and edges.

Performance. We show that the self-adjusting version of the standard DFS algorithm responds to changes efficiently. For the proof, we introduce some terminology. Let G be an ordered graph, i.e., a graph where the out-edges are totally ordered. Consider performing a DFS on G such that the out-edges of each node are visited in the order specified by their total order. Let T be the DFS-tree of the traversal, i.e., the tree that consists of the edges (u, v) whose destinations v are not visited during the time that the edge is traversed. Consider now a graph G' that is obtained from G by inserting/deleting an edge. Consider performing DFS on G' and let T' be its DFS-tree. We define the *affected nodes* as the nodes of T (or G) whose paths to the root are different in T and T'. Figure 4 shows two example graphs G and G', where G' is obtained from G by inserting the edge `(H,C)`. The DFS-trees of these graphs consist of edges that are drawn in bold. The affected nodes are `C`,`D`, and `E`, because these are the only nodes that are accessible through the newly inserted edge `(H,C)` from the root `A`.

Based on these definitions, we prove that DFS takes time proportional to the number of affected nodes. Since the total time will depend on the visitor (`f`) that determines the result of the DFS, we first show a bound disregarding visitor computations. We then consider a particular instantiation of the visitor for performing topological sort and show that the same bound holds for this application as well. For the proofs, which will be given in Section 5.4 after the change-propagation algorithm has been described, we assume that each node has constant out-degree.

Theorem 2.1 (DFS Response Time). *Disregarding read operations performed for the purposes of computing the results, the* `depthFirstSearch` *program responds to changes in time $O(m)$, where m is the number of affected nodes after an insertion/deletion.*

Our bound for topological sort is the same as that for DFS, i.e., we only pay for those nodes that are affected.

Values	v	::=	$()$ | n | x | l | $\lambda x.\,e$ | (v_1, v_2) | $\texttt{inl}\ v$ | $\texttt{inr}\ v$
Prim Ops	o	::=	$+$ | $-$ | $=$ | $<$ | $\ldots$
Exprs	e	::=	v | $o\,(v_1, \ldots, v_n)$ | $v_1\ v_2$ |
			$\texttt{mod}\ v$ | $\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e$ | $\texttt{write}\ v_1 \leftarrow v_2$ |
			$\texttt{memo}\ e$ | $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ | $\texttt{fst}\ v$ | $\texttt{snd}\ v$ |
			$\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow e_1 \mid \texttt{inr}\ x_2 \Rightarrow e_2$

Figure 5. Syntax

$$T \ ::= \ \varepsilon \mid \texttt{let}\ T_1\ T_2 \mid \texttt{mod}\ l \leftarrow v \mid \texttt{read}_{\,l \to x = v.e}\ T \mid \texttt{write}\ l \leftarrow v$$

Figure 6. Traces

Theorem 2.2 (Topological Sort). *Change propagation updates the topological sort of a graph in $O(m)$ time where m is the number of affected nodes.*

Interestingly, the self-adjusting DFS program exhibits algorithmic and asymptotic complexity behavior similar to previously proposed algorithms for topological sorting of incrementally changing graphs, for example that by Pearce and Kelly [20]. The main difference compared to their work is, of course, that we obtained our code by simply annotating a standard, non-adjusting version of DFS.

3. The Language

Since our consistency proof does not depend on type safety, we leave our language untyped. For simplicity, we assume all expressions to be in A-normal form [17]. Unlike in our previous work where it was necessary to enforce a write-once policy for modifiable references, we do not distinguish between stable and changeable computations. This simplifies the syntax of AIL considerably. Modifiable references now behave much like ordinary ML-style references: they are initialized at creation time and can be updated arbitrarily often by the program.

3.1 Values and expressions

The syntax of the language is shown in Figure 5. Value forms v include unit $()$, variables x, integers n, locations l, λ-abstractions $\lambda x.\,e$, pairs of values (v_1, v_2), and injections $\texttt{inl}\ v$ and $\texttt{inr}\ v$ into a sum.

Expressions e that are not themselves values v can be applications of primitive operations $o\,(v_1, \ldots, v_n)$ (where o is something like $+$, $-$, or $<$), function applications $v_1\ v_2$, allocation and initialization of modifiable references ($\texttt{mod}\ v$), scoped *read*-operations $\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e$ that bind the value stored at the location given by v to variable x and execute e in the scope of x, *write*-operations $\texttt{write}\ v_1 \leftarrow v_2$ that store v_2 into the location given by v_1, *let*-bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, projections from pairs ($\texttt{fst}\ v$ and $\texttt{snd}\ v$) and case analysis on sums ($\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow e_1 \mid \texttt{inr}\ x_2 \Rightarrow e_2$). The form $\texttt{memo}\ e$ marks an expression that is subject to memoization: evaluation of e may take advantage of an earlier evaluation of the same expression, possibly using change propagation to account for changes to the store.

3.2 Traces

Change propagation requires access to the "history" of an evaluation. A history is represented by a *trace*, and every evaluation judgment specifies an *output trace*. Traces (see Figure 6) can be empty (ε), combine two sub-traces obtained by evaluating the two sub-terms of a *let*-form ($\texttt{let}\ T_1\ T_2$), or record the allocation of a new modifiable reference l that was initialized to v ($\texttt{mod}\ l \leftarrow v$). A trace of the form $\texttt{read}_{\,l \to x = v.e}\ T$ indicates that reading l produced a value v that was bound to x for the evaluation of e which produced the sub-trace T. Finally, the trace $\texttt{write}\ l \leftarrow v$ records that an existing location l's contents have been updated to con-

$$\frac{}{\sigma, v \Downarrow^0 v, \sigma, \varepsilon}\ \textbf{(value)} \qquad \frac{v = \mathsf{app}(o,\ (v_1, \ldots, v_n))}{\sigma, o\,(v_1, \ldots, v_n) \Downarrow^1 v, \sigma, \varepsilon}\ \textbf{(primop)}$$

$$\frac{v_1 = \lambda x.\, e \qquad \sigma, e[v_2/x] \Downarrow^k v_3, \sigma', T_1}{\sigma, v_1\, v_2 \Downarrow^{k+1} v_3, \sigma', T_1}\ \textbf{(apply)}$$

$$\frac{\begin{array}{c}\sigma, e_1 \Downarrow^{k_1} v_1, \sigma_1, T_1 \\ \sigma_1, e_2[v_1/x] \Downarrow^{k_2} v_2, \sigma_2, T_2 \qquad \mathsf{alloc}(T_1) \cap \mathsf{alloc}(T_2) = \emptyset\end{array}}{\sigma, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow^{k_1+k_2+1} v_2, \sigma_2, \mathtt{let}\ T_1\ T_2}\ \textbf{(let)}$$

$$\frac{}{\sigma, \mathtt{mod}\ v \Downarrow^1 l, \sigma[l \leftarrow v], \mathtt{mod}\ l \leftarrow v}\ \textbf{(mod)}$$

$$\frac{\sigma, e[\sigma(l)/x] \Downarrow^k (), \sigma', T}{\sigma, \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e \Downarrow^{k+1} (), \sigma', \mathtt{read}_{\,l \to x = \sigma(l).e}\ T}\ \textbf{(read)}$$

$$\frac{}{\sigma, \mathtt{write}\ l \leftarrow v \Downarrow^1 (), \sigma[l \leftarrow v], \mathtt{write}\ l \leftarrow v}\ \textbf{(write)}$$

$$\frac{\sigma_0, e \Downarrow^{k_0} v, \sigma_0', T_0 \qquad \sigma, T_0 \curvearrowright^k \sigma', T}{\sigma, \mathtt{memo}\ e \Downarrow^{k_0+k} v, \sigma', T}\ \textbf{(memo)}$$

$$\frac{}{\sigma, \mathtt{fst}\ (v_1, v_2) \Downarrow^1 v_1, \sigma, \varepsilon}\ \textbf{(fst)} \qquad \frac{}{\sigma, \mathtt{snd}\ (v_1, v_2) \Downarrow^1 v_2, \sigma, \varepsilon}\ \textbf{(snd)}$$

$$\frac{\sigma, e_1[v/x_1] \Downarrow^k v', \sigma', T}{\sigma, \mathtt{case}\ \mathtt{inl}\ v\ \mathtt{of}\ \mathtt{inl}\ x_1 \Rightarrow e_1 \mid \mathtt{inr}\ x_2 \Rightarrow e_2 \Downarrow^{k+1} v', \sigma', T}\ \textbf{(case/inl)}$$

$$\frac{\sigma, e_2[v/x_2] \Downarrow^k v', \sigma', T}{\sigma, \mathtt{case}\ \mathtt{inr}\ v\ \mathtt{of}\ \mathtt{inl}\ x_1 \Rightarrow e_1 \mid \mathtt{inr}\ x_2 \Rightarrow e_2 \Downarrow^{k+1} v', \sigma', T}\ \textbf{(case/inr)}$$

Figure 7. Evaluation Rules

tain the new value v. The only difference between `write` $l \leftarrow v$ and `mod` $l \leftarrow v$ is that the former is not counted as an allocation: $\mathsf{alloc}(\mathtt{write}\ l \leftarrow v) = \emptyset$ while $\mathsf{alloc}(\mathtt{mod}\ l \leftarrow v) = \{l\}$. In general, $\mathsf{alloc}(T)$ denotes the set of locations that appear in `mod` $l \leftarrow v$ within the trace T (formal definition elided).

3.3 Stores

As we have explained earlier, the actual implementation of the store maintains multiple time-stamped versions of the contents of each cell. In our formal semantics, the version-tracking store is present implicitly in the trace: To look up the current version of the contents of l at a given point of the execution, one can simply walk the global trace backwards up to the most recent write operation on l.

Formalizing this idea, while possible, would require the semantics to pass around a representation of a *global* trace representing execution from the very beginning up to the current program point. Moreover, such a trace would need more internal structure to be able to support change propagation.

The alternative formalization that we use here takes advantage of the following observation: At any given point in time we only need the *current view* of the global version-keeping store. We refer to this view as "the store" because it simply maps locations to values. Thus, instead of representing the version store explicitly, our semantics keeps track of the changes to the *current view* of the version store by manipulating ordinary stores.

3.4 Operational Semantics

The operational semantics consists of rules for deriving *evaluation judgments* of the form $\sigma, e \Downarrow^k v, \sigma', T$, which should be read as: "In store σ, expression e evaluates in k steps to value v, resulting in store σ'. The computation is described by trace T." Step counts are irrelevant to the evaluation itself, but we will use them later in our logical relation for reasoning about program equivalence. The rules for deriving evaluation judgments are shown in Figure 7.

$$\frac{}{\sigma, \varepsilon \curvearrowright^0 \sigma, \varepsilon}\ \textbf{(empty)}$$

$$\frac{}{\sigma, \mathtt{mod}\ l \leftarrow v \curvearrowright^0 \sigma[l \leftarrow v], \mathtt{mod}\ l \leftarrow v}\ \textbf{(mod)}$$

$$\frac{}{\sigma, \mathtt{write}\ l \leftarrow v \curvearrowright^0 \sigma[l \leftarrow v], \mathtt{write}\ l \leftarrow v}\ \textbf{(write)}$$

$$\frac{\begin{array}{c}\sigma, T_1 \curvearrowright^{k_1} \sigma', T_1' \\ \sigma', T_2 \curvearrowright^{k_2} \sigma'', T_2' \qquad \mathsf{alloc}(T_1') \cap \mathsf{alloc}(T_2') = \emptyset\end{array}}{\sigma, \mathtt{let}\ T_1\ T_2 \curvearrowright^{k_1+k_2} \sigma'', \mathtt{let}\ T_1'\ T_2'}\ \textbf{(let)}$$

$$\frac{\sigma(l) = v \qquad \sigma, T \curvearrowright^k \sigma', T'}{\sigma, \mathtt{read}_{\,l \to x = v.e}\ T \curvearrowright^k \sigma', \mathtt{read}_{\,l \to x = v.e}\ T'}\ \textbf{(read/no ch.)}$$

$$\frac{\sigma(l) \neq v \qquad \sigma, e[\sigma(l)/x] \Downarrow^k (), \sigma', T'}{\sigma, \mathtt{read}_{\,l \to x = v.e}\ T \curvearrowright^{k+1} \sigma', \mathtt{read}_{\,l \to x = \sigma(l).e}\ T'}\ \textbf{(read/ch.)}$$

Figure 8. Change Propagation Rules

The rule for `memo` has a premise of the form $\sigma, T \curvearrowright^k \sigma', T'$. This is a *change propagation judgment* and should be read as: "The computation described by T is adjusted in k steps to a new computation described by T' and a corresponding new store σ'." The rules for deriving change propagation judgments are shown in Figure 8. Memoization is modeled by starting at some "previous" evaluation of e (in some other store σ_0) that is now adjusted to the current store σ.

Evaluation rules. Values and primitive operations, which are considered pure, add nothing to the trace (rules **value** and **primop**). Application evaluates the body of the function after substituting the argument for the formal parameter. The resulting trace is the one produced while evaluating the body (rule **apply**). A `let`-expression is evaluated by running the two sub-terms in sequence, substituting the result of the first for the bound variable in the second. The trace is the concatenation (using the `let`-constructor for traces) of the two sub-traces (rule **let**). Evaluating `mod` v picks a location l, stores v at l, and returns l. This action (including l and v) is recorded in the trace (rule **mod**). A `read`-expression substitutes the value stored at location to be read for the bound variable in the body. Evaluating the resulting body must return the unit value $()$. The `read`-operation—including location, bound variable, value read, and body—is recorded in the trace (rule **read**). `Write`-operations modify the store by associating the location to be written with the new value. The result of a `write` is unit. Both value and location are recorded in the trace (rule **write**). Evaluating a `memo`-expression uses an evaluation of its body in some arbitrary store σ_0—provided that evaluation can be adjusted to the current store via change propagation (rule **memo**). The rules **fst** and **snd** are the standard projection rules. Projections are pure and, therefore, add nothing to the trace. Similarly, rules **case/inl** and **case/inr** are the standard elimination rules for sums. In each case, the trace records whatever happened in the branch that was taken. The case analysis itself is pure and does not need to be recorded.

Change-propagation rules. Rule **empty** is the base case and deals with the empty trace. The **mod**- and **write**-rules re-executes their respective write operations to the store. There are two possible scenarios that make this necessary: (1) there may have been a `write` operation that altered the contents of the location after it

$$\begin{array}{l} \sigma : \eta \rightsquigarrow \mathcal{L} \stackrel{\text{def}}{=} \\ \qquad \mathcal{L} = \eta \cup \bigcup^{l \in \mathcal{L}} FL(\sigma(l)) \;\wedge\; \mathsf{dom}(\sigma) \supseteq \mathcal{L} \;\wedge \\ \qquad \forall \mathcal{L}^{\dagger} \subseteq \mathcal{L}. \\ \qquad\qquad \eta \subseteq \mathcal{L}^{\dagger} \;\wedge\; (\forall l \in \mathcal{L}^{\dagger}.\, FL(\sigma(l)) \subseteq \mathcal{L}^{\dagger}) \implies \mathcal{L} = \mathcal{L}^{\dagger} \end{array}$$

Figure 9. Store Reachability Relation

was originally created or written, or (2) in the rule for memo, the "original" store σ_0 was so different from σ that the location in question has a different value (or does not even exist) in σ. The **let**-rule simply performs change propagation on each of the sub-traces. The remaining two rules are those for read—one for the case that there is no change, the other for the case that there is a change. When the value at the location being read is still the same as the one recorded in the trace, then change propagation simply presses on (rule **read/noch**.). If the value is not the same, then the old sub-trace is thrown away and the body of the read is re-evaluated with the new value substituted for the bound variable (rule **read/ch**.).

Notice that as long as there is no change, the rules for change-propagation do not increment the step count, because unchanged computations have already been accounted for by **memo**.

Discussion. Our rules are given in a non-deterministic, declarative style. For example, the **mod**-rule does not place any special requirements on the location being allocated, i.e., the location could already exist in the store. For correctness, however, we insist that all locations allocated during the course of the entire program run be pairwise distinct. (This is enforced by side conditions on our **let**-rules.) Furthermore, allocated locations must not be reachable from the initial expression (see Section 3.5).

As in our previous work [6], the ability for mod to allocate an existing (garbage-) location during change propagation is crucial, since otherwise change propagation would not be able to retain any previous allocations. The ability to allocate an existing garbage location during ordinary evaluation is not as important, but disallowing the possibility (e.g., by adding a side-condition of $l \notin \mathsf{dom}(\sigma)$ to the premise of the evaluation rule for mod) would have two undesirable effects: It would weaken our result by reducing the number of possible evaluations, and it would make our formal framework for reasoning about program equivalence more complicated.

Some readers might notice that we no longer distinguish between memo "hits" and memo "misses" the way we did before [6]. The high degree of freedom in the choice of σ_0 makes memo misses special cases of memo hits: In the **memo**-rule, to simulate a memo miss we pick $\sigma_0 = \sigma$. Since change propagation starting from σ does nothing, this captures the essence of the memo miss: evaluation proceeds directly in the current store σ, not some other "previous" store σ_0.

Evaluating a read-form returns the unit value. Therefore, the only way for the body of the read-form to communicate to the rest of the program is by writing into other modifiable references, or even possibly the same reference that it read. This convention guarantees stability of values and justifies the rule for memo where we return the value computed during an arbitrary "earlier" evaluation in some other store σ_0. The value so computed cannot actually depend on the contents of σ_0. It can, of course, be a location pointing to values that do depend on σ_0, but those will be adjusted during change propagation.

3.5 Reachability and Valid Evaluations

Consistency holds only for so-called *valid evaluations*. Informally, an evaluation is valid if it does not allocate locations *reachable* from the initial expression e. Our technique for identifying the locations reachable from an expression is based on the technique used by Ahmed et al. [9] in their work on substructural state. Let $FL(e)$ be the *free locations* of e, i.e., those locations that are subexpressions of e. The locations $FL(e)$ are said to be *directly accessible* from e. The store reachability relation $\sigma : \eta \rightsquigarrow \mathcal{L}$ (Figure 9) allows us to identify the set of locations $\mathcal{L}$ reachable in a store σ from a set of "root" locations η. The relation $\sigma : \eta \rightsquigarrow \mathcal{L}$ requires that the reachable set $\mathcal{L}$ include the root locations η as well as all locations directly accessible from each $l \in \mathcal{L}$. It also ensures that all reachable locations are in σ. Furthermore, it requires that $\mathcal{L}$ be *minimal* — that is, it ensures that the set $\mathcal{L}$ does not contain any locations not reachable from the roots.

Thus, $\mathcal{L}$ is the set of locations reachable from an expression e in a store σ iff $\sigma : FL(e) \rightsquigarrow \mathcal{L}$. We define valid evaluations $\sigma, e \Downarrow^k_{\text{ok}} v, \sigma', T$ as follows.

Definition 3.1 (Valid Evaluation).

$$\begin{array}{lcl} \sigma, e \Downarrow^k_{\text{ok}} v, \sigma', T & \stackrel{\text{def}}{=} & \sigma, e \Downarrow^k v, \sigma', T \;\wedge \\ & & \exists \mathcal{L}.\ \sigma : FL(e) \rightsquigarrow \mathcal{L} \;\wedge\; \mathcal{L} \cap \mathsf{alloc}(T) = \emptyset \end{array}$$

3.6 Contextual Equivalence

A context C is an expression with a hole in it. We write $C : (\Gamma)$ to denote that C is a closed context (i.e. $FV(C) = \emptyset$) that provides bindings for variables in the set Γ. Thus, if $FV(e) \subseteq \Gamma$, then $C[e]$ is a closed term. We write $\sigma : \eta$ as shorthand for: $\exists \mathcal{L}.\ \sigma : \eta \rightsquigarrow \mathcal{L}$. We say e_1 contextually approximates e_2 if, given an arbitrary C that provides bindings for the free variables of both terms, running $C[e_1]$ in a store σ (that contains all the appropriate roots) returns n, then (1) there exists an evaluation for $C[e_2]$ in σ, and (2) all such evaluations also return n.

Definition 3.2 (Contextual Equivalence).
Let $\Gamma = FV(e_1) \cup FV(e_2)$.

$$\begin{array}{lcl} \Gamma \vdash e_1 \prec^{ctx} e_2 & \stackrel{\text{def}}{=} & \forall C : (\Gamma).\ \forall \sigma, \eta, n. \\ & & \eta = FL(C) \cup FL(e_1) \cup FL(e_2) \;\wedge\; \sigma : \eta \;\wedge \\ & & \sigma, C[e_1] \Downarrow_{\text{ok}} n, -, - \implies \\ & & (\,\exists v.\ \sigma, C[e_2] \Downarrow_{\text{ok}} v, -, -\,) \;\wedge \\ & & (\,\forall v.\ \sigma, C[e_2] \Downarrow_{\text{ok}} v, -, - \implies n = v\,) \end{array}$$

$$\Gamma \vdash e_1 \approx^{ctx} e_2 \stackrel{\text{def}}{=} \Gamma \vdash e_1 \prec^{ctx} e_2 \;\wedge\; \Gamma \vdash e_2 \prec^{ctx} e_1$$

4. A Step-Indexed Logical Relation for State

In this section, we prove that the semantics of AIL is *consistent* — i.e., that the non-determinism in the operational semantics is harmless — by showing that any two *valid* evaluations of the same program in the same store yield observationally (contextually) equivalent results. More formally, if e is a closed program, we wish to show that $\emptyset \vdash e \approx^{ctx} e$, which means that if we run $C[e]$ (where C is an arbitrary context) twice in the same store σ, then we get the same result value n.

It is difficult to prove $\emptyset \vdash e \approx^{ctx} e$ directly due to the quantification over *all* contexts in the definition of $\approx^{ctx}$. Instead we use the standard approach of using a logical relation in order to prove contextual equivalence — that is, we will show that any term e is logically related to itself (Theorem 4.4), and that the latter implies that e is contextually equivalent to itself (Theorem 4.5).

The two sources of non-determinism are allocation and memoization. Since they differ in nature, we deal with them using different techniques. The non-determinism caused by allocation only concerns the identity of locations. We handle this by maintaining a bijection between the locations allocated in different runs of the same program. When both runs execute a mod v, we extend the bijection with the pair of locations (l_1, l_2) returned by mod v. Notice that it will always be possible to prove that the result is a bijection because valid evaluations cannot reuse reachable locations.

If we start with identical programs (modulo the location bijection), then they will execute in lock-step until they encounter a `memo`, at which point the derivation trees for the evaluation judgments can differ dramatically. There is no way of relating the two executions directly. Fortunately, this is not necessary, since they need to be related only after change-propagation has brought them back into sync. A key insight is that at such sync points it is always possible to establish a bijection between those locations that are *reachable* from each of the two running programs.

Our relational model of AIL is based on the step-indexed logical relations for purely functional languages by Appel and McAllester [10] and Ahmed [8]. In those models, the relational interpretation $\mathcal{V}[\![\tau]\!]$ of a (closed) type τ is a set of triples of the form (k, v_1, v_2) where k is a natural number (called the *approximation index* or *step index*) and v_1 and v_2 are closed values. Intuitively, $(k, v_1, v_2) \in \mathcal{V}[\![\tau]\!]$ says that in any computation running for no more than k steps, v_1 approximates (or "looks like") v_2. Informally, we say that v_1 and v_2 are related for k steps.

A novel aspect of the logical relation that we present below is that it is untyped — that is, it is indexed only by step counts, unlike logical relations in the literature which are always indexed by types (or in the case of prior step-indexed logical relations [10, 8], by both types and step counts).

Another novelty is the way in which our model tracks relatedness of the stores of the two computations. The intuition is to start at the variables of each program that point into the respective stores (i.e., the roots of a tracing garbage collector), and construct graphs of the reachable memory cells by following pointers. Then the two program stores are related for k steps if these graphs are isomorphic, and if the contents of related locations (i.e., bijectively related vertices of the graphs) are related for $k-1$ steps. (Since reading a location consumes a step, $k-1$ suffices here.)

4.1 Preliminaries

We use the meta-variable $\mathcal{S}$ to denote sets of location pairs. We define the following abbreviations:

$$\mathcal{S}^1 \equiv \{\, l_1 \mid (l_1, l_2) \in \mathcal{S} \,\} \qquad \mathcal{S}^2 \equiv \{\, l_2 \mid (l_1, l_2) \in \mathcal{S} \,\}$$

We define the set of location bijections as follows:

$$\begin{array}{lcl} \mathit{bij}(\mathcal{S}) & \overset{\text{def}}{=} & \forall l \in \mathcal{S}^1.\ \exists! l_2 \in \mathcal{S}^2.\ (l_1, l_2) \in \mathcal{S}\ \wedge \\ & & \forall l \in \mathcal{S}^2.\ \exists! l_1 \in \mathcal{S}^1.\ (l_1, l_2) \in \mathcal{S} \\ \\ \mathit{LocBij} & = & \{\, \mathcal{S} \in 2^{\mathit{Locs} \times \mathit{Locs}} \mid \mathit{bij}(\mathcal{S}) \,\} \end{array}$$

4.2 Related Values

The value relation $\mathcal{V}$ specifies when two values are related. $\mathcal{V}$ is a set of tuples of the form (k, ψ, v_1, v_2), where k is the step index, v_1 and v_2 are closed values, and $\psi \in \mathit{LocBij}$ is a *local store description*. A set of "beliefs" ψ is a bijection on the locations directly accessible from v_1 and v_2. We refer to the locations in ψ^1 and ψ^2 as the *roots* of v_1 and v_2, respectively.

The definition of the value relation $\mathcal{V}$ is given in Figure 10. The value $(\,)$ is related to itself for any number of steps. Clearly, no locations appear as sub-expressions of $(\,)$; hence, the definition demands an empty local store description $\{\}$. Similarly, integers n_1 and n_2 are related under the empty store description if they are equal.

Two locations l_1 and l_2 are related if the local store description says that they are related. Furthermore, from the values l_1 and l_2, the only locations that are directly accessible are, respectively, the locations l_1 and l_2 themselves. Hence, the local store description must be $\{(l_1, l_2)\}$.

The pairs (v_1, v_1') and (v_2, v_2') are related for k steps if there exist local store descriptions ψ and ψ' such that the components of the pairs are related (i.e., $(k, \psi, v_1, v_2) \in \mathcal{V}$ and $(k, \psi', v_1', v_2') \in \mathcal{V}$) and if ψ and ψ' can be combined into a single set of beliefs (written $\psi \odot \psi'$, see Figure 11). Informally, two local store descriptions ψ and ψ' can be combined only if they are compatible; that is, if the beliefs in ψ do not contradict the beliefs in ψ', or more precisely, if the union of the two bijections is also a bijection.

$$\begin{array}{lcl} \mathcal{V} & = & \{\, (k,\, \{\},\, (\,),\, (\,)) \,\} \;\cup \\ & & \{\, (k,\, \{\},\, n,\, n) \,\} \;\cup \\ & & \{\, (k,\, \{(l_1,l_2)\},\, l_1,\, l_2) \,\} \;\cup \\ & & \{\, (k,\, \psi_c,\, \lambda x.\, e_1,\, \lambda x.\, e_2) \mid \\ & & \quad \forall j < k.\ \forall \psi_a, v_1, v_2. \\ & & \qquad (j,\, \psi_a,\, v_1,\, v_2) \in \mathcal{V}\ \wedge\ (\psi_c \odot \psi_a) \text{ defined } \implies \\ & & \qquad (j,\, \psi_c \odot \psi_a,\, e_1[v_1/x],\, e_2[v_2/x]) \in \mathcal{C} \,\} \;\cup \\ & & \{\, (k,\, \psi \odot \psi',\, (v_1, v_1'),\, (v_2, v_2')) \mid \\ & & \quad (k,\, \psi,\, v_1,\, v_2) \in \mathcal{V}\ \wedge\ (k,\, \psi',\, v_1',\, v_2') \in \mathcal{V} \,\} \;\cup \\ & & \{\, (k,\, \psi,\, \mathtt{inl}\, v_1,\, \mathtt{inl}\, v_2) \mid (k,\, \psi,\, v_1,\, v_2) \in \mathcal{V} \,\} \;\cup \\ & & \{\, (k,\, \psi,\, \mathtt{inr}\, v_1,\, \mathtt{inr}\, v_2) \mid (k,\, \psi,\, v_1,\, v_2) \in \mathcal{V} \,\} \end{array}$$

$$\begin{array}{lcl} \mathcal{C} & = & \{\, (k,\, \psi_s,\, e_1,\, e_2) \mid \\ & & \quad \forall j < k.\ \forall \sigma_1, \sigma_2, \psi_r, \mathcal{S}, v_1, \sigma_1', T_1. \\ & & \qquad \sigma_1, \sigma_2 :_k (\psi_s \odot \psi_r) \rightsquigarrow \mathcal{S}\ \wedge \\ & & \qquad \sigma_1, e_1 \Downarrow^j\ v_1, \sigma_1', T_1\ \wedge \\ & & \qquad \mathcal{S}^1 \cap \mathsf{alloc}(T_1) = \emptyset \implies \\ & & \qquad (\,\exists v_2, \sigma_2', T_2. \\ & & \qquad\quad \sigma_2, e_2 \Downarrow v_2, \sigma_2', T_2\ \wedge \\ & & \qquad\quad \mathcal{S}^2 \cap \mathsf{alloc}(T_2) = \emptyset\,)\ \wedge \\ & & \qquad (\,\forall v_2, \sigma_2', T_2. \\ & & \qquad\quad \sigma_2, e_2 \Downarrow v_2, \sigma_2', T_2\ \wedge \\ & & \qquad\quad \mathcal{S}^2 \cap \mathsf{alloc}(T_2) = \emptyset \implies \\ & & \qquad\quad \exists \psi_f, \mathcal{S}_f.\ (k-j,\, \psi_f,\, v_1,\, v_2) \in \mathcal{V}\ \wedge \\ & & \qquad\qquad \sigma_1', \sigma_2' :_{k-j} (\psi_f \odot \psi_r) \rightsquigarrow \mathcal{S}_f\ \wedge \\ & & \qquad\qquad \mathcal{S}_f^1 \subseteq \mathcal{S}^1 \cup \mathsf{alloc}(T_1)\ \wedge \\ & & \qquad\qquad \mathcal{S}_f^2 \subseteq \mathcal{S}^2 \cup \mathsf{alloc}(T_2)\,) \end{array}$$

$$\begin{array}{rcl} \mathcal{G}[\![\emptyset]\!] & = & \{\, (k,\, \{\},\, \emptyset,\, \emptyset) \,\} \\ \mathcal{G}[\![\Gamma, x]\!] & = & \{\, (k,\, \psi_\Gamma \odot \psi_x,\, \gamma_1[x \mapsto v_1],\, \gamma_2[x \mapsto v_2]) \mid \\ & & \quad (k,\, \psi_\Gamma,\, \gamma_1,\, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]\ \wedge \\ & & \quad (k,\, \psi_x,\, v_1,\, v_2) \in \mathcal{V} \,\} \end{array}$$

$$\begin{array}{rcl} \Gamma \vdash e_1 \preccurlyeq e_2 & \overset{\text{def}}{=} & \forall k \geq 0.\ \forall \psi_\Gamma, \gamma_1, \gamma_2. \\ & & \quad (k,\, \psi_\Gamma,\, \gamma_1,\, \gamma_2) \in \mathcal{V}_\Gamma \implies \\ & & \quad (k,\, \psi_\Gamma,\, \gamma_1(e_1),\, \gamma_2(e_2)) \in \mathcal{C} \end{array}$$

$$\begin{array}{rcl} \Gamma \vdash e_1 \approx e_2 & \overset{\text{def}}{=} & \Gamma \vdash e_1 \preccurlyeq e_2\ \wedge\ \Gamma \vdash e_2 \preccurlyeq e_1 \\ & & \text{(where } \Gamma = FV(e_1) \cup FV(e_2)\text{)} \end{array}$$

Figure 10. Logical Relation

$$\psi_1 \odot \psi_2 \quad \overset{\text{def}}{=} \quad \begin{cases} \psi_1 \cup \psi_2 & \text{if } (\psi_1 \cup \psi_2) \in \mathit{LocBij} \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

Figure 11. Join Partial Function

The left (right) injections into a sum $\mathtt{inl}\, v_1$ and $\mathtt{inl}\, v_2$ ($\mathtt{inr}\, v_1$ and $\mathtt{inr}\, v_2$) with local store description ψ are related for k steps if v_1 and v_2 are related for k steps with the same local store description (i.e., $(k, \psi, v_1, v_2) \in \mathcal{V}$).

Since functions are suspended computations, their relatedness is defined in terms of the relatedness of computations (see below). Two functions $\lambda x.\, e_1$ and $\lambda x.\, e_2$ with local store description ψ_c — where ψ_c describes *at least* the sets of locations directly accessible from the closures of the respective functions — are related for k steps if, at some point in the future, when there are $j < k$ steps left to execute, and there are related arguments v_1 and v_2 such that $(j, \psi_a, v_1, v_2) \in \mathcal{V}$, and the beliefs ψ_c and ψ_a are compatible, then $e_1[v_1/x]$ and $e_2[v_2/x]$ are related as computations for j steps. Note that j must be *strictly smaller* than k. The latter requirement is essential for ensuring that the logical relation is well-founded

$$\begin{array}{lcl} \sigma_1, \sigma_2 :_k \psi \rightsquigarrow \mathcal{S} & \overset{\text{def}}{=} & \mathcal{S} \in \mathit{LocBij} \;\wedge \\ & & \exists \mathcal{F}_\psi : \mathcal{S} \to \mathit{LocBij}. \\ & & \quad \mathcal{S} = \psi \odot \bigodot^{(l_1,l_2)\in\mathcal{S}} \mathcal{F}_\psi(l_1,l_2) \;\wedge \\ & & \quad \mathsf{dom}(\sigma_1) \supseteq \mathcal{S}^1 \wedge \mathsf{dom}(\sigma_2) \supseteq \mathcal{S}^2 \;\wedge \\ & & \quad \forall (l_1,l_2) \in \mathcal{S}.\ \forall j < k. \\ & & \qquad (j,\, \mathcal{F}_\psi(l_1,l_2),\, \sigma_1(l_1),\, \sigma_2(l_2)) \in \mathcal{V} \end{array}$$

Figure 12. Related Stores

(despite the fact that it is *not* indexed by types). Intuitively, $j < k$ suffices because beta-reduction consumes a step.

A crucial property of the relation $\mathcal{V}$ is that it is closed under decreasing step index.

Lemma 4.1 (Downward Closed).
If $(k, \psi, v_1, v_2) \in \mathcal{V}$ *and* $j \leq k$, *then* $(j, \psi, v_1, v_2) \in \mathcal{V}$.

4.3 Related Stores

The store satisfaction relation $\sigma_1, \sigma_2 :_k \psi \rightsquigarrow \mathcal{S}$ (see Figure 12) says that the stores σ_1 and σ_2 are related (to approximation k) at the local store description ψ and the "global" store description $\mathcal{S}$ (where $\mathcal{S} \in \mathit{LocBij}$). We motivate the definition of $\sigma_1, \sigma_2 :_k \psi \rightsquigarrow \mathcal{S}$ by analogy with a tracing garbage collector. Here ψ correspond to (beliefs about) the portions of the stores directly accessible from a pair of values (or multiple pairs of values, when ψ corresponds to $\odot$-ed store descriptions). Hence, informally ψ corresponds to the (two sets of) root locations. Meanwhile, $\mathcal{S}$ corresponds to the set of reachable (root and non-root) locations in the two stores that would be discovered by the garbage collector.[2] In the definition of $\sigma_1, \sigma_2 :_k \psi \rightsquigarrow \mathcal{S}$, the function $\mathcal{F}_\psi$ maps each location pair $(l_1, l_2) \in \mathcal{S}$ to a local store description. It is our intention that, for each pair of locations (l_1, l_2), $\mathcal{F}_\psi(l_1, l_2)$ is an appropriate local store description for the values $\sigma_1(l_1)$ and $\sigma_2(l_2)$. Hence, we can consider $(\mathcal{F}_\psi(l_1, l_2))^1$ as the set of child locations traced from the contents of l_1 in store σ_1 (and similarly for $(\mathcal{F}_\psi(l_1, l_2))^2$ and the contents of l_2 in σ_2).

Having chosen $\mathcal{F}_\psi$, we must ensure that the choice is consistent with $\mathcal{S}$, which should in turn be consistent with the stores σ_1 and σ_2. The "global" store description $\mathcal{S}$ combines the local store descriptions of the roots with the local store descriptions of the contents of every pair of related reachable locations; the implicit requirement that $\mathcal{S}$ is defined ensures that the local beliefs of the roots and all the (pairs of) store contents are all compatible. The clauses $\mathsf{dom}(\sigma_1) \supseteq \mathcal{S}^1$ and $\mathsf{dom}(\sigma_2) \supseteq \mathcal{S}^2$ require that all of the reachable locations are actually in the two stores. Finally, $(j,\, \mathcal{F}_\psi(l_1, l_2),\, \sigma_1(l_1),\, \sigma_2(l_2)) \in \mathcal{V}$ ensures that the contents of locations l_1 and l_2 (in stores σ_1 and σ_2, respectively) with the local store description assigned by $\mathcal{F}_\psi$ are related (for $j < k$ steps).

Note that we do not require that $\mathcal{S}$ be the minimal set of locations reachable from the roots ψ. Such a requirement can be added but, as we will explain, is not necessary.

4.4 Related Computations

The computation relation $\mathcal{C}$ (see Figure 10) specifies when two closed terms e_1 and e_2 (with beliefs ψ, again corresponding to *at least* the locations appearing as sub-expressions of e_1 and e_2) are related for k steps. Informally, $\mathcal{C}$ says that if e_1 evaluates to a value v_1 in less than k steps and the evaluation is valid, then (1) there exists an evaluation derivation for e_2 that is also valid, and (2) given *any* valid evaluation of e_2 to some value v_2, it must be that v_1 and v_2 are related (with beliefs ψ_f). More precisely, we pick two starting stores σ_1 and σ_2 and a global store description $\mathcal{S}$ such that $\sigma_1, \sigma_2 :_k (\psi_s \odot \psi_r) \rightsquigarrow \mathcal{S}$, where ψ_r is the set of beliefs about the two stores held by the rest of the computation, i.e., the respective continuations. If a valid evaluation (i.e., where allocated locations are disjoint from those initially reachable in $\mathcal{S}^1$) of (σ_1, e_1) results in (v_1, σ_1', T_1) in $j < k$ steps, then the following hold: First, there must exist at least one valid evaluation for σ_2, e_2 (which may take any number of steps). Second, for any valid evaluation $\sigma_2, e_2 \Downarrow v_2, \sigma_2', T_2$ the following conditions should hold:

1. There must exist a set of beliefs ψ_f such that the values v_1 and v_2 are related for the remaining $(k - j)$ number of steps.
2. The following two sets of beliefs must be compatible: ψ_f (what v_1 and v_2 believe) and ψ_r (what the continuations believe — note that these beliefs remain unchanged).
3. There must exist a set of beliefs $\mathcal{S}_f$ about locations reachable from the new roots $(\psi_f \odot \psi_r)$ such that the final stores σ_1' and σ_2' satisfy the combined set of local beliefs $(\psi_f \odot \psi_r)$ and the global beliefs $\mathcal{S}_f$ for the remaining $k - j$ steps.
4. The set of reachable locations in σ_1' (and σ_2'), given by $\mathcal{S}_f^1$ (and $\mathcal{S}_f^2$), must be a subset of the locations reachable before evaluating e_1 (respectively e_2) — given by $\mathcal{S}^1$ (respectively $\mathcal{S}^2$) — and the locations allocated during this evaluation.

As noted earlier, the global store description $\mathcal{S}$ is not required to be the minimal set of locations reachable from the roots $(\psi_s \odot \psi_r)$, it only needs to include that set. This suffices because $\mathcal{S}_f^1$ and $\mathcal{S}_f^2$ only need to be subsets of $\mathcal{S}^1$ and $\mathcal{S}^2$ and the locations allocated during evaluation ($\mathsf{alloc}(T_1)$ and $\mathsf{alloc}(T_2)$). Thus, even though we may pick larger-than-necessary sets at the beginning of the evaluation, we ensure that we always add to them in a minimal way as the two evaluations progress.

4.5 Related Substitutions and Open Terms

Let $\Gamma = FV(e_1) \cup FV(e_2)$. We write $\Gamma \vdash e_1 \preccurlyeq e_2$ (pronounced "e_1 approximates e_2") to mean that for all $k \geq 0$, if γ_1 and γ_2 (mapping variables in Γ to closed values) are related substitutions with beliefs ψ_Γ (which is the combined local store description for the values in the range of γ_1 and γ_2), then $\gamma_1(e_1)$ and $\gamma_2(e_2)$, with root beliefs ψ_Γ, are related as computations for k steps. We write $\Gamma \vdash e_1 \approx e_2$ when e_1 approximates e_2 and vice versa, meaning that e_1 and e_2 are observationally equivalent.

4.6 Memo Elimination

We wish to prove $\Gamma \vdash e \approx e$ from which consistency — the property that any two valid evaluations of a closed term e in the same store yield observationally equivalent results — follows as a corollary. The proof of $\Gamma \vdash e \approx e$ proceeds by induction on the structure of e (see Theorem 4.4). Unfortunately, in the case of `memo` e, we cannot directly appeal to the induction hypothesis. To see why, consider the special case of the closed term `memo` e. We must show $(k, \{\}, \texttt{memo}\, e, \texttt{memo}\, e) \in \mathcal{C}$. Suppose (1) $\sigma_1, \sigma_2 :_k \{\} \odot \psi_r \rightsquigarrow \mathcal{S}$, (2) $\sigma_1, \texttt{memo}\, e \Downarrow^j v_1, \sigma_1', T_1$, and (3) $\mathcal{S}^1 \cap \mathsf{alloc}(T_1) = \emptyset$, where $j < k$. By the induction hypothesis we have $\emptyset \vdash e \approx e$ and hence $(k, \{\}, e, e) \in \mathcal{C}$. In order to proceed, we must instantiate the latter with two related stores (σ_1 and σ_2 are the only two stores we know of that are related) and provide a valid evaluation of e in the first store (i.e., we need $\sigma, e \Downarrow^{<k} -, -, T$ where T is such that $\mathcal{S}^1 \cap \mathsf{alloc}(T) = \emptyset$). From (2), by the operational semantics, we have $\sigma_{01}, e \Downarrow^{j_1} v_1, \sigma_{01}', T_{01}$ and $\sigma_1, T_{01} \curvearrowright^{j_2} \sigma_1', T_1$, where $j = j_1 + j_2$. But we know nothing about the store σ_{01} in which e was evaluated. What we need is a derivation for $\sigma_1, e \Downarrow^{\leq j} v_1, \sigma_1', T_1$. That is, we must show that evaluation in some store s_{01} followed by change propagation yields the same results as a from-scratch run in the store σ_1.

[2] To be precise, our definition requires only that $\mathcal{S}$ *include* the set of reachable locations.

$$
\begin{array}{rcl}
\mathcal{V}^{\mathsf{M}} & = & \{ (k, ()) \} \cup \{ (k, n) \} \cup \{ (k, l) \} \cup \\
& & \{ (k, \lambda x. e) \mid \forall j < k.\ \forall v.\ (j, v) \in \mathcal{V}^{\mathsf{M}} \implies \\
& & \qquad\qquad\qquad (j, e[v/x]) \in \mathcal{C}^{\mathsf{M}} \} \cup \\
& & \{ (k, (v, v')) \mid (k, v) \in \mathcal{V}^{\mathsf{M}} \wedge (k, v') \in \mathcal{V}^{\mathsf{M}} \} \cup \\
& & \{ (k, \mathtt{inl}\, v) \mid (k, v) \in \mathcal{V}^{\mathsf{M}} \} \cup \\
& & \{ (k, \mathtt{inr}\, v) \mid (k, v) \in \mathcal{V}^{\mathsf{M}} \}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}^{\mathsf{M}} & = & \{ (k, e) \mid \forall j < k.\ \forall \sigma_0, \sigma_0', \sigma, \sigma', v, T, T', j_1, j_2. \\
& & \quad \sigma_0, e \Downarrow^{j_1} v, \sigma_0', T \wedge \sigma, T \curvearrowright^{j_2} \sigma', T' \wedge \\
& & \quad j = j_1 + j_2 \implies \\
& & \quad \sigma, e \Downarrow^{\leq j} v, \sigma', T' \wedge (k - j, v) \in \mathcal{V}^{\mathsf{M}} \}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G}^{\mathsf{M}}[\![\emptyset]\!] & = & \{ (k, \emptyset) \} \\
\mathcal{G}^{\mathsf{M}}[\![\Gamma, x]\!] & = & \{ (k, \gamma[x \mapsto v]) \mid (k, \gamma) \in \mathcal{G}^{\mathsf{M}}[\![\Gamma]\!] \wedge (k, v) \in \mathcal{V}^{\mathsf{M}} \}
\end{array}
$$

$$
\Gamma \vdash e \overset{\text{def}}{=} \forall k \geq 0.\ \forall \gamma.\ (k, \gamma) \in \mathcal{G}^{\mathsf{M}}[\![\Gamma]\!] \implies (k, \gamma(e)) \in \mathcal{C}^{\mathsf{M}}
$$

Figure 13. Logical Predicate for Memo Elimination

To prove that each memo hit can be replaced by a regular evaluation (Lemma 4.3), we define a logical predicate (a unary logical relation) for memo elimination. Figure 13 defines $\mathcal{V}^{\mathsf{M}}$ and $\mathcal{C}^{\mathsf{M}}$ as sets of pairs (k, v) and (k, e) respectively, where k is the step index, v is a closed value, and e is a closed term. Essentially, $(k, e) \in \mathcal{C}^{\mathsf{M}}$ means that e has the memo-elimination property (i.e., if $\sigma_0, e \Downarrow^{j_1} v, \sigma_0', T$ and $\sigma, T \curvearrowright^{j_2} \sigma', T'$, then $\sigma, e \Downarrow^{\leq j_1 + j_2} v, \sigma', T'$), and if the combined evaluation plus change propagation consumed $j = j_1 + j_2$ steps, then v has the memo-elimination property for the remaining $k - j$ steps. Clearly, all values v have the memo-elimination property: since v is already a value, it evaluates to itself in 0 steps, producing the empty trace, which means that change propagation takes 0 steps and leaves both store and trace unchanged. Since a function is a suspended computation, we must require that its body also have the memo-elimination property for one fewer step.

Lemma 4.2. *If* $\Gamma{=}FV(e)$, *then* $\Gamma \vdash e$.

Proof sketch: By induction on the step index k and nested induction on the structure of e. All cases are straightforward. The only interesting case is that of read/no ch. where we read a value v out of the store and then have to use the outer induction hypothesis to show that v has the memo-elim property for a *strictly fewer* number of steps before we can plug v into the body of the read, appealing to the inner induction hypothesis to complete the proof. □

Corollary 4.3 (Memo Elimination). *Let e be a closed term, possibly with free locations. If* $\sigma_0, e \Downarrow^{j_1} v, \sigma_0', T$ *and* $\sigma, T \curvearrowright^{j_2} \sigma', T'$, *then* $\sigma, e \Downarrow^{\leq j_1 + j_2} v, \sigma', T'$.

4.7 Consistency

For lack of space we have omitted proof details here. More detailed proofs can be found in our extended technical report [2].

Theorem 4.4. *If* $\Gamma{=}FV(e)$, *then* $\Gamma \vdash e \approx e$.

Proof sketch: By induction on the structure of e. As explained above, in the memo case we use Lemma 4.3 before we can appeal to the induction hypothesis. Other interesting cases include mod, where the valid evaluation requirement (that the evaluation not allocate locations reachable from the initial expression) is critical in order to extend the bijection on locations; write, where the fact that the locations l_1 and l_2 being written to are reachable from the initial expression guarantees that (l_1, l_2) is already in the bijection; and read, where we need to know that the values being read are related, which we can conclude from the fact that the locations being read are reachable and related, together with the fact that related locations have related contents which follows from store relatedness. □

Theorem 4.5 (Consistency). *If* $\Gamma{=}FV(e)$, *then* $\Gamma \vdash e \approx^{ctx} e$.

Let us write $\Downarrow_{\emptyset}^{k}$ instead of $\Downarrow^{k}$ for evaluation judgments that have at least one derivation where every use of the $\mathbf{memo}$ rule picks $\sigma_0 = \sigma$. Such a derivation describes an evaluation without memo hits, i.e., that of an *ordinary imperative programs*. Since memo elimination (Lemma 4.3) can be applied repeatedly until no more memo-hits remain, we obtain the following result, which can be seen as a statement of *correctness* since it relates the self-adjusting semantics to an ordinary non-adjusting semantics:

Lemma 4.6 (Complete Memo Elimination). *Let e be a closed term, possibly with free locations. If* $\sigma, e \Downarrow^{k} v, \sigma', T$, *then* $\sigma, e \Downarrow_{\emptyset}^{\leq k} v, \sigma', T'$.

5. Implementation

Implementing the semantics directly would not be efficient because change-propagation traverses the complete trace. In this section, we describe data structures and algorithms for implementing the semantics efficiently.[3] The basic ideas are similar to those used in the pure setting which we described in previous work [3]. The details, however, are quite different: to handle mutable modifiables we make the computation dynamically persistent by keeping track of different versions of each modifiable. This techniques is inspired by previous work on persistent data structures [16].

5.1 Data Structures

Our algorithms require several data structures including an order-maintenance data structure and searchable ordered sets described below, as well as standard priority queues.

Order Maintenance (Time Stamps). We assume a data structure that maintains a set of *time stamps* while supporting all of the following operations in constant time: create a new time stamp, insert a newly created time stamp after another, delete a time stamp, and compare two time stamps [15].

Searchable Time-Ordered Sets. We assume a time-ordered sets data structure that supports the following operations.

- `new`: return a new empty set.
- `build` X: allocate and return a new data structure containing all the element in the set X.
- `find` (t): return the earliest element at time t or later.
- `prev` (t): return the element preceding t within the set.
- `insert` (x, t): insert the element x into the set at time t.
- `delete` (x): delete the element x from the set.

Using balanced binary search trees, all operations other than `build` require $O(\log m)$ time. When the sizes of the sets are constant, then a simple list based representation can be used to support all operations in $O(1)$ time.

5.2 The Primitives

To support the self-adjusting computation primitives we maintain a global time line and a global priority queue. The time line is an instance of an order-maintenance data structure with `current-time` pointing to the "current time" of the computation. During the initial run, the current-time is always the last time stamp, but during change propagation it can be any time in the past. The global priority queue contains a set of *affected readers* prioritized by their

[3] The source code for our implementation can also be found on the web at `http://ttic.uchicago.edu/~umut/imperative/impl.tar`.

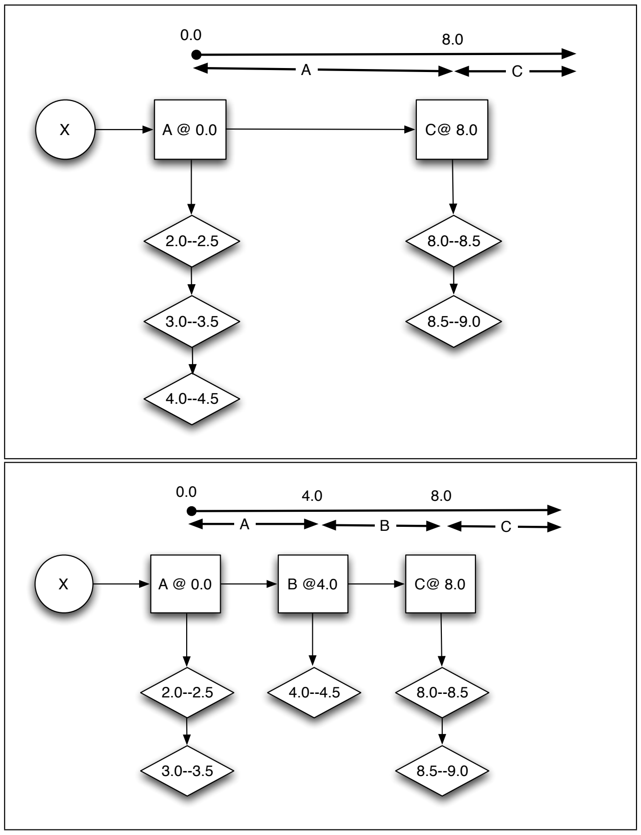

Figure 14. A modifiable, its writes and reads before (top) and after performing a write (bottom).

start time (we define readers more precisely below). In addition, we maintain a time stamp called *end-of-memo* for memoization purposes.

Modifiable References. We represent a modifiable reference as a pair consisting of a *version set* and a *reader set*, both of which are represented as searchable, time-ordered sets. The reader set of a modifiable l contains all the read operations whose source is l. The version set contains all the different contents that the modifiable takes over time. More precisely, the version set contains pairs (v, t) consisting of a value v and a time stamp t. The reader set contains *readers*, each of which is a triple (t_s, t_e, f) consisting of a start time t_s, end time t_e, and a function f corresponding to the body of the read operation.

Based on this representation, the operations on modifiable references can be performed as follows.

`newMod`**:** Create an empty version set and an empty reader set. Return a pointer to the pair containing them.

`read`**:** Identify the version (v, t_v) of the modifiable being read that comes immediately before the `current-time` by performing a combination of `find` and `prev` operations on the version set. Advance time to a new time stamp t_s. (To do that, create t_s, insert it after `current-time`, and set `current-time` to t_s.) Apply the body of the read to v. When the body returns, advance time again to a new t_e. Insert the reader r consisting of the body and the time interval(t_s, t_e) into the reader set of the modifiable being read.

`write`**:** Advance time to a new t_w. Create a new version consisting of the value v being written and t_w. Insert it into the modifiable being written. Inserting a new version can affect the readers whose start time comes after t_w but before the next version. To identify the affected readers, find the first read that comes at or after t_w by repeatedly performing `find` operations until we find a reader that come before the next version. We then delete these readers from the reader set and insert them into the priority queue of affected readers. Note that during the initial run, all writes take place at the most recent time. Thus, there are no affected readers.

`deref`**:** Identify the version of the dereferenced modifiable at the current time (v, t) by using a `find` operation and return v.

`change`**:** Identify the earliest version of the changed modifiable after time 0.0 by using a `find` operation, change the value of this version to the specified value, and inserts all the readers of this version into the priority queue. The readers of the version can be found by finding the next version (if any) and inserting all the readers between the two versions.

Figure 14 illustrates a particular representation of modifiables assuming that time stamps are real numbers and time-ordered sets are represented as sorted lists. The modifiable `x` points to version list (versions are drawn as squares) consisting of versions at the specified write; the versions are sorted with respect to their times. Each version points to a reader list (readers are drawn as diamonds) whose start and end times are specified. The readers stored in the reader list of a version are the ones that read that version; they are sorted with respect to their start times. Thus all the readers take place between the time of the version and the time of the next version. For example, in the top figure, all readers of version `A` take place between times `0.0` and `8.0`; the readers of version `C` take place after `8.0`. The bottom figure illustrates how the reads may be arranged if we create a new version `B` at time `4.0`. When this happens the reader that starts at time `4.0` will be become affected and will be inserted into the priority queue.

Memoization and Change Propagation. Figure 15 shows the pseudo code for memoization and change propagation. These operations are based on an `undo` function for rolling back the effects of a computation between two time stamps.

The `undo` primitive takes start and end time stamps t_s and t_e, and undoes the computation by deleting all its versions, readers, memo entries, and time stamps between t_s and t_e. For each time stamp t between t_s and t_e it checks if there is a version, reader, or memo entry at t. To delete a reader starting at t, it is dropped from both its reader set and the queue (if it was enqueued). To delete a memo entry that starts at t, it is simply removed from the memo table. Deleting a version is more complicated because it can affect the reads that come after it. To delete a version v at time t, we first identify the time t' of the earliest version in its version set that comes after it. (If none exists, then t' will be t_∞.) We then find all readers between t and t' and insert them into the priority queue. Since they may now read different values than they did before, they are affected by the deletion of v.

To create memoized functions, the library provides a `memo` primitive. A memoized function has access to a memo table for storing and re-using results. Each call takes the list of the arguments of the client function (`key`) and the client function itself. Before executing the client function, a memo lookup is performed. If no result is found, then a start time stamp t_s is created, the client is run, an end time stamp t_e is created, and the result along with interval (t_s, t_e) is stored in the memo table. If a result is found, then computations between the current time and the start of the memoized computation are undone, a change-propagation is performed into the memoized computation, and the result is returned. A memo lookup succeeds if and only if there is a result in the memo table whose key is the same key as that of the current call and whose time interval is nested within the current time interval

```
undo ($t_s, t_e$) =
  for each $t$. $t_s < t < t_e$ do
    if there is a version $v = (v_t, t)$ then
      $t'$ $\leftarrow$ time of successor($v$)
      delete version $v$ from its version-set
      $R$ $\leftarrow$ $\{r$ | $(t_1, t_2, f)$ is a reader $\wedge\, t < t_1 < t'\}$
      for each $r \in R$ do
        delete $r$ from its reader set
    if there is a reader $r = (t, \_, \_)$ then
      delete $r$ from its readers-set and from $Q$
    if there is a memo entry $m = (t, \_, \_)$ then
      delete $m$ from its table
    delete $t$ from time-stamps

memo () =
let
  table $\leftarrow$ new memo table
  fun mfun key f =
    case (find (table, key, now)) of
      NONE =>
        $t_1$ $\leftarrow$ advance-time ()
        $v$ $\leftarrow$ f ()
        $t_2$ $\leftarrow$ advance-time
        insert $(v, t_1, t_2)$ into table
        return $v$
      SOME $(v, t_1, t_2)$ =>
        undo (current-time,$t_1$)
        propagate ($t_2$)
        return $v$
in
  mfun
end

propagate ($t$) =
  while $Q \neq \emptyset$ do
    $(t_s, t_e, f)$ $\leftarrow$ checkMin ($Q$)
    if $t_s < t$ then
      deleteMin ($Q$)
      current-time $\leftarrow$ $t_s$
      tmp $\leftarrow$ end-of-memo
      end-of-memo $\leftarrow$ $t_e$
      f ()
      undo (current-time, $t_e$)
      end-of-memo $\leftarrow$ tmp
    else
      return
```

Figure 15. Pseudo code for memo and propagate primitives.

defined by the current time and end-of-memo. This lookup rule is critical to correctness. Informally, it ensures that side-effects are incorporated into the current computation accurately. (In the formal semantics, it corresponds to the integration of the trace of the re-used computation into the current trace.)

Undoing the computation between current-time and the start of the memoized computation serves some critical purposes: (1) It ensures that all versions read by the memoized computation are updated, and (2), it ensures that all computations that contain this computation are deleted and, thus, cannot be re-used.

The change propagation algorithm takes a queue of affected readers (set up by change operations) and processes them until the queue becomes empty. The queue is prioritized with respect to start time so that readers are processed in correct chronological order. To process a reader, we set current-time to the start time of the reader t_s, and run its body. After the body returns, we delete the computation between the current time and t_e.

5.3 Asymptotic Complexity

Self-adjusting computation primitives can be supported efficiently. For the analysis we distinguish between an *initial-run*, i.e., a from-scratch run of a self-adjusting program, and change propagation. Due to space constraints, we omit the proofs of these theorems and make them available separately [2].

Theorem 5.1 (Overhead). *All self-adjusting computation primitives can be supported in excepted constant time during the initial run, assuming that all memo functions have unique sets of keys. The expectation is taken over internal randomization used for representing memo tables.*

For the analysis of change propagation, we define several performance measures. Consider running the change-propagation algorithm, and let A denote the set of all affected readers, i.e., the readers that are inserted into the priority queue. Of these readers, some are re-evaluated and the others are deleted; we refer to the set of re-evaluated readers as A_e and the set of deleted readers as A_d. For a re-evaluated reader $r \in A_e$, let $|r|$ be its re-evaluation time complexity assuming that all self-adjusting primitives take constant time. Note that a re-evaluated r may re-use part of a previous computation via memoization and, therefore, take less time than a from-scratch re-execution. Let n_t denote the number of time stamps deleted during change propagation. Let n_q be the maximum size of the priority queue at any time during the algorithm. Let n_{rw} denote the maximum number of readers and versions (writes) that each modifiable may have.

Theorem 5.2 (Change Propagation). *Change propagation takes time in:*

$$O\left(|A| \log n_q + |A| \log n_{rw} + \sum_{r \in A_e} |r| \log n_{rw} + n_t \log n_{rw}\right)$$

For a special class of computations, where there is a constant bound on the number of times each modifiable is read and written, i.e., $n_{rw} = O(1)$, we have the following corollary.

Corollary 5.3 (Change Propagation with Constant Reads & Writes). *In the presence of a constant bound on the number of reads and writes per modifiable, change propagation takes*

$$O\left(|A| \log n_q + \sum_{r \in A_e} |r|\right).$$

amortized time where the amortization is over a sequence of change propagations.

Proof. By plugging in $n_{rw} = O(1)$ into Theorem 5.2, we get $O\left(|A| \log n_q + \sum_{r \in A_e} |r| + n_t\right)$. Since each time stamp is deleted at most once, and since the work performed during the deletion is constant when $n_{rw} = O(1)$, we can amortize the time for deletion to the creation of the time. Thus, change-propagation takes $O\left(|A| \log n_q + \sum_{r \in A_e} |r|\right)$ amortized time. □

5.4 Complexity of Depth First Search

We prove the theorems from Section 2 for DFS and topological-sorting.

Theorem 5.4 (DFS). *Disregarding read operations performed by the visitor function, the* depthFirstSearch *program responds to changes in time $O(m)$, where m is the number of affected nodes after an insertion/deletion.*

Proof. Let G be a graph and T be its DFS-tree. Let G' be a graph obtained from G by inserting an edge (u, v) into G. The first read affected by this change will be the one that corresponds to the edge (u, v) that visits u. There are a few cases to consider. If v is visited, then there is nothing to be done and change propagation completes in constant time. If v is not visited, then the algorithm visits v and start exploring out from v by traversing its out edges. Since all of these out-edges will be writing their results in newly allocated destination, none of these visits will cause a memo match.

Since each visit takes constant time, the total time will be $O(m)$. After the algorithm returns to v, it will return to node u. From this point on, there will be no other executed reads change propagation will complete. The only read that is ever inserted into the queue will be the one that corresponds to the edge (u, v) and, thus, queue size will no exceed one. By Theorem 5.2, the total time for change proapgation is $O(m)$. The case for deletions is symmetric. □

Theorem 5.5 (Topological Sort). *Change propagation updates the topological sort of a graph in $O(m)$ time where m is the number of affected nodes.*

Proof. Consider insertion of an edge (u, v). Since with modifiable references appends can be performed in constant time, each call of the visitor takes constant time. Thus the traversal of the affected component takes $O(m)$ time. After this traversal, the result from the inserted edge will be returned to the source of the edge u, which will return a list where u is the head of the list. This list will be identical to the list that is returned in the previous execution. Therefore, no more reads will need to be re-executed and change propagation completes in $O(m)$ time. □

6. Conclusions

Self-adjusting computation has been a success story so far. However, since previous techniques were limited to a purely functional programming model, there are broad classes of problems for which they could not yield optimal results.

In this paper we proposed to lift the restriction that modifiable references be written exactly once during any evaluation, thereby re-introducing an imperative programming model. Concretely:

- We defined a set of primitives (an "API") for imperative self-adjusting programming and implemented a prototype in ML.
- We described our implementation strategy which is based on a persistent graph structure for representing all versions of an imperative variable as its contents evolve over time.
- We proved that imperative self-adjusting programs are as efficient as self-adjusting programs in the pure setting if the number n_{rw} of read and write operations per location is bounded by a constant. More generally, without such a bound on n_{rw}, there is a $\log n_{rw}$ slowdown over the pure setting.
- We showed, using the example of depth-first search, that typical graph problems find convenient expression. In fact, self-adjusting imperative programs resemble their non-adjusting counterparts quite closely.
- To formalize imperative self-adjusting programming we defined the language AIL and gave it an operational semantics. The semantics consists of mutually recursive rule sets for deriving *evaluation judgments* and *change-propagation judgments*.
- We proved this semantics *consistent*, meaning that despite the presence of non-determinism due to allocation and memoization, the result of any evaluation of a program e is equivalent to the result of any other evaluation of the same program e in the same store.
- To be able to prove this result in the imperative setting, we made use of a novel form of syntactic logical relations. Since AIL is untyped, the relation is indexed only by a step counter and not by a type. In addition to the step index, our definition makes use of *bijections on reachable locations* in order to deal with memory locations, mutability, and possibly cyclic heap data structures.

References

[1] U. A. Acar and B. Hudson. Optimal-time dynamic mesh refinement with quad trees and off centers. Submitted. Technical Report CMU-CS-07-121, Dept. of Computer Science, Carnegie Mellon Univ., 2007.

[2] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. http://tti.uchicago.edu/~umut/imperative/tr.pdf.

[3] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *PLDI'06*, June 2006.

[4] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and J. L. Vittes. Kinetic algorithms via self-adjusting computation. In *European Symposium on Algorithms (ESA)*, pages 636–647, September 2006.

[5] U. A. Acar, G. E. Blelloch, and K. Tangwongsan. Kinetic 3d convex hulls via self-adjusting computation (an illustration. In *Symposium on Computational Geometry (SCG)*, 2007.

[6] U. A. Acar, M. Blume, and J. Donham. A consistent semantics of self-adjusting computation. In *ESOP'07*, Mar. 2007.

[7] U. A. Acar, A. Ihler, R. Mettu, and O. Sümer. Bayesian inference for dynamically changing graphs, 2007. Submitted.

[8] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP'06*, pages 69–83, Mar. 2006.

[9] A. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *ICFP'05*, pages 78–91, Sept. 2005.

[10] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, Sept. 2001.

[11] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA'05*, pages 86–101, 2005.

[12] N. Bohr and L. Birkedal. Relational reasoning for recursive types and references. In *APLAS'06*, Nov. 2006.

[13] M. Carlsson. Monads for incremental computing. In *ICFP'02*, pages 26–35, 2002.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[15] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *STOC'87*, pages 365–372, 1987.

[16] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38 (1):86–124, Feb. 1989.

[17] M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[18] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL'06*, Jan. 2006.

[19] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, May 1995.

[20] D. J. Pearce and P. H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *J. Exp. Algorithmics*, 11:1.7, 2006.

[21] N. Pippenger. Pure versus impure lisp. *TOPLAS*, 19(2):223–238, 1997.

[22] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *LICS'96*, June 1996.

[23] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *MFCS*, volume 711 of *LNCS*, pages 122–141. Springer-Verlag, 1993.

[24] M. D. Santambrogio, V. Rana, S. O. Memik, D. Sciuto, and U. A. Acar. A novel SoC design methodology for combined adaptive software descripton and reconfigurable hardware. In *IEEE/ACM Intl. Conf. on Computer Aided Design (ICCAD)*, 2007. To Appear.

[25] A. Shankar and R. Bodik. Ditto: Automatic incrementalization of data structure invariant checks (in java). In *PLDI'07*, June 2007.

[26] K. Sieber. New steps towards full abstraction for local variables. In *ACM SIGPLAN Workshop on State in Prog. Langs.*, 1993.

[27] I. D. B. Stark. *Names and Higher-Order Functions*. Ph. D. dissertation, Univ. of Cambridge, Cambridge, England, December 1994.