Imperative Self-Adjusting Computation

Umut A. Acar [*] Amal Ahmed Matthias Blume

Toyota Technological Institute at Chicago

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

Abstract

Self-adjusting computation enables writing programs that can automatically and efficiently respond to changes to their data (e.g., inputs). The idea behind the approach is to store all data that can change over time in *modifiable references* and to let computations construct *traces* that can drive *change propagation*. After changes have occurred, change propagation updates the result of the computation by re-evaluating only those expressions that depend on the changed data. Previous approaches to self-adjusting computation require that modifiable references be written at most once during execution—this makes the model applicable only in a purely functional setting.

In this paper, we present techniques for imperative self-adjusting computation where modifiable references can be written multiple times. We define a language SAIL (Self-Adjusting Imperative Language) and prove *consistency*, i.e., that change propagation and from-scratch execution are observationally equivalent. Since SAIL programs are imperative, they can create cyclic data structures. To prove equivalence in the presence of cycles in the store, we formulate and use an untyped, step-indexed logical relation, where step indices are used to ensure well-foundedness. We show that SAIL accepts an asymptotically efficient implementation by presenting algorithms and data structures for its implementation. When the number of operations (reads and writes) per modifiable is bounded by a constant, we show that change propagation becomes as efficient as in the non-imperative case. The general case incurs a slowdown that is logarithmic in the maximum number of such operations. We describe a prototype implementation of SAIL as a Standard ML library.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.2.0 [*Analysis of Algorithms and Problem Complexity*]: General; F.3.2 [*Semantics of Programming Languages*]: Operational Semantics

General Terms Languages, Design, Algorithms

Keywords Self-adjusting computation, incremental computation, step-indexed logical relations, imperative programming, change propagation, memoization, mutable state

[*] Acar is supported by a gift from Intel.

1. Introduction

Self-adjusting computation concerns the problem of updating the output of a computation while its input undergoes small changes over time. Recent work showed that a combination of *dynamic dependence graphs* (Acar et al. 2006c) and a particular form of memoization (Acar et al. 2003) can be combined to update computation orders of magnitudes faster than re-computing from scratch (Acar et al. 2006b). The approach has been applied to a number of problems including invariant checking (Shankar and Bodik 2007), computational geometry and motion simulation (Acar et al. 2006d), and statistical inference on graphical models (Acar et al. 2007c).

In self-adjusting computation, the programmer stores all data that can change over time in *modifiable references* or *modifiables* for short. Modifiables are write-once references: programs can read a modifiable as many times as desired but must write it exactly once. After a self-adjusting program is executed, the programmer can change the contents of modifiables and update the computation by performing *change propagation*. For efficient change propagation, as a program executes, an underlying system records the operations on modifiables in a *trace* and memoizes the function calls. Change propagation uses the trace, which is represented by a dynamic dependence graph, to identify the reads of changed modifiables and re-evaluates them. When re-evaluating a read, change-propagation re-uses previous computations via memoization.

By requiring that modifiables be written exactly once at the time of their creation, the approach ensures that self-adjusting programs are purely functional; this enables 1) inspecting the values of modifiables at any time in the past, 2) re-using computations via memoization. Since change propagation critically relies on these two properties, it was not known if self-adjusting computation could be made to work in an imperative setting. Although purely functional programming is fully general (i.e., Turing complete), it can be asymptotically slower than imperative models of computation (Pippenger 1997). Also, for some applications (e.g., graphs), imperative programming can be more natural.

In this paper, we generalize self-adjusting computation to support imperative programming by allowing modifiables to be written multiple times. We describe an untyped language, called SAIL (Self-Adjusting Imperative Language), that is similar to a higher-order language with mutable references. As in a conventional imperative language, a write operation simply updates the specified modifiable. A read operation takes the modifiable being read and the expression (the body of the scoped read) that uses the contents of the modifiable. To guarantee that all dependencies are tracked, SAIL ensures that values that depend on the contents of modifiables are themselves communicated via modifiables by requiring that read operations return a fixed (unit) value. Compared to a purely functional language for self-adjusting computation, SAIL is somewhat simpler because it does not have to enforce the write-once requirement on modifiables. In particular, purely functional languages for self-adjusting computation make a modal distinction between *stable* and *changeable* computations, e.g., Acar et al. (2006c), which is not necessary in SAIL.

We describe a Standard ML library for SAIL that allows the programmer to transform ordinary imperative programs into self-adjusting programs (Section 2). The transformation requires replacing the references in the input with modifiables and annotating the code with SAIL primitives. As an example, we consider depth-first-search and topological sorting on graphs. The resulting self-adjusting programs are algorithmically identical to the standard approach to DFS and topological sort, but via change propagation they can respond to changes faster than a from-scratch execution when the input is changed.

We formalize the operational semantics of SAIL and its change propagation semantics (Section 3). In the operational semantics, evaluating an expression returns a value and a trace that records the operations on modifiables. The semantics models memoization by using non-determinism: a memoized expression can either be evaluated to a value (a memo miss) or its trace and result can be re-used by applying change propagation to the trace of a previous evaluation (a memo hit).

We prove that the semantics of SAIL is *consistent*, i.e., that any two evaluations of the same expressions are observationally (or contextually) equivalent (Section 4). Since SAIL programs are imperative, it is possible to construct cycles in the store. This makes reasoning about equivalence challenging. In fact, in our prior work (Acar et al. 2007b), we proved consistency for purely functional self-adjusting computation by taking critical advantage of the absence of cycles. More specifically, we defined the notion of equivalence by using *lifting* operations that eliminated the store by substituting the contents of locations directly into expressions; lifting cannot even be defined in the presence of cycles.

Reasoning about equivalence of programs in the presence of mutable state has long been recognized as a difficult problem. The problem has been studied extensively starting with ALGOL-like languages which have local updatable variables but no first-class references (O'Hearn and Tennent 1995; Sieber 1993; Pitts 1996). The problem gets significantly harder in the presence of first class references and dynamic allocation (Pitts and Stark 1993; Stark 1994; Benton and Leperchey 2005) and harder still in the presence of cyclic stores. We know of only two recent results for proving equivalence of programs with first-class mutable references and cyclic stores, a proof method based on bisimulation (Koutavas and Wand 2006) and another on (denotational) logical relations (Bohr and Birkedal 2006) (neither of which is immediately applicable to proving the consistency of SAIL).

We prove consistency for imperative self-adjusting programs using *syntactic logical relations*, that is, logical relations based on the operational semantics of the language (not on denotational models). We use logical relations that are indexed *not by types* (since SAIL is untyped), but by a natural number that, intuitively, records the number of steps available for future evaluation (Appel and McAllester 2001; Ahmed 2006). The stratification provided by the step indices is essential for modeling the recursive functions (available via encoding **fix**) and cyclic stores present in the language.

We show that SAIL accepts an asymptotically efficient implementation by describing data structures for supporting its primitives and by giving a change propagation algorithm (Section 5). For each modifiable we keep all of its different contents, or *versions*, over time. The write operations create the versions. For example, writing the values values `0` and then `5` into the same modifiable `m` creates two versions of `m`. This versioning technique, which is inspired by previous work on persistent data structures (Driscoll et al. 1989), enables keeping track of the relationship between read operations and the values that they depend on. We keep the versions of a modifiable in a *version-set* and its readers in a *reader-set*. We represent these sets as searchable time-ordered sets that support various operations such as *find*, *insert*, and *delete*, all in time logarithmic in the size of the set. Using these data structures, we describe a change-propagation algorithm that implements the trace-based change propagation of the SAIL semantics 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 non-imperative case. When the number of writes is not bounded by a constant, then change propagation incurs a logarithmic overhead in the maximum number of writes and reads to any modifiable.

```
signature SELF_ADJUSTING =
  sig
    eqtype 'a mod

    val mod: ('a * 'a -> bool) -> '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 on graphs.

2.1 The ML library

Figure 1 shows the signature of our library. The library defines the equality type for modifiables `'a mod` and provides functions to create (`mod`), read (`read`), write (`write`), and hash (`hashMod`) modifiables. For memoization, the library provides the `memo` function. We define a *self-adjusting program* as a Standard ML program that uses these functions. 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 is created by the `mod` function that takes a conservative equality test on the contents of the modifiable and returns an uninitialized modifiable. A conservative equality 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 unnecessary change propagation by detecting that a `write` or `change` operation does not change the contents of a modifiable. For each modifiable allocated, the `mod` function generates a unique integer tag, which is returned by the `hashMod` function. The hashes are used when memoizing function calls. A `read` takes the modifiable to be read and a *reader* function, applies the contents of the modifiable to the reader, and returns unit. By making `read` operations return unit, the library ensures that no variables other than those bound by the readers can depend on the contents of modifiables. This forces readers to communicate by writing to modifiable references and makes it possible to track all dependencies by recording the operations on modifiables.

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

```
1   datatype node =
2      empty
3    | node of int * bool ref * node ref * node ref
4   fun depthFirstSearch f root =
5   let
6
7     fun dfs rn =
8       case !rn of
9         empty => f (NONE,NONE)
10      | node (id,rv,rn1,rn2) =>
11
12          if !rv then
13            let
14              val () = rf := true
15              val res1 = dfs rn1
16              val res2 = dfs rn2
17            in
18
19            f (SOME id, SOME(res1, res2))
20            end
21          else
22            NONE
23  in
24      dfs root
25  end
```

```
1  datatype node =
2     empty
3   | node of int * bool mod * node mod * node mod
4  fun depthFirstSearch eq f root =
5  let
6    val mfun = memo ()
7    fun dfs rn = let val rres = mod eq in read rn (fn n =>
8      case n of
9        empty => write rres (f(NONE,NONE))
10     | node (id,rv,rn1,rn2) =>
11         mfun [id, #rv, #rn1, #rn2, #rres] (fn () =>
12         read rv (fn v => if v then
13           let
14             val () = write rf true
15             val rres1 = dfs rn1
16             val rres2 = dfs rn2
17           in
18             read rres1 (fn res1 => read rres2 (fn res2 =>
19             write rres (f(SOME id, SOME(res1, res2)))))
20           end))
21         else
22           write rres NONE) end
23 in
24   dfs root
25 end
```

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

facilitate efficient memo lookups, memoized functions must hash their arguments to integers uniquely. This can be achieved by using standard boxing or tagging techniques.

Our library does not provide mechanisms for statically or dynamically ensuring the correctness of self-adjusting programs, i.e., for ensuring that change propagation updates computations correctly. We instead expect the programmer to adhere to certain *correct-usage* requirements. In particular, correct-usage requires that all the free variables of a memoized function be listed as an argument to the memoized function and be hashed, and that a memoized function be used to memoize only one function. It may be possible to enforce these requirements but this may require a significant burden on the programmer (Acar et al. 2006a).

2.2 Writing Self-Adjusting Programs

As an example of how modifiables can be used, Figure 2 shows the complete code for a depth-first-search (DFS) function on a graph with ordinary references (left) and with modifiable references (right). A graph is defined to be either empty or a node consisting of an integer identifier, a boolean *visited* flag, and two pointers to its neighbors. In the non-self-adjusting code, the neighbor pointers and the visited flag are placed in ordinary references; in the self-adjusting code, these are placed in modifiable references.

The static `depthFirstSearch` function takes a *visitor* 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 `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 the visitor 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 (the node was not visited before), then the flag is first set to true, the neighboring nodes are visited recursively, the results of the neighbors along with the identity of the visited node are passed to the visitor (`f`) to compute the result, and the result is written. Since during a DFS, the visited flag starts with value `false` and is later set to `true`, the DFS requires updateable modifiables.

We transform the static code into self-adjusting code by first replacing all the references in the input (the graph) with modifi-

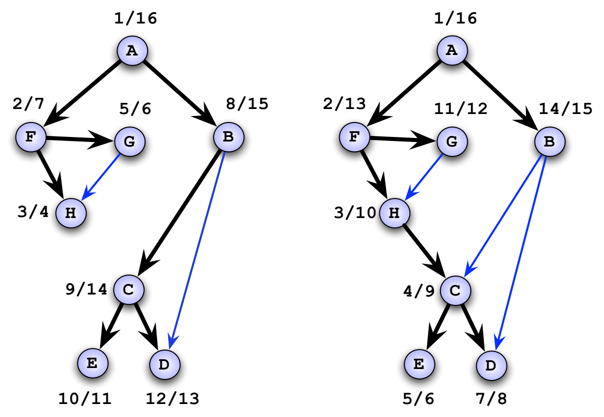

Figure 3. A graph before and after insertion of the edge `(H,C)`.

ables. We then replace dereference operations with `read` operations. Since the contents of modifiables are accessible only locally within the body of the read, when inserting a read operation, we identify the part of the code that becomes the body of the read. In our example, we insert a read for accessing the visitor flag (line 12). Since read operations can only return unit, we need to allocate a modifiable for the result to be written (line 7). To allocate the result modifiable, we use an equality function on the result type provided as an argument to the `depthFirstSearch` function. We finish the transformation by memoizing the `dfs` function. This requires creating a memo function (line 11) and applying it to the recursive branch of the case statement; since the base case performs constant work, it does not benefit from memoization. For brevity, in the code, we use `#` for `hashMod`.

One application of DFS is topological sort, which requires finding an ordering of nodes where every edge goes from a smaller to a larger node. As an example, consider the graphs in Figure 3, where each node is tagged with the first and the last time they were visited by the DFS algorithm. 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, e.g., Cormen et al. (1990). In Figure 3, 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`. We can compute the

```
structure SAIL: SELF_ADJUSTING = ...
structure Graph =
struct
  fun fromFile s = ...
  fun node i = ...
  fun newEdgeFrom (i) = ...
  fun DFSVisitSort = ...
end

fun test (s,i,j) =
let
   val _ = SAIL.init ()
   val (root,graph,n) = Graph.fromFile s
   val r = depthFirstSearch Graph.DFSVisitSort (root)

   val nr = Graph.newEdgeFrom (i)
   val () = SAIL.change nr (Graph.node j)
   val () = SAIL.propagate ();
in
  r
end
```

Figure 4. Example of changing input and change propagation.

topological sort of a graph by using the `depthFirstSearch` function (Figure 2). To do this, we first define the result type and its equality function, in this case a list consisting of the identifiers of the nodes in topologically sorted order. Since modifiable references accept equality, we can use ML's "equals" operator for comparing modifiables as follows.

```
datatype 'a list = nil | cons 'a * ('a list) mod
fun eqList (a,b) =
  case (a,b) of
    (nil,nil) => true
  | (ha::ta,hb::tb) =>ha=hb andalso ta=tb
  |   _ => false
```

We then write a visitor function that concatenates its argument lists (if any), and then inserts the node being visited at the head of the resulting list. This ordering corresponds to the topological sort ordering because a node is added to the beginning of the ordering after all of its out-edges are traversed. We can sort a graph with `depthFirstSearch` by passing the `eqList` function on lists and the visitor function.

2.3 Propagation

In self-adjusting computation, after the programmer executes a program, she can change the input to the program and update the output by performing change propagation. The programmer can repeat this change-and-propagate process with different input changes as many times as desired. Figure 4 shows an example that uses `depthFirstSearch` to perform a topological sort. The example assumes an implementation of a library, `SAIL`, that supplies primitives for self-adjusting computation and a `Graph` library that supplies functions for constructing graphs from a file, finding a node, making an edge starting at a particular node, etc.

The `test` function first constructs a graph from a file and then computes its topological sort using `depthFirstSearch`. The `DFSVisitSort` function from the `Graph` library, 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 a new edge from node i to node j as specified by its arguments. To insert the new edge, the function first gets a modifiable for inserting the edge at i and then changes the modifiable to point to node j. The function then performs change-propagation to update the result. For example, after sorting the graph in Figure 3, we can insert the edge from `H` to `C` and update the topological-sort by performing change propagation.

2.4 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 3 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 the thick edges. 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 5.5 after the change-propagation algorithm has been described, we assume that each node has constant out-degree.

Theorem 2.1 (DFS Response Time). *Disregarding the operations performed by the visitor, 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.

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.*

2.5 Implementation

We present techniques for implementing the library efficiently in Section 5. A prototype implementation of the library is available on the web page of the first author.

3. The Language

In this section, we present our Self-Adjusting Imperative Language SAIL. 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 (Felleisen and Hieb 1992). 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 SAIL 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

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

Figure 5. Syntax

$\lambda x.\, e$, pairs of values (v_1, v_2), and injections $\mathtt{inl}\ v$ and $\mathtt{inr}\ v$ 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 ($\mathtt{mod}\ v$), scoped *read*-operations $\mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ e$ that bind the value stored at the location given by v to variable x and execute e in the scope of x, *write*-operations $\mathtt{write}\ v_1 \leftarrow v_2$ that store v_2 into the location given by v_1, *let*-bindings $\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2$, projections from pairs ($\mathtt{fst}\, v$ and $\mathtt{snd}\, v$), and case analysis on sums ($\mathtt{case}\, v\, \mathtt{of}\, \mathtt{inl}\, x_1 \Rightarrow e_1 \mid \mathtt{inr}\, x_2 \Rightarrow e_2$). The form $\mathtt{memo}\ e$ 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*. The syntax of traces is as follows:

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

Traces can be empty (ε), combine two sub-traces obtained by evaluating the two sub-terms of a *let*-form ($\mathtt{let}\ T_1\ T_2$), or record the allocation of a new modifiable reference l that was initialized to v ($\mathtt{mod}\ l \leftarrow v$). A trace of the form $\mathtt{read}_{\,l \to x = v.e}\ T$ 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 $\mathtt{write}\ l \leftarrow v$ records that an existing location l's contents have been updated to contain the new value v. The only difference between the traces $\mathtt{write}\ l \leftarrow v$ and $\mathtt{mod}\ l \leftarrow v$ 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 $\mathtt{mod}\ l \leftarrow v$ within the trace T (formal definition elided).

3.3 Stores

As mentioned in Section 1, 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 the logical relation we formulate for reasoning about consistency. The rules for deriving evaluation judgments are shown in Figure 6.

The rule for $\mathtt{memo}$ has a premise of the form $\sigma, T \curvearrowright^k \sigma', T'$. This is a *change propagation judgment* and should be read as:

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

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

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

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

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

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

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

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

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

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

Figure 6. Evaluation Rules

"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 7. Memoization is modeled (Figure 6) 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 $\mathbf{value}$, $\mathbf{primop}$). 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 $\mathbf{apply}$). A $\mathtt{let}$-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 $\mathtt{let}$-constructor for traces) of the two sub-traces (rule $\mathbf{let}$). Evaluating $\mathtt{mod}\ v$ picks a location l, stores v at l, and returns l. This action (including l and v) is recorded in the trace (rule $\mathbf{mod}$). A $\mathtt{read}$-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 $\mathtt{read}$-operation—including location, bound variable, value read, and body—is recorded in the trace (rule $\mathbf{read}$). A $\mathtt{write}$-operation modifies the store by associating the location to be written with the new value. The result of a $\mathtt{write}$ is unit. Both value and location are recorded in the trace (rule $\mathbf{write}$).

Evaluation of a $\mathtt{memo}$-expression is non-deterministic. When evaluating an expression $\mathtt{memo}\, e$ in a store σ, we can either reuse an evaluation of e in some arbitrary ("previous") store σ_0—not necessarily the same as the current store σ—provided that the evaluation can be adjusted to the current store via change propagation, or we can evaluate e from scratch in the current store σ. The correspond-

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

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

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

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

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

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

Figure 7. Change Propagation Rules

ing evaluation rules, $\textbf{memo/hit}$ and $\textbf{memo/miss}$ respectively, may be written as follows:

$$\frac{\sigma_0, e \Downarrow v, \sigma_0', T_0 \qquad \sigma, T_0 \curvearrowright \sigma', T}{\sigma, \texttt{memo}\, e \Downarrow v, \sigma', T}\ \textbf{(memo/hit)}$$

$$\frac{\sigma, e \Downarrow v, \sigma', T}{\sigma, \texttt{memo}\, e \Downarrow v, \sigma', T}\ \textbf{(memo/miss)}$$

Our evaluation rule for `memo` (Figure 6) does not distinguish between memo hits and memo misses. The high degree of freedom in the choice of σ_0 makes a memo miss a special case of a memo hit: in the $\textbf{memo}$-rule, to simulate a memo miss we pick $\sigma_0 = \sigma$. If evaluation of e in σ produces a trace T, then change propagation of trace T in store σ does nothing (i.e., yields the same σ and T). Hence, picking $\sigma_0 = \sigma$ captures the essence of the memo miss—evaluation proceeds directly in the current store σ, not some other "previous" store σ_0.

The rules $\textbf{fst}$ and $\textbf{snd}$ are the standard projection rules. Projections are pure and, therefore, add nothing to the trace. Similarly, rules $\textbf{case/inl}$ and $\textbf{case/inr}$ 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.

The step counts in each of the evaluation rules are entirely straightforward—we simply count each operational step as we would in a small-step operational semantics. The one exception is the $\textbf{memo}$ rule: notice that according to the rule, `memo` v would evaluate to v in zero steps (since evaluation of v in σ_0 and change propagation of the resulting empty trace would both take zero steps). Intuitively, this reflects a completely arbitrary decision on our part to treat `memo` as a coercion (i.e., a zero-step operation) rather than as an actual operational step. Treating `memo` as a one-step operation would work just as well, though our proofs would have to be adjusted accordingly.

Change-propagation rules. Rule $\textbf{empty}$ is the base case and deals with the empty trace. The $\textbf{mod}$- and $\textbf{write}$-rules re-execute 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 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

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

Figure 8. Store Reachability Relation

$\textbf{let}$-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 $\textbf{read/noch.}$). 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 $\textbf{read/ch.}$).

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 the $\textbf{memo}$ evaluation rule.

Discussion. Our rules are given in a non-deterministic, declarative style. For example, the $\textbf{mod}$-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 $\textbf{let}$-rules.) Furthermore, allocated locations must not be reachable from the initial expression (see Section 3.5).

As in our previous work (Acar et al. 2007b), 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.

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. (2005) 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 8) 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 $\sigma, e \Downarrow^k_{\mathrm{ok}} v, \sigma', T$ as follows.

Definition 3.1 (Valid Evaluation).

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

4. Consistency via Logical Relations

In this section, we prove that the semantics of SAIL 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.

4.1 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 4.1 (Contextual Equivalence).
Let $\Gamma = FV(e_1) \cup FV(e_2)$.

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

$$\Gamma \vdash e_1 \approx^{ctx} e_2 \overset{\mathrm{def}}{=} \Gamma \vdash e_1 \prec^{ctx} e_2 \ \wedge \ \Gamma \vdash e_2 \prec^{ctx} e_1$$

4.2 Proving Consistency

Having defined contextual equivalence, we can be more precise about what we mean by consistency: 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 4.5), and that the latter implies that e is contextually equivalent to itself (Theorem 4.6).

Logical relations specify relations on terms, typically via structural induction on the syntax of types. (Since SAIL is untyped, we will define a logical relation via induction on (available) steps as discussed below.) Thus, for instance, logically related functions take logically related arguments to related results, while logically related pairs consist of components that are related pairwise. Two expressions are logically related if either they both diverge, or they both terminate and yield related values. For any logical relation, one must first prove the so-called Fundamental Property of the logical relation (also called the Basic Lemma) which says that any (well-typed) term is related to itself. If the logical relation is intended to be used for contextual equivalence, the next step is to show that if two terms are logically related, then they are contextually equivalent, which typically follows from the Fundamental Property. Since our logical relation is intended to be used to prove consistency, we will show that any term e that is logically related to itself—by the Fundamental Property of our logical relation, this is true of every e—is contextually equivalent to itself (i.e., $e \approx^{ctx} e$).

The two sources of non-determinism in SAIL 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. 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:

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

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. To show that change propagation does, in fact, bring the two executions into sync, we prove that each memo hit can be replaced by a regular evaluation (Section 4.7).

Our logical relation for consistency of SAIL is based on the step-indexed logical relations for purely functional languages by Appel and McAllester (2001) and Ahmed (2006). 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—e.g., Appel and McAllester (2001); Ahmed et al. (2005); Ahmed (2006)—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 those 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 (1) these graphs are isomorphic, and (2) 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.)

4.3 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 (i.e., $FL(v_1)$, $FL(v_2)$). We refer to the locations in ψ^1 and ψ^2 as the *roots* of v_1 and v_2, respectively.

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

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

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

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

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

Figure 9. Logical Relation

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

Figure 10. Join Local Store Descriptions

The definition of the value relation $\mathcal{V}$ is given in Figure 9. The value $(\,)$ is related to itself for any number of steps. Clearly, no locations appear as subexpressions 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 10). 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.

The left (right) injections into a sum $\texttt{inl}\,v_1$ and $\texttt{inl}\,v_2$ ($\texttt{inr}\,v_1$ and $\texttt{inr}\,v_2$) 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., $(k, \psi, v_1, v_2) \in \mathcal{V}$).

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

Figure 11. Related Stores

Since functions are suspended computations, their relatedness is defined in terms of the relatedness of computations (Section 4.5). 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 (despite the fact that it is *not* indexed by types). Intuitively, $j < k$ suffices because beta-reduction consumes a step.

Notice that the step-indexed technique of defining a logical relation yields not only a specification of the relation, but also guarantees the existence of the relation by making its well-foundedness explicit.

A crucial property of the relation $\mathcal{V}$ is that it is closed under decreasing step index—intuitively, if v_1 "looks like" v_2 for upto k steps, then they should look alike for fewer steps.

Lemma 4.2 (Downward Closed).
If $(k, \psi, v_1, v_2) \in \mathcal{V}$ and $j \leq k$, then $(j, \psi, v_1, v_2) \in \mathcal{V}$.

4.4 Related Stores

The store satisfaction relation $\sigma_1, \sigma_2 :_k \psi \rightsquigarrow \mathcal{S}$ (see Figure 11) 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.[1] 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

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

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.

4.5 Related Computations

The computation relation $\mathcal{C}$ (see Figure 9) specifies when two closed terms e_1 and e_2 (with beliefs ψ, again corresponding to *at least* the locations appearing as subexpressions 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 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 of (σ_1, e_1) (where locations allocated during evaluation are disjoint from those initially reachable in $\mathcal{S}^1$) results in (v_1, σ_1', T_1) in $j < k$ steps, then given any valid evaluation $\sigma_2, e_2 \Downarrow v_2, \sigma_2', T_2$ (which may take any number of steps), 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 can add to them in a minimal way as the two evaluations progress.

4.6 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.

4.7 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 4.5). Unfortunately, in the case of $\texttt{memo}\ e$, we cannot directly appeal to the induction hypothesis. To see why, consider the special case of the closed term $\texttt{memo}\ e$. We must show

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

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

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

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

Figure 12. Logical Predicate for Memo Elimination

$(k, \{\}, \texttt{memo}\ e, \texttt{memo}\ e) \in \mathcal{C}$. 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 σ_{01} followed by change propagation yields the same results as a from-scratch run in the store σ_1.

To prove that each memo hit can be replaced by a regular evaluation (Lemma 4.4), we define a *logical predicate* (i.e., a unary logical relation) for memo elimination. Figure 12 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 zero steps, producing the empty trace, which means that change propagation takes zero 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 (see Figure 12).

Lemma 4.3 (Fundamental Property of Logical Predicate for Memo Elim). *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 4.4 (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* $\sigma, e \Downarrow^{\leq j_1 + j_2} v, \sigma', T'$*.*

4.8 Consistency

For lack of space we have omitted proof details here. Detailed proofs of all lemmas can be found in our extended technical report (Acar et al. 2007a).

Theorem 4.5 (Fundamental Property of Logical Relation for Consistency). *If* $\Gamma = FV(e)$*, then* $\Gamma \vdash e \approx e$*.*

Proof sketch: By induction on the structure of e. As explained above (Section 4.7), in the memo case we use Lemma 4.4 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 4.6 (Consistency). *If* $\Gamma = FV(e)$*, then* $\Gamma \vdash e \approx^{ctx} e$*.*

Let us write $\Downarrow^k_\emptyset$ 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 program*. Since memo elimination (Lemma 4.4) 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 4.7 (Complete Memo Elimination). *Let* e *be a closed term, possibly with free locations. If* $\sigma, e \Downarrow^k v, \sigma', T$*, then* $\sigma, e \Downarrow^{\leq k}_\emptyset v, \sigma', T'$*.*

5. Implementation

We describe data structures and algorithms for implementing SAIL.

5.1 Data Structures

We use order-maintenance, searchable ordered-sets, and standard priority-queue data structures.

Order Maintenance (Time Stamps). An *order-maintenance* data structure maintains a set of *time-stamps* while supporting all of the following operations in constant time: insert a newly created time-stamp after another, delete a time stamp, and compare two time stamps (Dietz and Sleator 1987).

Searchable Time-Ordered Sets. A time-ordered set data structure that supports the following operations.

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

If a data structure contains no more than one element with a given time-stamp, then we can support all operations except for `build` in logarithmic time (in the size of the set) by using a balanced binary search tree keyed by the time-stamps. If the size of the set is bounded by a constant, then we can support all operations in constant time by using a simple representation that keeps all elements in a list.

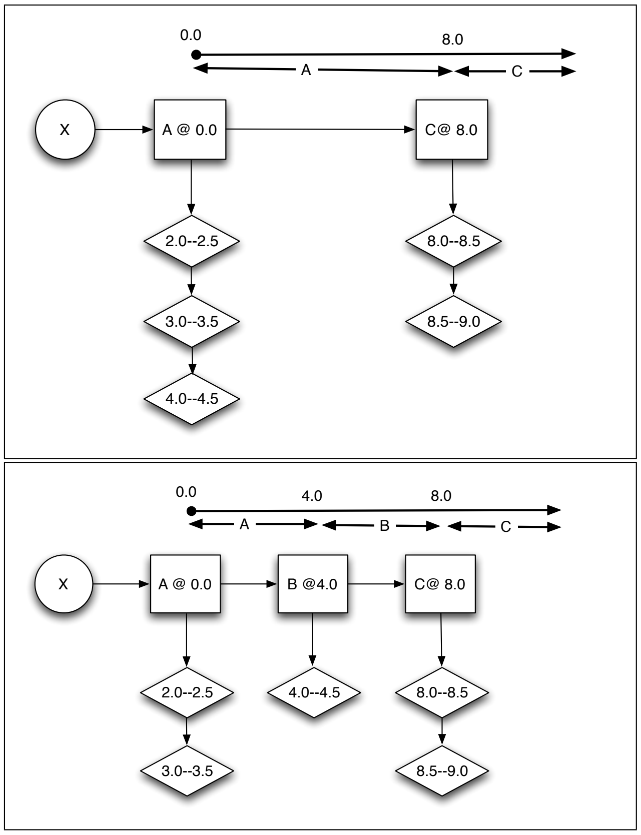

Figure 13. A modifiable, its writes and reads before (top) and after performing a write (bottom).

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. During evaluation and change propagation, we *advance* the time by inserting a new time stamp t immediately after `current-time` and setting the current time to t. In addition to the `current-time`, we also maintain a time stamp called *end-of-memo* for memoization purposes. For change propagation, we maintain a global priority queue that contains the *affected readers* prioritized by their start time (we define readers more precisely below).

Modifiable References. We represent a modifiable reference as a triple consisting of a *version-set*, a *reader-set*, and an equality function. The version-set and the reader-set are both instances of searchable, time-ordered sets. The version set contains all the different contents of the modifiable over time—that is, it contains pairs (v, t) consisting of a value v and a time stamp t. The reader set of a modifiable l contains all the read operations whose source is l. More precisely, 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.

`mod eq`: Create an empty version-set and an empty reader-set. Return a pointer to the triple consisting of the equality function `eq`, the version-set, and the reader-set. Since pointers in ML are equality types, so are modifiables—they can be compared by using ML' s "equal" operator.

`read` l f: Identify the version (v, t_v) of the modifiable being read l that comes immediately before `current-time` by performing a combination of `find` and `prev` operations on the version set. Advance time to a new time stamp t_s. Apply the body of the read f to v. When f returns, advance time again to a new time-stamp 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` l v: Advance time to a new t_w. Create a new version with the value v being written at time t_w. Insert it into the modifiable l being written. Since creating a new version for l can change the value that further reads of l may access, it can affect the readers whose start time comes after t_w but before the next version. To identify the affected readers, we check first that the value v being written is different than that of the previous version by using the equality test of l; if not, then no readers are affected. Otherwise, we find the readers that come at or after t_w by repeatedly performing `find` operations on the reader set of l; we stop when we find a reader that comes after 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` l: Identify the version (v, t) of the dereferenced modifiable l at the `current-time` by using a `find` operation and return v.

`change` l v: Identify the earliest version (v', t) of the changed modifiable l (at the beginning of time) by using a `find` operation, change the value of this version to v. If v is equal to v', then the change does not affect the readers of l. Otherwise, inserts all the readers of the initial version into the priority queue. The readers can be found by finding the next version (if any) and inserting all the readers between the two versions.

Figure 13 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 a 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 become affected and will be inserted into the priority queue.

Memoization and Change Propagation. Figure 14 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` function takes a start and an end time-stamp, t_s and t_e respectively, and undoes the computation between t_s and t_e by deleting all the 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, we drop it from both its reader set and the queue (if it was inserted into the priority queue). To delete a memo

```
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
      $R$ $\leftarrow$ $\{r \mid (t_1, t_2, f)$ 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 14. Pseudo code for `undo`, `memo`, and `propagate`.

entry that starts at t, we remove it from the memo table. Deleting a version is more complicated because it can affect the reads that come after it by changing the value that they read. To delete a version (v, t) of a modifiable l at time t, we first identify the time t' of the earliest version of l 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 a different value than they did before, these reads are affected by the deletion of the version.

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 on the computation being re-used, 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 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, remember the `end-of-memo` in a temporary variable, and run the body of the reader. After the body returns, we undo the computation between the current time and t_e and restore `end-of-memo`.

5.3 Relationship to the Semantics

A direct implementation of the semantics of SAIL (Section 3) is not efficient because change propagation relies on a complete traversal of the trace 1) to find the affected readers, and 2) to find the version of a modifiable at a given time during the computation and update all versions correctly. To find the versions and the affected readers quickly, the implementation maintains the version-set and the readers-set of each modifiable in a searchable time-ordered set data structure. By using these data structures and the `undo` function, the implementation avoids a complete traversal of the trace during change propagation.

The semantics of SAIL does not specify how to find memoized computations for re-use. In our implementation, we remember the results and the time frames of memoized computations in a memo table and re-use them when possible. For a memoized computation to be re-usable, we require its time-frame to fall within the interval defined by `current-time` and `end-of-memo`. This ensures that when a memoized computation is re-used, the write operations performed by the computation are available in the current store. When we re-use a memoized computation, we delete the computations between the `current-time` and the beginning of the memoized computation. This guarantees that any computation is re-used at most once (by deleting all other computations that may contain it) and updates the versions of modifiables.

The semantics of SAIL uses term equality to determine whether a reader is affected or not. Since in ML we do not have access to such equality checks, we rely on user-provided equality tests. Since modifiables are equality types, the user can use ML's "equals" operator for comparing them.

5.4 Asymptotic Complexity

We analyze the asymptotic complexity of self-adjusting computation primitives. 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 (Acar et al. 2007a).

Theorem 5.1 (Overhead). *All self-adjusting computation primitives can be supported in expected 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. Some of the affected readers 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*

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

time.

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.

5.5 Complexity of Depth First Search

We prove the theorems from Section 2 for DFS and topological-sorting. Both theorems use the fact that the DFS algorithm shown in Figure 2 reads from and writes to each modifiable at most once, if the visitor function does the the same. Since initializing a graph requires writing to each modifiable at most once, an application that constructs a graph and then performs a DFS with a single-read and single-write visitor reads from each modifiable once and writes to each modifiable at most twice.

Theorem 5.4 (DFS). *Disregarding read operations performed by the visitor function and the reads of the values returned 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 read of the edge (u, v) performed when visiting u. There are a few cases to consider. If v has been visited, then v will not be visited again and change propagation will complete. If v has not been visited, then it will be visited now and the algorithm will start exploring out from v by traversing its out-edges. Since all of these out-edge traversals will be writing their results into newly allocated destinations, none of these visits will cause a memo match. Since each visited node now has a different path to the root of the DFS tree that passes through the new edge (u, v), each node visited during this exploration process is affected. Since each visit takes constant time, this will require a total of $O(m)$ time. After the algorithm completes the exploration of the affected nodes, it will return to v and then to u. From this point on, there will be no other executed reads and change propagation will complete. Since the only read that is ever inserted into the queue is the one that corresponds to the edge (u, v), the queue size will not exceed one. By Theorem 5.2, the total time for change propagation is $O(m)$. The case for deletions is symmetric. □

We show that the same bound holds for topological sort, which is an application of DFS. For computing the topological sort of a graph with DFS, we use a visitor function that takes as arguments the topological sorts of the subgraph originating at each neighbor of a node u, concatenates them and adds u to the head of the resulting list and returns that list. These operations can be performed in constant time by writing to the tails of the lists involved. Since a modifiable ceases to be at a tail position after a concatenation with a non-empty list, each modifiable in the output list is written at most once by the visitor function. Including the initialization, the total number of writes to each modifiable is bounded by two.

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 change propagation after inserting an edge (u, v). Since the visitor function takes constant time, the traversal of the affected nodes takes $O(m)$ time. After the traversal of the affected nodes completes, `depthFirstSearch` will return a result list that starts with the node u. Since this list is equal to the list that is returned in the previous execution based on the equality tests on modifiable lists (Section 2), it will cause no more reads to be re-executed, and change propagation completes. □

6. Related Work

The problem of enabling computations to respond to changes automatically has been studied extensively. Most of the early work took place under the title of *incremental computation*. Here we review the previously proposed techniques that are based on dependence graphs and memoization and refer the reader to the bibliography of Ramalingam and Reps (1993) for other approaches such as those based on partial evaluation, e.g., Field and Teitelbaum (1990); Sundaresh and Hudak (1991).

Dependence-graph techniques record the dependences between data in a computation, so that a change-propagation algorithm can update the computation when the input is changed. Demers, Reps, and Teitelbaum (1981) and Reps (1982) introduced the idea of *static dependence graphs* and presented a change-propagation algorithm for them. The main limitation of static dependence graphs is that they do not permit the change-propagation algorithm to update the dependence structure. This significantly restricts the types of computations to which static-dependence graphs can be applied. For example, the INC language (Yellin and Strom 1991), which uses static dependence graphs for incremental updates, does not permit recursion. To address this limitation, Acar, Blelloch, and Harper (2006c) proposed *dynamic dependence graphs (or DDGs)*, presented language-techniques for constructing DDGs as programs execute, and showed that change-propagation can update the dependence structure as well as the output of the computation efficiently. The approach makes it possible to transform purely functional programs into self-adjusting programs that can respond to changes to its data automatically. Carlsson (2002) gave an implementation of the approach in the Haskell language. Further research on DDGs showed that, in some cases, they can support incremental updates as efficiently as special-purpose algorithms (Acar et al. 2006c, 2004).

Another approach to incremental computation is based on memoization, where we remember function calls and reuse them when possible (Bellman 1957; McCarthy 1963; Michie 1968). Pugh (1988) and Pugh and Teitelbaum (1989) were the first to apply memoization (also called function caching) to incremental computation. One motivation behind their work was the lack of a general-purpose technique for incremental computation—static-dependence-graph techniques that existed then applied only to certain computations (Pugh 1988). Since Pugh and Teitelbaum's work, other researchers investigated applications of various kinds of memoization to incremental computation (Abadi et al. 1996; Liu et al. 1998; Heydon et al. 2000; Acar et al. 2003).

Until recently dependence-graph based techniques and memoization were treated as two independent approaches to incremental computation. Recent work (Acar et al. 2006b) showed that there is, in fact, an interesting duality between DDGs and memoization in the way that they provide for result re-use and presented techniques for combining them. Other work (Acar et al. 2007b) presented a semantics for the combination and proved that change propagation is consistent with respect to a standard purely functional semantics. The work on this paper builds on these findings. Initial experimental results based on the combination of DDGs and memoization show the combination to be effective in practice for a reasonably broad range of applications (Acar et al. 2006b).

Self-adjusting computation based on DDGs and memoization has recently been applied to other problems. Shankar and Bodik (2007) gave an implementation of the approach in the Java language that targets invariant checking. They show that the approach is effective in speeding up run-time invariant checks significantly compared to non-incremental approaches. Other applications of self-adjusting computation include motion simulation (Acar et al. 2006d), hardware-software codesign (Santambrogio et al. 2007), and machine learning (Acar et al. 2007c).

7. Conclusions

Self-adjusting computation has been shown to be effective for a reasonably broad range of applications where computation data changes slowly over time. Previously proposed techniques for self-adjusting computation, however, were applicable only in a purely functional setting. In this paper, we introduce an imperative programming model for self-adjusting computation by allowing modifiable references to be written multiple times.

We develop a set of primitives for imperative self-adjusting computation and provide implementation techniques for supporting these primitives. The key idea is to maintain different versions that modifiables take over time and keep track of dependences between versions and their readers. We prove that the approach can be implemented efficiently (essentially with the same efficiency as in the purely functional case) when the number of reads and writes of the modifiables is constant. In the general case, the implementation incurs a logarithmic-time overhead in the number of reads and writes per modifiable. As an example, we consider the depth-first search (DFS) problem on graphs and show that it can be expressed naturally. We show that change propagation requires time proportional to the number of nodes whose paths to the root of the DFS tree changes after insertion/deletion of an edge.

Since imperative self-adjusting programs can write to memory without any restrictions, they can create cyclic data structures making it difficult to prove consistency, i.e., that the proposed techniques respond to changes correctly. To prove consistency, we formulate a syntactic logical relation and show that any two evaluations of an expression e.g., a from-scratch evaluation or change propagation, are contextually equivalent. An interesting property of the logical relation is that it is untyped and is indexed only by the number of steps available for future evaluation. To handle the unobservable effects of non-deterministic memory allocation, our logical relations carry location bijections that pair corresponding locations in the two evaluations.

Remaining challenges include giving an improved implementation and a practical evaluation of the proposed approach, reducing the annotation requirements by simplifying the primitives or developing an automatic transformation from static/ordinary into self-adjusting programs that can track dependences selectively.

References

Martin Abadi, Butler W. Lampson, and Jean-Jacques Levy. Analysis and caching of dependencies. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 83–91, 1996.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

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

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148(2), 2006a. Also in *Proceedings of the ACM-SIGPLAN Workshop on ML. 2005.*

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

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

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

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. Technical Report TR-2007-17, Department of Computer Science, University of Chicago, November 2007a.

Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. In *Proceedings of the 16th Annual European Symposium on Programming (ESOP)*, 2007b.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive bayesian inference. In *Neural Information Systems (NIPS)*, 2007c.

Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*, pages 69–83, 2006.

Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional programming (ICFP)*, pages 78–91, 2005.

Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, September 2001.

Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, pages 86–101, 2005.

Nina Bohr and Lars Birkedal. Relational reasoning for recursive types and references. In *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS)*, 2006.

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

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 105–116, 1981.

P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LISP and Functional Programming*, pages 307–322, June 1990.

Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 311–320, 2000.

Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

Yanhong A. Liu, Scott Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

D. Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, 1968.

Peter W. O'Hearn and Robert D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

Nicholas Pippenger. Pure versus impure lisp. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):223–238, 1997.

Andrew M. Pitts. Reasoning about local variables with operationally-based logical relations. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, 1996.

Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, pages 122–141. Springer-Verlag, 1993.

William Pugh. *Incremental computation via function caching*. PhD thesis, Department of Computer Science, Cornell University, August 1988.

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

G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 502–510, 1993.

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

Marco D Santambrogio, Vincenzo Rana, Seda Ogrenci Memik, Umut A. Acar, and Donatella Sciuto. A novel soc design methodolofy for combined adaptive software descripton and reconfigurable hardware. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2007.

Ajeet Shankar and Rastislav Bodik. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming language Design and Implementation (PLDI)*, 2007.

Kurt Sieber. New steps towards full abstraction for local variables. In *ACM SIGPLAN Workshop on State in Programming Languages*, 1993.

Ian D. B. Stark. *Names and Higher-Order Functions*. Ph. D. dissertation, University of Cambridge, Cambridge, England, December 1994.

R. S. Sundaresh and Paul Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–13, 1991.

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