Self-Adjusting Programming

Umut A. Acar[*] Guy E. Blelloch[†] Matthias Blume[‡] Robert Harper[§]

Abstract

This papers proposes techniques for writing *self-adjusting* programs that can adjust to any change to their data (*e.g.*, inputs, decisions made during the computation) *etc.* We show that the techniques are efficient by considering a number of applications including several sorting algorithms, and the Graham Scan, and the quick hull algorithm for computing convex hulls. We show that the techniques are flexible by showing that self-adjusting programs can be trivially transformed into a kinetic programs that maintain their property as their input move continuously. We show that the techniques are practical by implementing a Standard ML library for kinetic data structures and applying the library to kinetic convex hulls. We show that the kinetic programs written with the library are more than an order of magnitude faster than recomputing from scratch. These results rely on a combination of memoization and dynamic dependence graphs. We show that the combination is sound by presenting a semantics based on abstraction of memoization via an oracle.

1 Introduction

Traditionally, a program is thought as a function that maps static inputs to an output. The inputs are *static* in the sense that they remain the same over time. In many real-world applications, however, computations must interact with a data that changes over time. For example, the input to a compiler evolves over time as the programmer modifies the code. Similarly, the model of the world that a search-and-rescue robot maintains in order to maps its route changes when an obstacle appears on the planned route.

One way to deal with this type of problems is the *ad hoc* approach, where the programmer devises their programs by considering all the possible ways that the data can change. This is how these problems are addressed in the algorithms community. There researchers devise *dynamic algorithms* that can adjust to *discrete* changes to their data such as insertions/deletions. For example, a dynamic convex hull algorithm can maintaint the convex hull of a set of points as the user insert/deletes points into/from the input. To deal with motion, researchers devise *kinetic algorithms (or data structures)* that maintain propoerties of moving objects. For example, a kinetic convex-hull algorithm maintains the convex hull of a set of moving points.

The advantage to the *ad hoc* approach is that the resulting algorithms can be very fast, even optimal. The disadvantage is that these algorithms are not composable (imagine composing two hash tables)! Furthermore these problems can be very difficult to solve, even for problems that are simple when in the static case (when inputs don't change or move). As an example, consider the dynamic and kinetic convex hulls. These problems has been studied for more than two decades now [34, 33, 12, 14].

Another way to address these problems is to come up with general-purpose techniques and programming abstractions based on these techniques. In the programming-languages community, researchers developed techniques for these problems in the context of *incremental computation.* Previously proposed techniques were based on static dependence graphs [17, 39, 26, 45], memoization [36, 37, 31, 30], and partial evaluation [20, 19, 42]. Recent work on dynamic dependence graphs generalized dependence-graphs based approaches so that they can be applied to any purely functional program [4, 6].

Unfortunately, none of these techniques are capable of adjusting computations to general changes efficiently. For example, for incremental list sorting with memoization, the best known bound is linear [30]. Dynamic dependence graphs can achieve efficient updates for certain kinds of computations such as certain parallel computations [6, 7] and for certain kinds of changes. For example, with the quick sort algorithm, dynamic dependence graphs can achieve logarithmic time updates for insertions/deletions at the end of the input list [4], but for most changes, they are no better than rerunning from scratch and require $\Theta(n \log n)$ time. Especially when a program involves heavy use of composition, these techniques quickly degenerate into from-scratch execution.

This paper addresses this long-standing problem by devising techniques for writing programs that *self-adjust* to changes. The techniques enable transforming an ordinary program that assumes that its data is static to a program that can self-adjust to any change to its input. The key properties of the techniques are that they are general purpose and efficient. They apply to any purely functional program, and enable programs to adjust to any change to their data whatsoever. The techniques yield program that are efficient even when compared to dynamic and kinetic algorithms developed in the algorithms community both in theory and in practice.

The contributions of this paper are the following:

1. A technique for combining dynamic dependence graphs and memoization.

2. Soundness proof of the combination.

3. Applications to several dynamic problems including sorting, and convex hull under insertions/deletions.

4. Applications to kinetic data structures and experimental results.

Key to the effectiveness of the technique is a combination of dynamic dependence graphs (DDGs) [4] and memo-

[*]`umut@tti-c.org`. Toyota Technological Institute, Chicago, IL.
[†]`guyb@cs.cmu.edu`. Carnegie Mellon University, Pittsburgh, PA.
[‡]`blume@tti-c.org`. Toyota Technological Institute, Chicago, IL.
[§]`rwh@cs.cmu.edu`. Carnegie Mellon University, Pittsburgh, PA.

ization [37, 5]. In our previous papers, we conjectured that DDGs and memoizatino can be combined to obtain a general purpose technique, and effective technique for adjusting computations to changes. This paper proves this conjecture.

This combination of DDGs and memoization is highly nontrivial. This is because of the interaction between memoization, which requires purely functional code, and the implementation of DDGS, which critically rely on side effects. We prove that the combination is correct by presenting a semantics for evaluation and change propagation based on an abstract *oracle* that models memoization. In this approach, an evaluation returns a trace that represents the relationships between the computation data and evaluated expressions. During the evaluation, every function application consults an oracle which can either return a computation as a result or can return no results. If the oracle returns a computation, then it is also required to return a set of changes that characterize the differences between the evaluation store of the returned computation and the current store. If at a function application, the oracle returns no results, then the function application is evaluated as usual. If the oracle returns a computation, then change propagation is applied to the returned computation to adjust it to the current evaluation store and the resulting computation is re-used in place of the function application. Based on this semantics, we show that our techniques are sound by showing that they yield the same result and store and as a from-scratch evaluation where no result re-use and no change-propagation takes place.

To demonstrate the effectiveness of the technique, we consider a number of applications including combining values in a list (*a.k.a.*, “reduce” operation), quick sort and merge sort algorithms, the Graham's Scan [22] and quick hull algorithms [11] for computing convex hulls, the tree-contraction algorithm of Miller and Reif [32]. For all these algorithms, we state bounds that are within an expected constant factor of the best bounds obtained in the algorithms community. To appreciate the complexity of these problems, it should suffice to look at any one of these papers on dynamic convex hulls [34, 33, 14], or Jacob's thesis [27] which solely addresses this problem, and the papers on dynamic trees [40, 41, 15, 38, 25, 44, 9, 21, 10, 43].

To demonstrate the practical effectiveness of the techniques, and their flexibility, we consider kinetic data structures. We describe a library that makes it nearly trivial to transform self-adjusting programs into kinetic programs by making small changes to the code. We apply the library to kinetic convex hull problems and show experimental results. The experiments show that the kinetic programs obtained using our techniques adjust to continuous changes by nearly a factor of 50 faster than computing from-scratch. For discrete changes (insertions/deletions), we show elsewhere that the techniques are two orders of magnitude faster than recomputing from scratch. The techniques are also compared to special-purpose dynamic algorithms elsewhere [7]. There, we show that the approach performs very well even for sophisticated problems.

2 Overview of the paper

The paper consists of two parts. The first part studies the proposed technique in the context of a Standard ML library. Section 3.1 briefly describes the library, Section **??** shows how to transform the an implementation of the ordinary quick-hull algorithm [11] to a self-adjusting algorithm. Section 3.3 describes the combination of DDGs and memoization in an informal (algorithmic) setting, and Section 3.4 describes the change propagation algorithm. Section **??** describes the library for kinetic data structures and applications to kinetic convex hulls.

The second part of the paper is devoted to proving the soundness of the combination of memoization and DDGs.

```
signature BOXED_VALUE = sig type t = ...  end
structure Box:BOXED_VALUE = struct type t = ...  end

signature COMBINATORS = sig
  type α modref
  type α cc

  val modref :  α cc → α modref
  val write :  (α * α → bool) → α → α cc
  val read :  β modref → (β → α cc) → α cc

  val mkLift :  (α*α → bool) →
                (int list * α) →
                (α modref → β) → β
  val mkLiftCC : ((α*α → bool) * (β*β → bool)) →
                 (int list * α) →
                 (α modref → β cc) → β cc

  val change:  'a modref * 'a -> unit
  val propagate:  unit -> unit
end
```

Figure 1: The interface to the ML library.

3 A Framework for Self-Adjusting Programming

This section presents an informal (algorithmic) overview our approach in the context of an ML library, considers several applications and shows asymptotic bounds for these applications. As a practical example, we consider the quick-hull algorithm for computing convex hulls and show how to make the algorithm self-adjusting under both dynamic changes (insertions/deletions of points), and kinetic changes (continuous motion).

3.1 The Library

Figure 1 shows the interface to the library. To support constant-time equality tests, we rely on tag equality based on boxes [5]. The `BOXED_VALUE` module supplies functions for operating on boxed values. Since boxing is very well understood, we do not discuss it in detail here. The signature supplies the `new` function for creating boxed values, the `indexOf` function for accessing the index (*a.k.a.*, tag or hash) of a boxed value, the `eq` function for comparing two boxed values based on their indices. The `COMBINATORS` module supplies operations for *modifiable references* (*modifiables* for short) and changeable computations. Every execution of a changeable computation of type α `cc` starts with the creation of a fresh modifiable of type α `modref`. The modifiable is written at the end of the computation. For the duration of the execution, the reference never becomes explicit. Instead, it is carried “behind the scenes” in a way that is strongly reminiscent of a monadic computation.

Changeable computations of type α `cc` are constructed using `write`, `read`, and `mkLiftCC` functions. The `modref` function executes a given computation on a freshly generated modifiable and returns that modifiable. The `write` function creates a trivial computation that writes the given value into the underlying modifiable. To avoid unnecessary propagation of changes, old and new values are compared for equality at the time of `write` by using the supplied equality predicate. The `read` combinator takes an existing modifiable reference and a receiver for the value read. The result of the `read` combinator is a changeable computation that reads from the modifiable and applies the receiver on the value read.

Functions `mkLift` and `mkLiftCC` support memoization. They take one and two comparison operations respectively and return lift functions that are used for memoizing expressions. A *lift function* memoizes an function based on its strict and one non-strict argument. It performs a memo look up only on the strict arguments and writes non-strict arguments into modifiable references and passes these modifiables to the memoized function. In case of a memo hit, a lift function adjusts the re-used computation to the values of the non-strict arguments via a change propagation on the DDG of re-used computation. In its simplest form, a lift function takes a list of type `int list` of indices of strict arguments, and a non-strict argument of type α, and a function of type α `modref ->` β (or α `modref ->` β `cc`) and returns value of type β (or β `cc`). A `mkLift` function takes an equality test of type α `*` α `-> bool` for the non-strict argument and yields a lift function. Similarly a `mkLiftCC` function function takes an equality test of type α `*` α `->` `bool` for the non-strict argument, and an equality function of type β `*` β `-> bool` for the underlying value of the changeable computation, and yields a lift function. In general, a lift function can have many non-strict arguments. Due to typing restrictions, these will have to be supported separately. The library therefore contains `mkLift2`, `mkLiftCC2`, `mkLift3`, `mkLiftCC3`, *etc.* to support more than one non-strict argument; these are now shown here for brevity.

For changing the values of modifiables and propagating the changes, the library also support two meta-operations: `change` and `propagate`. It is only safe to use these operations at the top level, not within a program.

3.2 Making an application self-adjusting

Using the ML library, the programmer can transform an ordinary (non-self-adjusting) program into a self-adjusting program in two steps. These step involve applying modifiable references and memoization respectively. The first step involves determining and placing changeable parts of the input data into modifiables and applying the `modref`, `read`, `write` operations to the rest of the code. This transformation is guided by the types: values stored in modifiables can only be accessed by a `read` operation, which must end with a `write`, and which must take place within the context of a `modref`. This transformation is described in more detail in previos work [4].

The second step of the transformation involves applying memoization. To memoize an expression, the programmer determines its strict and non-strict arguments, creates a lift function for the expression, and applies the lift function to the expression. Although the programmer can chose any subset of the free variables to be strict or non-strict, we found that the following *strictness principle* leads to programs whose behavior can be analyzed using high-level analysis techniques based on stability analysis. Consider an expression e, if an argument x appears in x at an argument position, *i.e.*, is only passed as an argument to some other function, then x is non-strict; otherwise x is strict. Intuitively, the arguments that contribute directly to the result of the expressions are considered strict, whereas arguments that contribute to the result through some other function are considered non-strict.

Figure **??** shows the code for ordinary quick hull algorithm (left) and its self-adjusting version (right). The quick hull algorithm find the convex hull of a set of points, *i.e.*, the smallest polygon enclosing the points. Both versions are is based on a `Geo` structure that supplies geometry primitives. The ordinary version is a standard implementation of the quick-hull algorithm [**?**]. It relies on a `List` structure that supports the `filter` and `min` functions for filtering a list and finding the minimum element of a list based on the supplied functions. To make the code for quick hull self-adjusting, we follow the two step transformation process. We first change the input of `qhull` to a modifiable list (`MOD_LIST`) where each tail is placed into a modifiable reference. We then insert `mod`, `read`, and `write` combinators into appropriate places in the code. Next, we apply memoization by considering each function separately. Note that the function `qhull` performs trivial work, so it need not be memoized. For `split`, we note that the `NIL` branch performs trivial work, and therefore need not be memoized. To memoize the `CONS` branch, we determine the free variables of the branch—they are `l, p, q, hull`. By applying the strictness principle, we find that `l` and `hull` are non-strict, and `p` and `q` are strict arguments. Using `mkLiftCC2`, we construct a lift function for lifting these variables, and apply the function to the branch.

The self-adjusting version adjusts to "discrete changes",*i.e.*, insertions and deletions to its input automatically. After running `qhull` with some modifiable list `l`, the list can be changed arbitrarily using the `Comb.change` function and the convex hull can be updated automatically by running `Comb.propagate`. We describe how transform this code into a kinetic program in Section 3.7.

3.3 Combining Memoization and DDGs

As a self-adjusting program executes, it creates a record of the operations on modifiables in the form of a dynamic dependence graph (DDG) and remembers memoized computations. Dynamic dependence graphs have been described in previous work [4]. Here, we give an overview of DDGs and focus on the how DDGs may be combined with memoization.

A dynamic dependence graph consists of nodes and edges. The execution of `modref` adds a node, and an evaluation of `read` adds an edge to the DDG. In a `read`, the node being read becomes the source, and the target of the read (the modifiable that the reader finished by writing to) becomes the target. Each edge is tagged with the *reader*, which is a *closure*, and a *containment hierarchy* is maintained on the edges. A read e is *contained* within another read e' if e was created during the execution of e'.

Containment hierarchy is represented using time-stamps. Each edge in is tagged with a *start time stamp* and a *stop time stamp* corresponding to the execution (time) interval of that edge. Time intervals are then used to identify the containment relationship of reads: a read R_a is

```
fun split (p,q,pts,hull) =
let


  fun splitM (p,q,pts,hull) =
  let
    val l = List.filter
              (Geo.above (p,q)) pts
  in
    case l of
      nil => cons(p,hull)
    | cons _ =>

    let
      val max = List.min
                  (Geo.closer (p,q)) l
      val r = split (max,q,l,hull)
    in
      splitM (p,max,l,r)
    end
  end
in
  splitM (p,q,pts,hull)
end

(* qhull:  Geo.Point.t list ->
 * Geo.Point.t list
 *)
fun qhull l =
  case l of
    nil => nil
  | cons(_,t) =>
    case t of
      nil => nil
    | _ =>
    let
      val m = List.min Geo.toLeft l
      val x = List.min Geo.toRight l
      val hull = cons(x,nil)


    in
      split (m,x,l,hull)
    end
```

```
signature MOD_LIST = sig
  datatype α cell =
      NIL
    | CONS of (α * α cell modref)
  type α t = α cell modref

  val eq:  α Box.t cell * α Box.t cell → bool
end
structure ML:MOD_LIST = struct...end

fun split (p,q,pts,hull) =
let
  val lift = mkLift (ML.eq,ML.eq,ML.eq)

  fun splitM (p,q,pts,hull) =
  let
    val l = ML.filter
              Geo.above (p,q) pts
  in read l (fn cl =>
    case cl of
      ML.NIL => write ML.eq (ML.CONS(p,hull))
    | ML.CONS _ => read hull (fn ch =>
      lift ([Box.i p,Box.i q],cl,ch) (fn (l,hull) =>
      let
        val max = ML.min (Geo.closer (p,q)) l
        val r = modref (read max (fn max =>
                     splitM (max,q,l,hull)))
      in
        read max (fn max=>splitM (p,max,l,r))
      end)))
  end
in read p (fn vp => read q (fn vq =>
  splitM (vp,vq,pts,hull)))
end

(* qhull:  Geo.Point.t ML.t ->
 * Geo.Point.t ML.t
 *)
fun qhull l = modref (read l (fn cl =>
  case cl of
    ML.NIL => write ML.eq ML.NIL
  | ML.CONS(_,t) => read t (fn ct =>
      case ct of
        ML.NIL => write ML.eq ML.NIL
      | _ =>
      let
        val m = ML.min Geo.toLeft l
        val x = ML.min Geo.toRight l
        val hull = modref (read x (fn x' =>
                     write ML.eq (ML.CONS(x',
                     modref (write ML.eq ML.NIL)))))
      in
        split (m,x,l,hull)
      end)))
```

Figure 2: The code for ordinary quick hull (left), and its self-adjusting version.

```
(**
 ** $(V,E) = G$ is an DDG
 ** $\phi$ is the memo finger
 ** currentTime is the current time
 **)
lift (i,a) f =
  if (lookup($f, i,$ currentTime$, \phi$) then
    $(b, (t_1, t_2), m) \leftarrow$ find($f, i,$ currentTime$, \phi$)
    $E' \leftarrow \{e' \in E \mid$ currentTime $\leq t_s(e') \leq t_e(e') \leq t_1\}$
    $V \leftarrow V - \{v \mid \exists v'. (v,v') \in E' \vee (v',v) \in E'\}$
    change $(m, a)$
    propagateUntil $(t_2)$
    currentTime $\leftarrow t_2$
    return b
  else
    $m \leftarrow$ newModifiable ()
    change $(m, a)$
    $t_1 \leftarrow$ currentTime
    $b \leftarrow$ f $(m)$
    $t_2 \leftarrow$ currentTime
    remember $f$ with $i$ yields $(b, (t_1, t_2), m)$
```

Figure 3: The lift operation.

contained in a read R_b if and only if the time interval (t_1, t_2) of R_a lies within the time interval (t_3, t_4) of R_b, $i.e.t_3 < t_1 < t_2 < t_4$. The system The maintains the time stamps in an ordered list and keep track of the current time, written `currentTime`. All time stamps are generated with respect to the `currentTime` by inserting a new time stamp immediately after the `currentTime` and immediately before the time stamp following `currentTime`. When a new time stamp t is created, the `currentTime` is advanced to t. Time stamps are compared based on their order in the list, a time stamps t_1 is less than another time stamp t_2 if and only if t_1 is closer to the head of the list than t_2. To operate on time stamps efficiently in constant time, the time stamps are maintenaned in an order-maintenance data structure [18].

To support memozation, the system maintains a memo table for each memoized expression. The memo table of an expression consists of entries, each of which represents an evaluation of that expressions. Each entry maps the non-strict arguments of the evaluation to a triple consisting of the result, and the time interval for that evaluation, and the modifiables that non-strict arguments are written to. Everytime an memoized expression is evaluated, the system performs a memo look up and takes action based on the outcome. Figure 3 shows the pseudo-code the lift operation for evaluating memoized expressions; for brevity, the only the case with one non-strict argument is considered.

- **No memo match:** The system creates a new modifiable for each non-strict argument and writes the value of the non-strict argument to that modifiable. Then, the memoized expression is evaluated with these modifiables as argument. When the expression completes executing, the system creates a new memo table entry mapping the indices of the non-strict arguments to a triple consisting of the result, the time interval of the evaluation, and the modifaibles created.
- **Memo match:** Let (t_1, t_2) be the time interval of the memo entry found. To re-use the computation, the system deletes all the edges whole time stamps are contained within the interval (`currentTime`$, t_1$) and the vertices adjacent to these edges. Then, the system copies the values of the non-strict arguments to the modifiables stored in the memo table entry matched and run change propagation on the re-used computation by calling `propagateUntil` (t_2).

Memo lookups are different than ordinary memo lookups. They take place with respect to a time stamp, called the *(memo) finger*, denoted ϕ, that the system maintains. The finger is initialized to the first time stamp ever created, and is changed only by the change propgation algorithm. In a memo lookup, the systems seeks an entry that has the same strict arguments as the expression being evaluated (this is checked by comparing the index lists) and that has a time interval (t_1, t_2) that lies within the interval (`currentTime`$, \phi$) defined by the current time and the finger. If such an entryu

Since the values stored in modifiables can be side effected my the `write` operation and by the `change` meta-operation, and since the system uses shallow equality (by comparing indices of boxed values) for memo look ups, `propagateUntil` must be run on the re-used computation. This ensure that the re-used computation is adjusted to the changes.

3.4 Change Propagation

Given a dynamic dependence graph and a set of changed modifiables, the *change-propagation algorithm* updates a self-adjusting computation by propagating changes through the DDG. The change-propagation algorithm in the context of DDGs have been described in previous work [4]. In this section, we extend the algorithm to support memoization.

Figure 4 shows the pseudo-code for `propagate` and `propagateUntil` functions. The `propagateUntil` function is only for internal use. The function `propagate` is implemented by calling `propagateUntil` with the largest time stamp, denoted t_∞. The algorithm maintains a priority queue of edges that are *affected*, *i.e.*, the value of their source has changed. The queue is prioritized on the time stamp of each edge, and is initialized with the out-edges of the changed modifiables. During change propagation, the queue is populated by `write` operations; if the contents of a modifiable is changed as a result of a `write` all reads of that modifiable are inserted into the priority queue. Each iteration of the while loop processes one affected edge by re-evaluating the reader associated with the edge. Before re-evaluating the reader, the algorithm remembers the current memo finger ϕ and sets it to t_2 (the end time-stamp of the edge), and set `currentTime` back to t_1 (the start time of the edge). After the re-evaluation, the algorithm deletes all the edges whose time intervals are within the interval defined by the `currentTime` and t_2, and restores the memo finger.

Setting the finger ϕ to t_2 makes the results from the previous evaluation of the read available for re-use. As described in the Section 3.3, a computation is re-used only if its interval is between the `currentTime` and the finger. Note that since ϕ is initialized to the first time stamp and is only changed by change-propagation, result re-use takes place only during change propagation.

For correctness, it is critical that all the edges and vertices that are not re-used during change-propagation be

```
(**
 ** $\chi$ is the set of changed modifiables
 ** $(V, E) = G$ is an DDG
 ** $\phi$ is the finger
 ** currentTime is the current time
 **)
propagateUntil (t) =
Q := $\bigcup_{v \in \chi}$ outEdges($v$)
while $Q$ is not empty
  $e$ := findMin($Q$)
  $(t_1, t_2) \leftarrow$ timeInterval($e$)
  if $t_1 > t$ then
    return
  else
    $\phi' \leftarrow \phi$
    $\phi \leftarrow t_2$
    currentTime $\leftarrow t_1$
    $v' \leftarrow$ apply(reader($e$), val(src($e$)))
    $V \leftarrow V - \{v | \exists v'. (v, v') \in E' \vee (v', v) \in E'\}$
    $E' \leftarrow \{e' \in E |$ currentTime $< t_s(e') < t_e(e') < t_2\}$
    $E \leftarrow E - E'$
    $Q \leftarrow Q - E'$
    $\phi \leftarrow \phi'$

propagate () =
  propagateUntil ($t_\infty$)
```

Figure 4: The change-propagation algorithm.

deleted. This is ensured by the lift operations and the change-propagation algorithm. The lift operation deletes all the vertices and edges created by the computation between the `currentTime` and the re-used computation. Therefore, when re-execution of an edge completes, all unused edges are between the `currentTime` and the end time stamp of the re-executed edge t_2.

3.5 Asymptotic Complexity of Change Propagation

The key property of the combination and the change propagation algorithm described in Sections 3.3 and 3.4 respectively is that they are efficient and their performance can be determined using algorithmic analysis.

Chief to the efficiency of the techniques is two restrictions on computation re-use. These restrictions are

1. a computation can be re-used only once,
2. computation cannot be re-used out of order with respect to their time stamps.

These restrictions are enforced by re-using results that are only within the time frame defined by the `currentTime` and the memo finger, and by advancing the time to the end of the reused result after change-propagating into it (see Figure 3). These restriction enables re-using result in constant time by using a constant-time order-maintenance data structure [18].

The first author's thesis gives a precise treatment of the combination and the change-propagation algorithm described in Section 3.3 and shows that the techniques can be implemented with constant time overhead [2]. This claim is also verified experimentally in the thesis and elsewhere [2, 3].

Based on the change propagation algorithm, we analyzed a number of applications including combining values in a list, the quick sort and merge sort algorithms, the graham scan algorithm for convex hulls [22], and the tree-contraction algorithm [32] and showed that these algorithms adjust to changes efficiently. These bounds are within a constant factor of the best known bounds. We state the theorems here and refer the interested reader to Acar's thesis for the proofs [2]. All the algorithms are randomized and all expectations are taken over internal randomization unless otherwise stated. These bounds are experimentally verified in other work [2, 7, 3].

Theorem 1 (Self-Adjusting List Combine)
The list combine algorithm self-adjusts to insertions/deletions in expected $O(\log n)$ time.

Theorem 2 (Self-Adjusting Quick Sort)
Quick sort self-adjusts to an insertion/deletion at a uniformly random position in expected $O(\log n)$ time. The expectation is taken over all permutations of the input.

Theorem 3 (Self-Adjusting Merge sort)
Merge sort algorithm self-adjusts to insertions/deletions anywhere in the input in expected $O(\log n)$ time.

Theorem 4 (Self-Adjusting Graham's Scan)
The Graham's Scan algorithm self-adjusts to an insertion/deletion at a uniformly randomly chosen location in expected $O(\log n)$ time.

Theorem 5 (Dynamic Trees)
The tree contraction algorithm adjusts to a single edge insertion/deletion in expected $O(\log n)$ time.

As with the ordinary quick hull algorithm, it is not possible to give tight bounds for the self-adjusting version of quick hull—the algorithm has poor worst case stability. As with the ordinary quick hull algorithm, however, the self-adjusting version performs well in practice.

3.6 Implementation

The full code for the SML library (Section 3.1) is presented elsewhere [3, 2]. Our experiments with this implementation show that its has a constant factor overhead of eight over from-scratch execution, and that change-propagation can yield speed up of up to two orders of magnitude over from-scratch execution [3, 2].

3.7 Kinetic Data Structures

Kinetic algorithms or data structures maintain a combinatorial property of continuously moving objects. For example a kinetic convex hull algorithm maintains the convex hull of a set of points as the points move continously in the plane. Since their invention by Basch, Guibas, and Hershberger [12], kinetic algorithms have been studied extensively [13, 8, 23, 12, 24, 29, 16, 28, 1] maintain properties of moving objects. In this section, we show how a self-adjusting program can be transformed into a kinetic program trivially and show some preliminary experimental results.

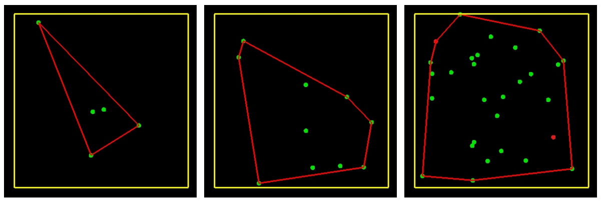

Figure 5: Snapshots of a kinetic simulation.

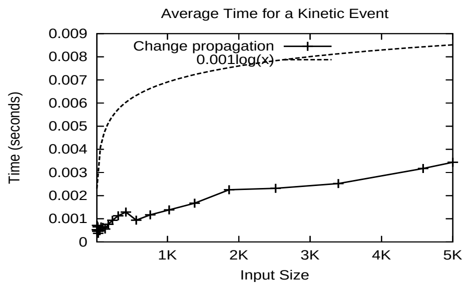

Figure 6: Timings with self-adjusting quick hull.

Kinetic data structures are based on the idea of creating a *certificate* for each predicate computed by a static algorithm. Each certificate is tagged with a *failure time* at which it is known to change value. The certificates generated by a computation can be thought as a proof of the property being computed. To simulate motion, the certificates are placed in a priority queue of an event scheduler. The event scheduler simulates motion in a stepwise fashion. At each step, the scheduler removes the earliest failing certificate from the queue, advances the simulation time to the failure time of the certificate and runs an update algorithm. The *update algorithm* updates the computation according to the new value of the certificate.

The key component of a kinetic algorithm is the update algorithm. In all previous work, update algorithms are designed on a per problem basis [13, 23, 8, 29, 16, 28]. We use the change-propagation algorithm as a general-purpose update algorithm.

Based on this idea we implemented an ML-library for kinetic data structures. Using our library, the programmer can transform a self-adjusting program into a kinetic program in three steps. The first involves replacing each comparison with a library supplied *kinetic comparison*. A kinetic comparison generates a certificate for the comparison. The second step involves applying the library supplied `certify` function to each kinetic comparison. The `certify` function writes the value of the certificate into a modifiable and inserts that modifiable into the event scheduler's queue based on its failure time. The third step involves modifying the self-adjusting program so that the values returned by `certify` functions, which are modifiables, are read. For example, given our `Kinetic` library, and a library `KGeo` of certifying comparisons, the main change to the code for the quick-hull algorithm is to replace Geo.above, Geo.closer ... with (Kinetic.certify o KGeo.above), (Kinetic.certify o KGeo.closer) ... respectively.

To determine the effectiveness of our approach, we performed some experiments with the kinetic quick-hull program. Figure 5 shows snapshots of a simulation with our kinetic quick hull program. All points move inside the box with randomly determined initial velocities and directions. Every time a kinetic event happens, *i.e.*, a certificate fails, the simulation inserts a random set of points and deletes a random set of points. In the example simulation, insertions are four times more likely than deletions. This ability to support discrete and continuous changes simultaneously is unique to our approach.

Figure **??** show the average time per kinetic event with the our kinetic quick-hull program for inputs sizes of up to 5000 points. We measure this by assigning random velocity vector to points uniformly randomly selected within a bounding box and performing a simulation until there are no more kinetic events. A brute-force way to implement kinetic algorithms is to rerun the computation from scratch every time the value of a comparison changes. Figure **??**. In our experiments, we measured the ratio of time for a from-scratch execution to the time for change propagation (this is the speed up obtained by change propagation) to be 40. These timings also include the time spent by the SML/NJ garbage collector which can be as high as 60% of the total time. This is due to known problems in the interaction of SML/NJ garbage collector and impure program (programs with side effects). Note that our techniques heavily rely on side effects—(ordinary reference are used to implement modifiable references).

4 The Language

4.1 Abstract syntax

Types	τ	$::=$	$\texttt{int} \mid \texttt{unit} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid$
			$\tau\ \texttt{mod} \mid \tau_1 \xrightarrow{\mathsf{S}} \tau_2 \mid \tau_1 \xrightarrow{\mathsf{C}} \tau_2$
Values	v	$::=$	$() \mid n \mid x \mid l \mid (v_1, v_2) \mid$
			$\texttt{inl}_{\tau_1+\tau_2} v \mid \texttt{inr}_{\tau_1+\tau_2} v \mid$
			$\texttt{funs}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e_s\ \texttt{end} \mid$
			$\texttt{func}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e_c\ \texttt{end}$
Op.	o	$::=$	$\texttt{not} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots$
Exp.	e	$::=$	$e_s \mid e_c$
St. Exp.	e_s	$::=$	$v \mid o(v_1, \ldots, v_n) \mid \texttt{mod}_\tau\ e_c \mid$
			$\texttt{apply}_{\mathsf{S}}(v_1, v_2) \mid$
			$\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_s'\ \texttt{end} \mid$
			$\texttt{let}\ x_1{\times}x_2\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} \mid$
			$\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\,(x_1{:}\tau_1) \Rightarrow e_s$
			$\mid\ \texttt{inr}\,(x_2{:}\tau_2) \Rightarrow e_s'$
Ch. Exp.	e_c	$::=$	$\texttt{write}_\tau(v) \mid \texttt{apply}_{\mathsf{C}}(v_1, v_2) \mid$
			$\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_c\ \texttt{end} \mid$
			$\texttt{let}\ x_1{\times}x_2\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} \mid$
			$\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c\ \texttt{end} \mid$
			$\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\,(x_1{:}\tau_1) \Rightarrow e_c$
			$\mid\ \texttt{inr}\,(x_2{:}\tau_2) \Rightarrow e_c'$
Program	p	$::=$	e_s

Figure 7: The abstract syntax of SLf.

The abstract syntax of SLf types and terms is given in Figure 7. It is very similar to the language AFL that we have described earlier [4]. We use the meta-variables x, y, and z (and variants) to range over an unspecified set of variables, and the meta-variable l (and variants) to range over a separate, unspecified set of locations—the locations are modifiable references. As before, the syntax of SLf is restricted to "2/3-cps", or "named form", to streamline the presentation of the dynamic semantics.

The types language includes the base types `int` and `unit`, sums and products, the stable function type $\tau_1 \xrightarrow{\mathsf{S}} \tau_2$, the changeable function type $\tau_1 \xrightarrow{\mathsf{C}} \tau_2$, and the type $\tau\ \mathtt{mod}$ of modifiable references of type τ. Extending SLf with recursive types presents no fundamental difficulties, but they are omitted here for the sake of brevity.

Expressions are classified into three categories: values, *stable* expressions, and *changeable* expressions. Values are constants, variables, locations, and the introduction forms for sums, products, and functions. The value of a stable expression is not sensitive to modifications to the inputs, whereas the value of a changeable expression may, directly or indirectly, be affected by them.

The familiar mechanisms of functional programming are embedded in SLf as stable expressions. We have a sequential `let` construct for ordering evaluation, elimination forms for products and sums, applications of *stable functions*, and the creation (including initialization) of new modifiables. The body of a stable function must itself be a stable expression. The expression $\mathtt{mod}_\tau\ e_c$ allocates a new modifiable reference and initializes it by executing the changeable expression e_c. Note that the modifiable itself is stable, even though its contents is subject to change.

Changeable expressions always execute in the context of an enclosing `mod`-expression which provides the implicit target location that every such execution ultimately writes to. The changeable expression $\mathtt{write}_\tau(v)$ writes the value v of type τ into the target. The expression $\mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ e_c\ \mathtt{end}$ binds the contents of the modifiable v to the variable x, then continues evaluation of e_c. A `read` is considered changeable because the contents of the modifiable on which it depends is subject to change. A changeable function is stable as a value, but its body is changeable; correspondingly, the application of a changeable function is a changeable expression. Like in the stable case there are also changeable versions of the sequential `let` and the elimination forms for sums and products.

The static semantics of SLf is shown in Appendix A.

4.2 Dynamic semantics

The evaluation judgements of SLf have one of two forms. The judgement $\sigma, \chi, e \Downarrow^{\mathsf{S}} v, \sigma', \chi', \mathtt{T}_s$ states that evaluation of the stable expression e, relative to the input store σ, yields the value v, the trace $\mathtt{T}_s$, and the updated store σ'. Moreover, it relates *change sets* χ and χ', which are sets of locations: if χ contains those locations of σ relevant to e that potentially have been modified (relative to a hypothetical previous run), then χ' accounts for (at least) all the locations of σ' that have changed relative to the same previous run.

Similarly, the judgement $\sigma, \chi, l \leftarrow e \Downarrow^{\mathsf{C}} \sigma', \chi', \mathtt{T}_c$ states that evaluation of the changeable expression e, relative to the input store σ, writes its value to the target l, and yields the trace $\mathtt{T}_c$ and the updated store σ'. The meaning of the change sets is analogous to the stable case.

In the dynamic semantics, a *store*, σ, is a finite function mapping each location in its domain, $\mathtt{dom}(\sigma)$, to either a value v or a "hole" $\square$. The *defined domain*, $\mathtt{gen}(\sigma)$, of σ consists of those locations in $\mathtt{dom}(\sigma)$ not mapped to $\square$ by σ. When a location is created, it is assigned the value $\square$ to reserve that location while its value is being determined. With a store σ, we associate a location typing Λ and write $\sigma : \Lambda$, if the store satisfies the typing Λ.

A *trace* is a finite data structure recording the adaptive aspects of evaluation. Like the expressions whose evaluations they describe, traces come in stable and changeable varieties:

The abstract syntax of traces is given by the following grammar:

$$\begin{array}{lll} \textit{Stable} & \mathtt{T}_s & ::= \ \epsilon \mid \mathtt{mod}_\tau\ l \leftarrow \mathtt{T}_c \mid \diamond\ \mathtt{T}_s \mid \mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_s \\ \textit{Changeable} & \mathtt{T}_c & ::= \ \mathtt{write}_\tau \mid \diamond\ \mathtt{T}_c \mid \mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_c \mid \mathtt{read}_{l \to x.e}\mathtt{T}_c \end{array}$$

Notice that every evaluation step extends the trace, although sometimes just with a dummy item. We use this technical trick to be able to perform proofs by induction on the length of the trace.

A stable trace records the sequence of allocations of modifiables that arise during the evaluation of a stable expression. The trace $\mathtt{mod}_\tau\ l \leftarrow \mathtt{T}_c$ records the allocation of the modifiable, l, its type, τ, and the trace of the initialization code for l. The trace $\mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_s'$ results from evaluation of a `let` expression in stable mode, the first trace resulting from the bound expression, the second from its body.

A changeable trace has one of three forms. A write, $\mathtt{write}_\tau$, records the storage of a value of type τ in the target. A sequence $\mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_c$ records the evaluation of a `let` expression in changeable mode, with $\mathtt{T}_s$ corresponding to the bound stable expression, and $\mathtt{T}_c$ corresponding to its body. A read $\mathtt{read}_{l \to x.e}\mathtt{T}_c$ trace specifies the location read (l), the context of use of its value ($x.e$) and the trace, $\mathtt{T}_c$, of the remainder of evaluation with the scope of that read. This records the dependency of the target on the value of the location read.

The dynamic dependency graphs described in Section **??** may be seen as an efficient representation of traces. Time stamps may be assigned to each read and write operation in the trace in left-to-right order. These correspond to the time stamps in the DDG representation. The containment hierarchy is directly represented by the structure of the trace. Specifically, the trace $\mathtt{T}_c$ (and any read in $\mathtt{T}_c$) is contained within the read trace $\mathtt{read}_{l \to x.e}\mathtt{T}_c$.

Stable Evaluation. The evaluation rules for stable expressions are given in Figure 8. Most of the rules are standard for a store-passing semantics. For example, the `let` rule sequences evaluation of its two expressions, and performs binding by substitution. Less conventionally, it yields a trace consisting of the sequential composition of the traces of its sub-expressions and maintains a change set.

The most interesting rule is the evaluation of $\mathtt{mod}_\tau\ e$. Given a store σ, a fresh location l is allocated and initialized to $\square$ prior to evaluation of e. The sub-expression e is evaluated in changeable mode, with l as the target. Preallocating l ensures that the target of e is not accidentally re-used during evaluation; the static semantics ensures that l cannot be read before its contents is set to some value v.

Each location allocated during the evaluation of a stable expression is recorded in the trace and is written to: If $\sigma, \chi, e \Downarrow^{\mathsf{S}} v, \sigma', \chi', \mathtt{T}_s$, then $\mathtt{dom}(\sigma') = \mathtt{dom}(\sigma) \cup \mathtt{gen}(\mathtt{T}_s)$, and $\mathtt{gen}(\sigma') = \mathtt{gen}(\sigma) \cup \mathtt{gen}(\mathtt{T}_s)$. Furthermore, all locations

$$\frac{}{\sigma,\chi,v\ \Downarrow^{\mathtt{S}}\ v,\sigma,\chi,\varepsilon}\ \textbf{(value)}$$

$$\frac{(v=\mathtt{app}(o,(v_1,\ldots,v_n)))}{\sigma,\chi,o(v_1,\ldots,v_n)\ \Downarrow^{\mathtt{S}}\ v,\sigma,\chi,\varepsilon}\ \textbf{(primitives)}$$

$$\frac{\begin{array}{c}(l\notin\mathtt{dom}(\sigma))\\ \sigma[l\to\Box],\chi,l\leftarrow e\ \Downarrow^{\mathtt{C}}\ \sigma',\chi,\mathtt{T}\end{array}}{\sigma,\chi,\mathtt{mod}_\tau\ e\ \Downarrow^{\mathtt{S}}\ l,\sigma',\chi,\mathtt{mod}_\tau\ l\leftarrow\mathtt{T}}\ \textbf{(mod)}$$

$$\frac{\begin{array}{c}(v_1=\mathtt{fun}_{\mathtt{C}}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end})\\ (e'=[v_1/f,v_2/x]\ e)\\ \sigma,e'\ \uparrow^{\mathtt{S}}\\ \sigma,\chi,e'\ \Downarrow^{\mathtt{S}}\ v,\sigma',\chi',\mathtt{T}_s\end{array}}{\sigma,\chi,\mathtt{apply}_{\mathtt{S}}(v_1,v_2)\ \Downarrow^{\mathtt{S}}\ v,\sigma',\chi',\diamond\ \mathtt{T}'_s}\ \textbf{(apply/miss)}$$

$$\frac{\begin{array}{c}(v_1=\mathtt{fun}_{\mathtt{C}}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end})\\ \sigma,[v_1/f,v_2/x]\ e\ \downarrow^{\mathtt{S}}\ v,\mathtt{T}_s,\chi'\\ \sigma,\mathtt{T}_s,\chi\cup\chi'\ \overset{\mathtt{S}}{\curvearrowright}\ \sigma',\chi'',\mathtt{T}'_s\end{array}}{\sigma,\chi,\mathtt{apply}_{\mathtt{S}}(v_1,v_2)\ \Downarrow^{\mathtt{S}}\ v,\sigma',\chi'',\diamond\ \mathtt{T}'_s}\ \textbf{(apply/hit)}$$

$$\frac{\begin{array}{lll}\sigma,\chi,e_1 & \Downarrow^{\mathtt{S}} & v_1,\sigma_1,\chi',\mathtt{T}_1\\ \sigma_1,\chi',[v_1/x]e_2 & \Downarrow^{\mathtt{S}} & v_2,\sigma_2,\chi'',\mathtt{T}_2\end{array}}{\sigma,\chi,\mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end}\ \Downarrow^{\mathtt{S}}\ v_2,\sigma_2,\chi'',\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2}\ \textbf{(let)}$$

$$\frac{\sigma,\chi,[v_1/x_1,v_2/x_2]e\quad\Downarrow^{\mathtt{S}}\quad v,\sigma',\chi',\mathtt{T}}{\sigma,\chi,\mathtt{let}\,x_1{\times}x_2\,\mathtt{be}\,(v_1,v_2)\,\mathtt{in}\,e\,\mathtt{end}\ \Downarrow^{\mathtt{S}}\ v,\sigma',\chi',\diamond\ \mathtt{T}}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma,\chi,[v/x_1]\ e_1\ \Downarrow^{\mathtt{S}}\ v',\sigma',\chi',\mathtt{T}}{\begin{array}{llll}\sigma,\chi, & \mathtt{case}\ \mathtt{inl}_{\tau_2+v\tau_1} & \mathtt{of} & \Downarrow^{\mathtt{S}}\ v',\sigma',\chi',\diamond\ \mathtt{T}\sigma\\ & \quad\mathtt{inl}\,(x_1{:}\tau_1) & \Rightarrow e_1 & \\ & \quad\mid\ \mathtt{inr}\,(x_2{:}\tau_2) & \Rightarrow e_2 & \end{array}}\ \textbf{(case/inl)}$$

$$\frac{\sigma,\chi,[v/x_2]\ e_2\ \Downarrow^{\mathtt{S}}\ v',\sigma',\chi',\mathtt{T}}{\begin{array}{llll}\sigma,\chi, & \mathtt{case}\ \mathtt{inr}_{\tau_2+v\tau_1} & \mathtt{of} & \Downarrow^{\mathtt{S}}\ v',\sigma',\chi',\diamond\ \mathtt{T}\\ & \quad\mathtt{inl}\,(x_1{:}\tau_1) & \Rightarrow e_1 & \\ & \quad\mid\ \mathtt{inr}\,(x_2{:}\tau_2) & \Rightarrow e_2 & \end{array}}\ \textbf{(case/inr)}$$

$$\frac{}{\sigma,\chi,l\leftarrow\mathtt{write}_\tau(v)\ \Downarrow^{\mathtt{C}}\ \sigma[l\leftarrow v],\chi,\mathtt{write}_\tau}\ \textbf{(write)}$$

$$\frac{\begin{array}{c}(v_1=\mathtt{fun}_{\mathtt{C}}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end})\\ (e'=[v_1/f,v_2/x]\ e)\\ \sigma,e'\ \uparrow^{\mathtt{C}}\\ \sigma,\chi,l\leftarrow e'\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\mathtt{T}_c\end{array}}{\sigma,\chi,l\leftarrow\mathtt{apply}_{\mathtt{C}}(v_1,v_2)\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\diamond\ \mathtt{T}'_c}\ \textbf{(apply/miss)}$$

$$\frac{\begin{array}{c}(v_1=\mathtt{fun}_{\mathtt{C}}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end})\\ \sigma,l\leftarrow[v_1/f,v_2/x]\ e\ \downarrow^{\mathtt{C}}\ \mathtt{T}_c,\chi'\\ \sigma,\chi\cup\chi',l\leftarrow\mathtt{T}_c\ \overset{\mathtt{S}}{\curvearrowright}\ \sigma',\chi'',\mathtt{T}'_c\end{array}}{\sigma,\chi,l\leftarrow\mathtt{apply}_{\mathtt{C}}(v_1,v_2)\ \Downarrow^{\mathtt{C}}\ \sigma',\chi'',\diamond\ \mathtt{T}'_c}\ \textbf{(apply/hit)}$$

$$\frac{\begin{array}{lll}\sigma,\chi,e_1 & \Downarrow^{\mathtt{S}} & v,\sigma_1,\chi',\mathtt{T}_1\\ \sigma_1,\chi,l\leftarrow[v/x]e_2 & \Downarrow^{\mathtt{C}} & \sigma_2,\chi'',\mathtt{T}_2\end{array}}{\sigma,\chi,l\leftarrow\mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end}\ \Downarrow^{\mathtt{C}}\ \sigma_2,\chi'',\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2}\ \textbf{(let)}$$

$$\frac{\sigma,\chi,l\leftarrow[v_1/x_1,v_2/x_2]e\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\mathtt{T}}{\sigma,\chi,l\leftarrow\mathtt{let}\,x_1{\times}x_2\,\mathtt{be}\,(v_1,v_2)\,\mathtt{in}\,e\,\mathtt{end}\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\diamond\ \mathtt{T}}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma,\chi,l'\leftarrow[\sigma(l)/x]\ e\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\mathtt{T}_c}{\sigma,\chi,l'\leftarrow\mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end}\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\mathtt{read}_{l\to x.e}\mathtt{T}_c}\ \textbf{(read)}$$

$$\frac{\sigma,\chi,l\leftarrow[v/x_1]\ e_1\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\mathtt{T}}{\begin{array}{llll}\sigma,\chi,l\leftarrow & \mathtt{case}\ \mathtt{inl}_{\tau_2+v\tau_1} & \mathtt{of} & \Downarrow^{\mathtt{C}}\ \sigma',\chi',\diamond\ \mathtt{T}\\ & \quad\mathtt{inl}\,(x_1{:}\tau_1) & \Rightarrow e_1 & \\ & \quad\mid\ \mathtt{inr}\,(x_2{:}\tau_2) & \Rightarrow e_2 & \end{array}}\ \textbf{(case/inl)}$$

$$\frac{\sigma,\chi,l\leftarrow[v/x_2]\ e_2\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\mathtt{T},}{\begin{array}{llll}\sigma,\chi, & \mathtt{case}\ \mathtt{inr}_{\tau_2+v\tau_1} & \mathtt{of} & \Downarrow^{\mathtt{C}}\ \sigma',\chi',\diamond\ \mathtt{T}\\ & \quad\mathtt{inl}\,(x_1{:}\tau_1) & \Rightarrow e_1 & \\ & \quad\mid\ \mathtt{inr}\,(x_2{:}\tau_2) & \Rightarrow e_2 & \end{array}}\ \textbf{(case/inr)}$$

Figure 8: Evaluation of stable and changeable expressions.

read during evaluation are defined in the store, $\mathtt{dom}(\mathtt{T}_s)\subseteq\mathtt{gen}(\sigma')$.

Changeable Evaluation. The evaluation rules for changeable expressions are also given in Figure 8. The `let` rule is similar to the corresponding rule in stable mode, except that the bound expression, e, is evaluated in stable mode, whereas the body, e', is evaluated in changeable mode. The `read` expression substitutes the binding of location l in the store σ for the variable x in e, and continues evaluation in changeable mode. The read is recorded in the trace, along with the expression that employs the value read. The `write` rule simply assigns its argument to the target. Finally, application of a changeable function passes the target of the caller to the callee, avoiding the need to allocate a fresh target for the callee and a corresponding read to return its value to the caller.

Each location allocated during the evaluation of a changeable expression is recorded in the trace and is written; the destination is also written: If $\sigma,\chi,l\leftarrow e\ \Downarrow^{\mathtt{C}}\ \sigma',\chi',\mathtt{T}_c$, then $\mathtt{dom}(\sigma')=\mathtt{dom}(\sigma)\cup\mathtt{gen}(\mathtt{T}_c)$, and $\mathtt{gen}(\sigma')=\mathtt{gen}(\sigma)\cup\mathtt{gen}(\mathtt{T}_c)\cup\{l\}$. Furthermore, all locations read during evaluation are defined in the store, $\mathtt{dom}(\mathtt{T}_c)\subseteq\mathtt{gen}(\sigma')$.

4.3 The Oracle

Evaluation of function application (both stable and changeable) refers to an *oracle*. The oracle is used to model memoization. Corresponding to a memo-table miss, the oracle can decide to give a negative answer. We write this as $\sigma,e_s\ \uparrow^{\mathtt{S}}$ or $\sigma,l\leftarrow e_c\ \uparrow^{\mathtt{C}}$, respectively. Alternatively, it can report value and trace of an earlier evaluation of a given expression, represented by judgments $\sigma,e_s\ \downarrow^{\mathtt{S}}\ v,\mathtt{T}_s,\chi$ and $\sigma,l\leftarrow e_c\ \downarrow^{\mathtt{C}}\ \mathtt{T}_c,\chi$. The change set χ that is part of a positive answer contains the locations that differ between σ (the store in which we want to evaluate e_s now) and the

	stable	changeable
miss	$\dfrac{}{\sigma, e_s \ \uparrow^{\mathtt{S}}}$	$\dfrac{}{\sigma, l \leftarrow e_c \ \uparrow^{\mathtt{C}}}$
hit	$\dfrac{\begin{array}{c}\sigma', \chi', e_s \ \Downarrow^{\mathtt{S}} \ v, \sigma'', \chi'', \mathtt{T}_s \\ \sigma \sqsupseteq \sigma''\vert_{\mathtt{gen}(\mathtt{T}_s)} \\ \mathtt{dom}(\sigma) \supseteq \mathtt{dom}(\sigma') \\ \chi = \mathtt{change}(\sigma, \sigma')\end{array}}{\sigma, e_s \ \downarrow^{\mathtt{S}} \ v, \mathtt{T}_s, \chi}$	$\dfrac{\begin{array}{c}\sigma', l \leftarrow e_c \ \Downarrow^{\mathtt{S}} \ \sigma'', \mathtt{T}_c \\ \sigma \sqsupseteq \sigma''\vert_{\mathtt{gen}(\mathtt{T}_c)} \\ \mathtt{dom}(\sigma) \supseteq \mathtt{dom}(\sigma') \\ \chi = \mathtt{change}(\sigma, \sigma')\end{array}}{\sigma, l \leftarrow e_c \ \downarrow^{\mathtt{C}} \ \mathtt{T}_c, \chi}$

Figure 9: The oracle.

original store that was in effect when the earlier evaluation was memoized. A positive answer from the oracle still needs to be adjusted to these changes. Change propagation is explained in Section 5.

Notice that the oracle makes the dynamic semantics formally non-deterministic. Lemmas 7 and 8 in Section 6 express the important property that this non-determinism does not matter: change propagation will correctly update the positive answers of the oracle and make everything look as if the oracle always answered negatively. We write $\sigma, \chi, e \Downarrow^{\mathtt{S}}_{\emptyset} v, \sigma', \chi', \mathtt{T}_s$ and $\sigma, \chi, l \leftarrow e \Downarrow^{\mathtt{C}}_{\emptyset} \sigma', \chi', \mathtt{T}_c$ if $\sigma, \chi, e \Downarrow^{\mathtt{S}} v, \sigma', \chi', \mathtt{T}_s$ and $\sigma, \chi, l \leftarrow e \Downarrow^{\mathtt{C}} \sigma', \chi', \mathtt{T}_c$, respectively, can be derived without ever using a **hit** rule. We call an evaluation that does not use **hit** a *clean* evaluation.

5 Change propagation

The change propagation rules are relatively simple: Most rules simply skip ahead. Only when a **read** finds that the location read has changed, it re-runs the changeable computation that is in its, thereby replacing the corresponding part of the trace. As explained earlier, the actual implementation is more efficient than what the rules suggest: By keeping the **read**s that need attention on a priority queue, change propagation can skip ahead arbitrarily far in a single step.

6 Correctness

As we have already pointed out, the operational semantics of SLf is non-deterministic because the evaluation of function calls has the option to execute the call normally or to use and adjust the result of an earlier such call. Of course, it is undesirable for two different evaluation paths to lead to different results. We call memoization and change propagation *incorrect* if this were possible.

Naively we might expect that correctness can be characterized by saying that an arbitrary evaluation should yield results (value, store, and trace) that are identical to those obtained from a clean (**hit**-free) evaluation:

$$\sigma, \emptyset, e \Downarrow^{\mathtt{S}} v, \sigma', \_, \mathtt{T} \quad \Rightarrow \quad \sigma, \emptyset, e \Downarrow^{\mathtt{S}}_{\emptyset} v, \sigma', \_, \mathtt{T}$$

6.1 Structural Equality and Lifting

Unfortunately, we cannot prove this because it is not even generally true. This problem is due to differences in the stores that are involved: re-executing `mod`-expressions will allocate fresh locations, and there is no control over their identity. However, we are not really interested in the identity of locations. We consider a value v_1 in a store σ_1 equal to a value v_2 in a store σ_2 if they are *structurally equal* in the sense of, e.g., Lisp's **EQUAL** function. This works because SLf programs cannot create cyclic memory graphs. Moreover, like memory addresses of cons-cells in Standard ML, SLf programs cannot compare locations for equality and, therefore, cannot observe sharing within data structures.

We formalize the notion of structural equality using the *lift operation* $e \uparrow \sigma$. Intuitively, starting with the locations in e, this recursively replaces all locations l with their respective values $\sigma(l)$. The definition of $e \uparrow \sigma$ is shown in Figure 11. A similar definition for lifting of traces is omitted.

6.2 The Correctness Theorem

Using lift, we can now formulate the main theorem which states correctness of memoization and change-propagation. Memoization and change propagation are correct if their use is undetectable: any evaluation should produce results that are structurally equal to those obtained from a clean run:

Theorem 6 (Correctness)

$$\begin{array}{lll} \sigma, \emptyset, e & \Downarrow^{\mathtt{S}} & v', \sigma', \_, T' \\ \sigma, \emptyset, e & \Downarrow^{\mathtt{S}}_{\emptyset} & v'', \sigma'', \_, T'', \end{array} \quad \Longrightarrow \quad \begin{array}{lll} v' \uparrow \sigma' & = & v'' \uparrow \sigma'' \\ T' \uparrow \sigma' & = & T'' \uparrow \sigma'' \end{array}$$

To be able to prove this, we need to strengthen the statement of the theorem. The problem with the theorem as stated is that it considers two evaluations that start with identical expressions and identical stores. However, after taking a few steps on non-identical evaluation paths, expressions and stores can become different, and the theorem can no longer be used as the induction hypothesis. To formulate the strengthening of the theorem that will let us use a proof by induction, we need to introduce another technical device: pegging.

6.3 Pegging a Store

We need to be able to talk about expressions (and values, and traces) that are "equal up to changes given by a change set χ." We do that by lifting them using a *pegged store.* A pegged store $\sigma \uparrow \chi$ is like the original store σ except for locations that appear in the change set χ; locations in χ are mapped to a dummy value $\bullet$:[1]

$$(\sigma \cdot \chi)(l) = \begin{cases} \bullet & ; \ l \in \chi \\ \sigma(l) & ; \ l \notin \chi \end{cases}$$

6.4 Change Propagation Lemmas

The strengthening of the correctness theorem (Theorem 6) consists of two lemmas, one for the stable case, and one for the changeable case. Each lemma has seven conditions (1)-(7) that must hold before it can be applied. To give some intuition, let us describe the scenario that Lemma 7 characterizes:

At some earlier time, an evaluation of e_1 in store σ_1 (1) has been memoized as is now reported by the oracle. The current evaluation of the same expression e_1 is performed in a store σ_3 which contains all the locations of σ_1 and also all locations—with their corresponding values—allocated during the memoized evaluation (2). This means

[1]For example, the dummy value could be just `()` or 0.

$$\frac{}{\sigma, \chi, \varepsilon \overset{\mathtt{S}}{\curvearrowright} \sigma, \chi, \varepsilon}$$

$$\frac{\sigma, \chi, l \leftarrow \mathtt{T}_c \overset{\mathtt{C}}{\curvearrowright} \sigma', \chi', \mathtt{T}'_c}{\sigma, \chi, \mathtt{mod}_\tau\ l \leftarrow \mathtt{T}_c \overset{\mathtt{S}}{\curvearrowright} \sigma', \chi', \mathtt{mod}_\tau\ l \leftarrow \mathtt{T}'_c} \ \textbf{(mod)}$$

$$\frac{\begin{array}{ccc} \sigma, \chi, \mathtt{T}_s & \overset{\mathtt{S}}{\curvearrowright} & \sigma', \chi', \mathtt{T}''_s \\ \sigma', \chi', \mathtt{T}'_s & \overset{\mathtt{S}}{\curvearrowright} & \sigma'', \chi'', \mathtt{T}'''_s \end{array}}{\sigma, \chi, \mathtt{let}\ \mathtt{T}_s\ \mathtt{T}'_s \overset{\mathtt{S}}{\curvearrowright} \sigma'', \chi'', \mathtt{let}\ \mathtt{T}''_s\ \mathtt{T}'''_s} \ \textbf{(let)}$$

$$\frac{\sigma, \chi, \mathtt{T}_s \quad \overset{\mathtt{S}}{\curvearrowright} \quad \sigma', \chi', \mathtt{T}'_s}{\sigma, \chi, \diamond\ \mathtt{T}_s \overset{\mathtt{S}}{\curvearrowright} \sigma', \chi', \diamond\ \mathtt{T}'_s} \ \textbf{(skip)}$$

$$\frac{}{\sigma, \chi, l \leftarrow \mathtt{write}_\tau \overset{\mathtt{C}}{\curvearrowright} \sigma, \chi, \mathtt{write}_\tau} \ \textbf{(write)}$$

$$\frac{\begin{array}{ccc} \sigma, \chi, \mathtt{T}_s & \overset{\mathtt{C}}{\curvearrowright} & \sigma', \chi', \mathtt{T}'_s \\ \sigma', \chi', l' \leftarrow \mathtt{T}_c & \overset{\mathtt{C}}{\curvearrowright} & \sigma'', \chi'', \mathtt{T}'_c \end{array}}{\sigma, \chi, l' \leftarrow (\mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_c) \overset{\mathtt{C}}{\curvearrowright} \sigma'', \chi'', (\mathtt{let}\ \mathtt{T}'_s\ \mathtt{T}'_c)} \ \textbf{(let)}$$

$$\frac{\begin{array}{c} (l \notin \chi) \\ \sigma, \chi, l' \leftarrow \mathtt{T}_c \overset{\mathtt{C}}{\curvearrowright} \sigma', \chi', \mathtt{T}'_c \end{array}}{\sigma, \chi, l' \leftarrow \mathtt{read}_{l \to x.e} \mathtt{T}_c \overset{\mathtt{C}}{\curvearrowright} \sigma', \chi', \mathtt{read}_{l \to x.e} \mathtt{T}'_c} \ \textbf{(read/no change)}$$

$$\frac{\begin{array}{c} (l \in \chi) \\ \sigma, \chi, l' \leftarrow [\sigma(l)/x]e \Downarrow^{\mathtt{C}} \sigma', \chi', \mathtt{T}'_c \end{array}}{\sigma, \chi, l' \leftarrow \mathtt{read}_{l \to x.e} \mathtt{T}_c \chi \overset{\mathtt{C}}{\curvearrowright} \sigma', \chi' \cup \{l'\}, \mathtt{read}_{l \to x.e} \mathtt{T}'_c} \ \textbf{(read/change)}$$

$$\frac{\sigma, \chi, l \leftarrow \mathtt{T}_c \quad \overset{\mathtt{C}}{\curvearrowright} \quad \sigma', \chi', \mathtt{T}'_c}{\sigma, \chi, l \leftarrow \diamond\ \mathtt{T}_c \overset{\mathtt{C}}{\curvearrowright} \sigma', \chi', \diamond\ \mathtt{T}'_c} \ \textbf{(skip)}$$

Figure 10: Change propagation rules (stable and changeable).

that considering $\mathtt{T}_1$ in σ_3 does not suffer from dangling references, and parts of $\mathtt{T}_1$ that allocate new modifiables but which get skipped by change propagation are still valid in σ_3. The reported change set χ is as subset of the domain of σ_1 (3) and correctly characterizes the differences between the original store σ_1 and the current store σ_3 as far as they concern e_1 (4). Change propagation is performed on the trace of the original evaluation (5). Expression e_1 in the current store σ_3 is equal to some other expression e_2 in its store σ_2 (6). For comparison purposes, we consider a clean evaluation (without **hit**) of that other expression in its store (7).

Under these conditions, change propagation will correctly fix everything up: in the resulting store σ'_3 the change-propagated trace $\mathtt{T}'_1$ looks like the clean trace $\mathtt{T}_2$ in its store, the value v_1 looks like v_2, and the resulting change set χ' correctly characterizes the differences between σ'_3 and the original σ'_1.

We state the lemma for the stable case here. The changeable case together with proofs for both lemmas is given in Appendix B.

Lemma 7 (Change Propagation/Stable)
If

(1) $\sigma_1, \chi_1, e_1 \Downarrow^{\mathtt{S}} v_1, \sigma'_1, \chi'_1, \mathtt{T}_1$,

(2) $\sigma_3 \sqsupseteq \sigma'_1|_{gen(\mathtt{T}_1)}, dom(\sigma_3) \supseteq dom(\sigma_1)$,

(3) $\chi \subseteq dom(\sigma_1)$,

(4) $e_1 \uparrow \sigma_1 \cdot \chi = e_1 \uparrow \sigma_3 \cdot \chi$,

(5) $\sigma_3, \chi, \mathtt{T}_1 \overset{\mathtt{S}}{\curvearrowright} \sigma'_3, \chi', \mathtt{T}'_1$

(6) $e_1 \uparrow \sigma_3 = e_2 \uparrow \sigma_2$,

(7) $\sigma_2, \chi_2, e_2 \Downarrow^{\mathtt{S}}_{\emptyset} v_2, \sigma'_2, \chi'_2, \mathtt{T}_2$

then

I $\mathtt{T}'_1 \uparrow \sigma'_3 = \mathtt{T}_2 \uparrow \sigma'_2$,

II $v_1 \uparrow \sigma'_3 = v_2 \uparrow \sigma'_2$,

III $v_1 \uparrow \sigma'_1 \cdot \chi' = v_1 \uparrow \sigma'_3 \cdot \chi'$.

6.5 Proof of correctness

Theorem 6 is an instance of Lemma 7 by considering a change set that is empty:
Proof: We set $e_1 = e_2 = e$, $v_1 = v'$, $v_2 = v''$, $\sigma_1 = \sigma_2 = \sigma$, $\sigma'_1 = \sigma_3 = \sigma'$, $\sigma'_2 = \sigma''$, $\chi_1 = \chi_2 = \emptyset$, $\mathtt{T}_2 = \mathtt{T}''$. With $\chi = \emptyset$ we have $\mathtt{T}_1 = \mathtt{T}'_1 = \mathtt{T}'$ and $\sigma_3 = \sigma'_3 = \sigma'$. It is easy to show that conditions (1)-(7) of Lemma 7 are satisfied and that its conclusions imply those of the theorem. ■

References

[1] M. A. Abam and M. de Berg. Kinetic sorting and kinetic convex hulls. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 190–197, New York, NY, USA, 2005. ACM Press.

[2] U. A. Acar. *Self-Adjusting Computation.* PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[3] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. A library for self-adjusting computation. In *ACM SIGPLAN Workshop on ML*, 2005.

[4] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[5] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.

$\tau \uparrow \sigma$	$=$	τ
$v \uparrow \sigma$	$=$	$v, v \in \{(), n, x\}$
$l \uparrow \sigma$	$=$	$\sigma[l] \uparrow \sigma$
$(v_1, v_2) \uparrow \sigma$	$=$	$(v_1 \uparrow \sigma, v_2 \uparrow \sigma)$
$\texttt{inl}_{\tau_1+\tau_2} v$	$=$	$\texttt{inl}_{\tau_1+\tau_2}(v \uparrow \sigma)$
$\texttt{inr}_{\tau_1+\tau_2} v$	$=$	$\texttt{inr}_{\tau_1+\tau_2}(v \uparrow \sigma)$
$\texttt{fun}_{\texttt{S}}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e_s\ \texttt{end}$	$=$	$\texttt{fun}_{\texttt{S}}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e_s \uparrow \sigma\ \texttt{end}$
$\texttt{fun}_{\texttt{C}}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e_c\ \texttt{end}$	$=$	$\texttt{fun}_{\texttt{C}}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e_c \uparrow \sigma\ \texttt{end}$
$o \uparrow \sigma$	$=$	$o, o \in \{\texttt{not}, \texttt{+}, \texttt{-}, \texttt{=}, \texttt{<}, \ldots\}$
$o(v_1, \ldots, v_n) \uparrow \sigma$	$=$	$o(v_1 \uparrow \sigma, \ldots, v_n \uparrow \sigma)$
$\texttt{mod}_\tau\ e_c \uparrow \sigma$	$=$	$\texttt{mod}_\tau\ e_c \uparrow \sigma$
$\texttt{apply}_{\texttt{S}}(v_1, v_2) \uparrow \sigma$	$=$	$\texttt{apply}_{\texttt{S}}(v_1 \uparrow \sigma, v_2 \uparrow \sigma)$
$\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e'_s\ \texttt{end} \uparrow \sigma$	$=$	$\texttt{let}\ x\ \texttt{be}\ e_s \uparrow \sigma\ \texttt{in}\ e'_s \uparrow \sigma\ \texttt{end}$
$\texttt{let}\ x_1 \times x_2\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} \uparrow \sigma$	$=$	$\texttt{let}\ x_1 \times x_2\ \texttt{be}\ v \uparrow \sigma\ \texttt{in}\ e_s \uparrow \sigma\ \texttt{end}$
$\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\ (x_1 : \tau_1) \Rightarrow e_s \mid \texttt{inr}\ (x_2 : \tau_2) \Rightarrow e'_s \uparrow \sigma$	$=$	$\texttt{case}\ v \uparrow \sigma\ \texttt{of}\ \texttt{inl}\ (x_1 : \tau_1) \Rightarrow e_s \uparrow \sigma \mid \texttt{inr}\ (x_2 : \tau_2) \Rightarrow e'_s \uparrow \sigma$
$\texttt{write}_\tau(v) \uparrow \sigma$	$=$	$\texttt{write}_\tau(v \uparrow \sigma)$
$\texttt{apply}_{\texttt{C}}(v_1, v_2) \uparrow \sigma$	$=$	$\texttt{apply}_{\texttt{C}}(v_1 \uparrow \sigma, v_2 \uparrow \sigma)$
$\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_c\ \texttt{end} \uparrow \sigma$	$=$	$\texttt{let}\ x\ \texttt{be}\ e_s \uparrow \sigma\ \texttt{in}\ e_c \uparrow \sigma\ \texttt{end}$
$\texttt{let}\ x_1 \times x_2\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} \uparrow \sigma$	$=$	$\texttt{let}\ x_1 \times x_2\ \texttt{be}\ v \uparrow \sigma\ \texttt{in}\ e_c \uparrow \sigma\ \texttt{end}$
$\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c\ \texttt{end} \uparrow \sigma$	$=$	$\texttt{read}\ v \uparrow \sigma\ \texttt{as}\ x\ \texttt{in}\ e_c \uparrow \sigma\ \texttt{end}$
$\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\ (x_1 : \tau_1) \Rightarrow e_c \mid \texttt{inr}\ (x_2 : \tau_2) \Rightarrow e'_c \uparrow \sigma$	$=$	$\texttt{case}\ v \uparrow \sigma\ \texttt{of}\ \texttt{inl}\ (x_1 : \tau_1) \Rightarrow e_c \uparrow \sigma \mid \texttt{inr}\ (x_2 : \tau_2) \Rightarrow e'_c \uparrow \sigma$

Figure 11: The lift operation

[6] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[7] U. A. Acar, G. E. Blelloch, and J. L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*, 2005.

[8] P. K. Agarwal, D. Eppstein, L. J. Guibas, and M. R. Henzinger. Parametric and kinetic minimum spanning trees. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 596–605, 1998.

[9] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming*, pages 270–280, 1997.

[10] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully-dynamic trees with top trees, 2003. The Computing Research Repository (CoRR)[cs.DS/0310065].

[11] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[12] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.

[13] J. Basch, L. J. Guibas, C. D. Silverstein, and L. Zhang. A practical evaluation of kinetic data structures. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 388–390, New York, NY, USA, 1997. ACM Press.

[14] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 617–626, 2002.

[15] R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–61, 1991.

[16] A. Czumaj and C. Sohler. Soft kinetic data structures. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 865–872. Society for Industrial and Applied Mathematics, 2001.

[17] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

[18] 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*, pages 365–372, 1987.

[19] J. Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems.* PhD thesis, Department of Computer Science, Cornell University, Nov. 1991.

[20] 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.

[21] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997.

[22] R. L. Graham. An efficient algorithm for determining the convex hull of a finete planar set. *Information Processing Letters*, 1:132–133, 1972.

[23] L. J. Guibas. Kinetic data structures—a state of the art report. In *Proceedings of the Third Workshop on Algorithmic Foundations of Robotics*, 1998.

[24] L. J. Guibas and M. I. Karavelas. Interval methods for kinetic simulations. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 255–264. ACM Press, 1999.

[25] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[26] R. Hoover. *Incremental Graph Evaluation.* PhD thesis, Department of Computer Science, Cornell University, May 1987.

[27] R. Jakob. *Dynamic Planar Convex Hull.* PhD thesis, Department of Computer Science, University of Aarhus, 2002.

[28] H. Kaplan, R. E. Tarjan, and K. Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 836–844. Society for Industrial and Applied Mathematics, 2001.

[29] M. I. Karavelas and L. J. Guibas. Static and kinetic geometric spanners with applications. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 168–176. Society for Industrial and Applied Mathematics, 2001.

[30] Y. A. Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach*. PhD thesis, Department of Computer Science, Cornell University, Jan. 1996.

[31] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.

[32] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 487–489, 1985.

[33] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling (extended abstract). In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 180–196, 1991.

[34] M. H. Overmans and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[35] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

[36] W. Pugh. *Incremental computation via function caching*. PhD thesis, Department of Computer Science, Cornell University, August 1988.

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

[38] T. Radzik. Implementation of dynamic trees with in-subtree operations. *ACM Journal of Experimental Algorithms*, 3:9, 1998.

[39] T. Reps. *Generating Language-Based Environments*. PhD thesis, Department of Computer Science, Cornell University, Aug. 1982.

[40] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[41] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[42] R. S. Sundaresh and P. Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on POPL*, pages 1–13, Jan. 1991.

[43] R. Tarjan and R. Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[44] R. E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:167–177, 1997.

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

A Static semantics

The SLf type system is inspired by the type theory of modal logic given by Pfenning and Davies [35]. We distinguish two modes, the *stable* and the *changeable*, corresponding to the distinction between terms and expressions, respectively, in Pfenning and Davies' work.

The judgement $\Lambda; \Gamma \vdash_{\mathtt{S}} e : \tau$ states that e is a well-formed stable expression of type τ, relative to Λ and Γ. The judgement $\Lambda; \Gamma \vdash_{\mathtt{C}} e : \tau$ states that e is a well-formed changeable expression of type τ, relative to Λ and Γ. Here Λ is a *location typing* and Γ is a *variable typing*; these are finite functions assigning types to locations and variables, respectively.

The typing judgements for stable and changeable expressions are shown in Figure 12 respectively. For primitive functions, we assume a typing relation $\vdash_o$. For stable expression, the interesting rules are the `mod` and the changeable functions. The bodies of these expressions are changeable expressions and therefore they are typed in the changeable mode. For changeable expressions, the interesting rule is the `let` rule. The body of `let` is a changeable expression and thus typed in the changeable mode; the expression bound, however, is a stable expression and thus typed in the stable mode. The `mod` and `let` rules therefore provide inclusion between two modes.

B Proofs

The proof of the change-propagation lemmas relies on several properties of the lift operation, in particular the following:

Theorem 9
Let e be an expression and σ, σ' be two stores and χ, χ' two changed sets. If

1. $\sigma' \sqsupseteq \sigma$,
2. $\chi' \supseteq \chi$, *and*
3. $\chi' \setminus \chi \subseteq \mathtt{dom}(\sigma') \setminus \mathtt{dom}(\sigma)$

then $e \uparrow \sigma \cdot \chi = e \uparrow \sigma' \cdot \chi'$.

B.1 Proof of change propagation lemmas

Lemma 7 (Change Propagation/Stable)
If

(A) $\sigma_1, \chi_1, e_1 \Downarrow^{\mathtt{S}} v_1, \sigma_1', \chi_1', T_1$,

(B) $\sigma_3 \sqsupseteq \sigma_1'|_{\mathtt{gen}(T_1)}, \mathtt{dom}(\sigma_3) \supseteq \mathtt{dom}(\sigma_1)$,

(C) $\chi \subseteq \mathtt{dom}(\sigma_1)$,

(D) $e_1 \uparrow \sigma_1 \cdot \chi = e_1 \uparrow \sigma_3 \cdot \chi$,

(E) $\sigma_3, \chi, T_1 \overset{\mathtt{S}}{\curvearrowright} \sigma_3', \chi', T_1'$

(F) $e_1 \uparrow \sigma_3 = e_2 \uparrow \sigma_2$,

(G) $\sigma_2, \chi_2, e_2 \Downarrow^{\mathtt{S}}_{\emptyset} v_2, \sigma_2', \chi_2', T_2$

then

1. $T_1' \uparrow \sigma_3' = T_2 \uparrow \sigma_2'$,
2. $v_1 \uparrow \sigma_3' = v_2 \uparrow \sigma_2'$,
3. $v_1 \uparrow \sigma_1' \cdot \chi' = v_1 \uparrow \sigma_3' \cdot \chi'$.

Lemma 8 (Change Propagation/Changeable)
If

(A) $\sigma_1, \chi_1, l_1 \leftarrow e_1 \Downarrow^{\mathtt{C}} \sigma_1', \chi_1', T_1$,

(B) $\sigma_2, \chi_2, l_2 \leftarrow e_2 \Downarrow^{\mathtt{C}}_{\emptyset} \sigma_2', \chi_2', T_2$,

(C) $\chi \subseteq \mathtt{dom}(\sigma_1)$ *s.t.* $e_1 \uparrow \sigma_1 \cdot \chi \geq e_2 \uparrow \sigma_2$,

$$\frac{}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{()} : \mathtt{unit}}\ \textbf{(unit)} \qquad \frac{}{\Lambda;\Gamma \vdash_{\mathsf{S}} n : \mathtt{int}}\ \textbf{(number)}$$

$$\frac{(\Gamma(x) = \tau)}{\Lambda;\Gamma \vdash_{\mathsf{S}} x : \tau}\ \textbf{(variable)} \qquad \frac{(\Lambda(l) = \tau)}{\Lambda;\Gamma \vdash_{\mathsf{S}} l : \tau\ \mathtt{mod}}\ \textbf{(location)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_1 : \tau_1 \quad \Lambda;\Gamma \vdash_{\mathsf{S}} v_2 : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{S}} (v_1, v_2) : \tau_1 \times \tau_2}\ \textbf{(pair)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau_1}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{inl}_{\tau_1+\tau_2} v : \tau_1 + \tau_2}\ \textbf{(sum/inl)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{inr}_{\tau_1+\tau_2} v : \tau_1 + \tau_2}\ \textbf{(sum/inr)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_i : \tau_i \quad (1 \le i \le n) \quad \vdash_o o : (\tau_1, \ldots, \tau_n)\ \tau}{\Lambda;\Gamma \vdash_{\mathsf{S}} o(v_1, \ldots, v_n) : \tau}\ \textbf{(prim.)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{C}} e : \tau}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{mod}_\tau\ e : \tau\ \mathtt{mod}}\ \textbf{(modifiable)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_1 : (\tau_1 \xrightarrow{\mathsf{S}} \tau_2) \quad \Lambda;\Gamma \vdash_{\mathsf{S}} v_2 : \tau_1}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{apply}_{\mathsf{S}}(v_1, v_2) : \tau_2}\ \textbf{(apply)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} e : \tau \quad \Lambda;\Gamma, x : \tau \vdash_{\mathsf{S}} e' : \tau'}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ e'\ \mathtt{end} : \tau'}\ \textbf{(let)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau_1 \times \tau_2 \quad \Lambda;\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathsf{S}} e : \tau}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{let}\, x_1{\times}x_2\, \mathtt{be}\, v\, \mathtt{in}\, e\, \mathtt{end} : \tau}\ \textbf{(let }\times\textbf{)}$$

$$\frac{\Lambda;\Gamma,\ f : \tau_1 \xrightarrow{\mathsf{S}} \tau_2,\ x : \tau_1 \vdash_{\mathsf{S}} e : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{fun}_{\mathsf{S}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end} : (\tau_1 \xrightarrow{\mathsf{S}} \tau_2)}\ \textbf{(fun/st.)}$$

$$\frac{\Lambda;\Gamma,\ f : \tau_1 \xrightarrow{\mathsf{C}} \tau_2,\ x : \tau_1 \vdash_{\mathsf{C}} e : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{fun}_{\mathsf{C}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end} : (\tau_1 \xrightarrow{\mathsf{C}} \tau_2)}\ \textbf{(fun/ch.)}$$

$$\frac{\begin{array}{c}\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau_1 + \tau_2 \\ \Lambda;\Gamma,\ x_1 : \tau_1 \vdash_{\mathsf{S}} e_1 : \tau \\ \Lambda;\Gamma,\ x_2 : \tau_2 \vdash_{\mathsf{S}} e_2 : \tau\end{array}}{\begin{array}{l}\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}\,(x_1{:}\tau_1) \Rightarrow e_1\ : \tau \\ \qquad\qquad\qquad\ \mathtt{|}\ \ \mathtt{inr}\,(x_2{:}\tau_2) \Rightarrow e_2\end{array}}\ \textbf{(case)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{write}_\tau(v) : \tau}\ \textbf{(write)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_1 : (\tau_1 \xrightarrow{\mathsf{C}} \tau_2) \quad \Lambda;\Gamma \vdash_{\mathsf{S}} v_2 : \tau_1}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{apply}_{\mathsf{C}}(v_1, v_2) : \tau_2}\ \textbf{(apply)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} e : \tau \quad \Lambda;\Gamma, x : \tau \vdash_{\mathsf{C}} e' : \tau'}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ e'\ \mathtt{end} : \tau'}\ \textbf{(let)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau_1 \times \tau_2 \quad \Lambda;\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathsf{C}} e : \tau}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{let}\, x_1{\times}x_2\, \mathtt{be}\, v\, \mathtt{in}\, e\, \mathtt{end} : \tau}\ \textbf{(let }\times\textbf{)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau\ \mathtt{mod} \quad \Lambda;\Gamma, x : \tau \vdash_{\mathsf{C}} e : \tau'}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end} : \tau'}\ \textbf{(read)}$$

$$\frac{\begin{array}{c}\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau_1 + \tau_2 \\ \Lambda;\Gamma,\ x_1 : \tau_1 \vdash_{\mathsf{C}} e_1 : \tau \\ \Lambda;\Gamma,\ x_2 : \tau_2 \vdash_{\mathsf{C}} e_2 : \tau\end{array}}{\begin{array}{l}\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}\,(x_1{:}\tau_1) \Rightarrow e_1\ : \tau \\ \qquad\qquad\qquad\ \mathtt{|}\ \ \mathtt{inr}\,(x_2{:}\tau_2) \Rightarrow e_2\end{array}}\ \textbf{(case)}$$

Figure 12: Typing of stable and changeable expressions.

(D) $\sigma_3 \sqsupseteq \sigma'_1|_{\texttt{gen}(T_1)}, \texttt{dom}(\sigma_3) \supseteq \texttt{dom}(\sigma_1)$,

(E) $e_1 \uparrow \sigma_3 = e_2 \uparrow \sigma_2$,

(F) $e_1 \uparrow \sigma_1 \cdot \chi = e_1 \uparrow \sigma_3 \cdot \chi$,

(G) $\sigma_3, T_1, \chi \overset{\mathsf{S}}{\curvearrowright} \sigma'_3, T'_1, \chi'$,

then

1. $T'_1 \uparrow \sigma'_3 = T_2 \uparrow \sigma'_2$,
2. $l_1 \uparrow \sigma'_3 = l_2 \uparrow \sigma'_2$,
3. $l_1 \uparrow \sigma'_1 \cdot \chi' \geq l_2 \uparrow \sigma'_2$,
4. $l_1 \uparrow \sigma'_1 \cdot \chi' = l_1 \uparrow \sigma'_3 \cdot \chi$.

Key cases of the proof are shown in Appendix B.

Proof:

We consider finite stores only. The proofs for Lemmas 7 and 8 proceed *together* by simultaneous induction on $(|\sigma_3| \times |\texttt{T}'_1|)$, i.e., on pairs consisting of the number of bound locations in σ_3 and the length of trace $\texttt{T}'_1$, under lexicographic ordering. The case for values and the case for primitives, where the trace is empty constitute the bases cases.

Notice that condition (F) in each of the two lemmas implies that we only consider expressions e_1 and e_2 which have the same syntactic structure up to values and locations. In particular, this means that their outermost syntactic forms are guaranteed to be equal.

• values: Assume that

(A) $\sigma_1, \chi_1, v_1 \Downarrow^{\mathsf{S}} v_1, \sigma_1, \chi'_1, \varepsilon$

(B) $\sigma_3 \sqsupseteq \sigma_1|_{\texttt{gen}(\varepsilon)}, \texttt{dom}(\sigma_3) \supseteq \texttt{dom}(\sigma_1)$

(C) $\chi \subseteq \texttt{dom}(\sigma_1)$

(D) $v_1 \uparrow \sigma_1 \cdot \chi = v_1 \uparrow \sigma_3 \cdot \chi$

(E) $\sigma_3, \varepsilon, \chi \overset{\mathsf{S}}{\curvearrowright} \sigma_3, \varepsilon, \chi$.

(F) $v_1 \uparrow \sigma_3 = v_2 \uparrow \sigma_2$

(G) $\sigma_2, v_2, \chi_2 \Downarrow^{\mathsf{S}}_{\emptyset} v_2, \sigma_2, \chi_2, \varepsilon$

Then the following hold.

1. $\varepsilon \uparrow \sigma_3 = \varepsilon \uparrow \sigma_2$; follows trivially.
2. $v_1 \uparrow \sigma_3 = v_2 \uparrow \sigma_2$; follows by property F.
3. $v_1 \uparrow \sigma_1 \cdot \chi = v_1 \uparrow \sigma_3 \cdot \chi$; follows by propoerty D.

• primitives: Assuming that the following hold.

(A) $\sigma_1, \chi_1, o(u_1, \ldots, u_n) \Downarrow^{\mathsf{S}} v_1, \sigma_1, \chi_1, \varepsilon$

(B) $\sigma_3 \sqsupseteq \sigma_1|_{\texttt{gen}(\varepsilon)}$

(C) $\chi \subseteq \texttt{dom}(\sigma_1)$

(D) $o(u_1, \ldots, u_n) \uparrow \sigma_1 \cdot \chi = o(u_1, \ldots, u_n) \uparrow \sigma_3 \cdot \chi$

(E) $\sigma_3, \texttt{T}_1, \chi \overset{\mathsf{S}}{\curvearrowright} \sigma_3, \texttt{T}_1, \chi$

(F) $o(u_1, \ldots, u_n) \uparrow \sigma_3 = o(w_1, \ldots, w_n) \uparrow \sigma_2$

(G) $\sigma_2, \chi_2, o(w_1, \ldots, w_n) \Downarrow^{\mathsf{S}}_{\emptyset} v_2, \sigma_2, \chi_2, \varepsilon$

we show that the following are true.

1. $\varepsilon \uparrow \sigma_3 = \varepsilon \uparrow \sigma_2$ holds trivially.
2. $v_1 \uparrow \sigma_3 = v_2 \uparrow \sigma_2$. By property F, we know that $o(u_1, \ldots, u_n) \uparrow \sigma_3 = o(w_1, \ldots, w_n) \uparrow \sigma_2$. Since the arguments to primitives cannot contain locations, $o(u_1, \ldots, u_n) = o(w_1, \ldots, w_n)$, and thus $v_1 = v_2$. Since v_1 and v_2 contain no locations, the property follows.
3. $v_1 \uparrow \sigma_1 \cdot \chi = v_1 \uparrow \sigma_3 \cdot \chi$. Like the prevoius property this property following from the fact that $v_1 = v_2$ and that v_1 and v_2 contains no locations.

• `mod`: Assume that the following hold:

(A) $\sigma_1, \chi_1, \texttt{mod}_\tau\ e_1 \Downarrow^{\mathsf{S}} l_1, \sigma'_1, \chi'_1, \texttt{mod}_\tau\ l_1 \leftarrow \texttt{T}_1$

(B) $\sigma_3 \sqsupseteq \sigma'_1|_{\texttt{gen}(\texttt{mod}_\tau\ \texttt{T}_1)}, \texttt{dom}(\sigma_3) \supseteq \texttt{dom}(\sigma_1)$

(C) $\chi \subseteq \texttt{dom}(\sigma_1)$

(D) $\texttt{mod}_\tau\ e_1 \uparrow \sigma_1 \cdot \chi = \texttt{mod}_\tau\ e_1 \uparrow \sigma_3 \cdot \chi$

(E) $\sigma_3, \chi, \texttt{mod}_\tau\ \texttt{T}_1 \overset{\mathsf{S}}{\curvearrowright} \sigma'_3, \chi', \texttt{mod}_\tau\ \texttt{T}'_1$

(F) $\texttt{mod}_\tau\ e_1 \uparrow \sigma_3 = \texttt{mod}_\tau\ e_2 \uparrow \sigma_2$

(G) $\sigma_2, \chi_2, \texttt{mod}_\tau\ e_2 \Downarrow^{\mathsf{S}}_{\emptyset} l_2, \sigma'_2, \chi'_2, \texttt{mod}_\tau\ l_2 \leftarrow \texttt{T}_2$

we show that the following are true.

1. $\texttt{mod}_\tau\ \texttt{T}'_1 \uparrow \sigma'_3 = \texttt{mod}_\tau\ \texttt{T}_2 \uparrow \sigma'_2$,
2. $l_1 \uparrow \sigma'_3 = l_2 \uparrow \sigma'_2$,
3. $l_1 \uparrow \sigma'_1 \cdot \chi' = l_1 \uparrow \sigma'_3 \cdot \chi'$.

We first consider the evaluation, and change-propagation rules for `mod` expressions (shown below) and apply the induction hypothesis to the antecedents.

$$\frac{\begin{array}{c}(l_1 \notin \texttt{dom}(\sigma_1))\\ \sigma_1[l_1 \to \square], \chi_1, l_1 \leftarrow e_1 \Downarrow^{\mathsf{C}} \sigma'_1, \chi'_1, \texttt{T}_1\end{array}}{\sigma_1, \chi_1, \texttt{mod}_\tau\ e_1 \Downarrow^{\mathsf{S}} l_1, \sigma'_1, \chi'_1, \texttt{mod}_\tau\ l_1 \leftarrow \texttt{T}_1}\ \textbf{(mod/1)}$$

$$\frac{\begin{array}{c}(l_2 \notin \texttt{dom}(\sigma_1))\\ \sigma_2[l_2 \to \square], \chi_2, l_2 \leftarrow e_2 \Downarrow^{\mathsf{C}} \sigma'_2, \chi'_2, \texttt{T}_2\end{array}}{\sigma_2, \chi_2, \texttt{mod}_\tau\ e_2 \Downarrow^{\mathsf{S}} l, \sigma'_1, \chi'_2, \texttt{mod}_\tau\ l_2 \leftarrow \texttt{T}_2}\ \textbf{(mod/2)}$$

$$\frac{\sigma_3, \chi, l_1 \leftarrow \texttt{T}_1 \overset{\mathsf{C}}{\curvearrowright} \sigma'_3, \chi', \texttt{T}'_1}{\sigma_3, \chi_1, \texttt{mod}_\tau\ l_1 \leftarrow \texttt{T}_1 \overset{\mathsf{S}}{\curvearrowright} \sigma'_3, \chi', \texttt{mod}_\tau\ l_1 \leftarrow \texttt{T}'_1}\ \textbf{(mod/c.p.)}$$

We first show that the following are true.

(AA) $\sigma_1, \chi_1, l_1 \leftarrow e_1 \Downarrow^{\mathsf{C}} \sigma'_1, \chi'_1, \texttt{T}_1$

(BB) $\sigma_3 \sqsupseteq \sigma'_1|_{\texttt{gen}(\texttt{T}_1)}, \texttt{dom}(\sigma_3) \supseteq \texttt{dom}(\sigma_1)$

(CC) $\chi \subseteq \texttt{dom}(\sigma_1)$

(DD) $e_1 \uparrow \sigma_1 \cdot \chi = e_1 \uparrow \sigma_3 \cdot \chi$

(EE) $\sigma_3, \chi, \texttt{T}_1 \overset{\mathsf{S}}{\curvearrowright} \sigma'_3, \texttt{T}'_1, \chi'$

(FF) $e_1 \uparrow \sigma_3 = e_2 \uparrow \sigma_2$

(GG) $\sigma_2, \chi_2, l_2 \leftarrow e_2 \Downarrow^{\mathtt{C}}_{\emptyset} \sigma'_2, \chi'_2, \mathtt{T}_2$

The properties AA, EE, GG follow from A,E,G and by evaluation and change-propagation rules for `mod` expressions shown above. The properties BB,CC,DD,FF follow trivially from B,C,D,and F respectively. Since the $\mathtt{T}_2$ is shorter than $\mathtt{mod}_\tau\ l_2 \leftarrow \mathtt{T}_2$, the induction hypothesis applies, and by Lemma 8 and we know that the following are true.

11. $\mathtt{T}'_1 \uparrow \sigma'_3 = \mathtt{T}_2 \uparrow \sigma'_2$,

22. $l_1 \uparrow \sigma'_3 = l_2 \uparrow \sigma'_2$,

44. $l_1 \uparrow \sigma'_1 \cdot \chi' = l_1 \uparrow \sigma'_3 \cdot \chi'$.

Consider the properties 1,2, and 4 that we want to show. Of these 2, and 4 follow directly from 22, and 44. The property 1 follows directly from 11 and by the definition of the lift operation for traces. More precisely, $\mathtt{mod}_\tau\ \mathtt{T}'_1 \uparrow \sigma'_3 = \mathtt{mod}_\tau\ (\mathtt{T}'_1 \uparrow \sigma'_3) = \mathtt{mod}_\tau\ (\mathtt{T}_2 \uparrow \sigma'_2) = \mathtt{mod}_\tau\ \mathtt{T}_2 \uparrow \sigma'_2$.

• stable `apply`: Assume that the following hold:

(A) $\sigma_1, \chi_1, \mathtt{apply}_{\mathtt{S}}(u_1, u_2) \Downarrow^{\mathtt{S}} v, \sigma'_1, \chi'_1, \diamond\ \mathtt{T}_1$

(B) $\sigma_3 \sqsupseteq \sigma'_1|_{\mathtt{gen}(\mathtt{T}_1)}, \mathtt{dom}(\sigma_3) \supseteq \mathtt{dom}(\sigma_1)$

(C) $\chi \subseteq \mathtt{dom}(\sigma_1)$

(D) $\mathtt{apply}_{\mathtt{S}}(u_1, u_2) \uparrow \sigma_1 \cdot \chi = \mathtt{apply}_{\mathtt{S}}(u_1, u_2) \uparrow \sigma_3 \cdot \chi$

(E) $\sigma_3, \chi, \diamond\ \mathtt{T}_1 \overset{\mathtt{S}}{\curvearrowright} \sigma'_3, \chi', \diamond\ \mathtt{T}'_1$

(F) $\mathtt{apply}_{\mathtt{S}}(u_1, u_2) \uparrow \sigma_3 = \mathtt{apply}_{\mathtt{S}}(w_1, w_2) \uparrow \sigma_2$

(G) $\sigma_2, \chi_2, \mathtt{apply}_{\mathtt{S}}(w_1, w_2) \Downarrow^{\mathtt{S}}_{\emptyset} v_2, \sigma'_2, \chi'_2, \diamond\ \mathtt{T}_2$

Furthermore, let $u_1 = \mathtt{fun}_{\mathtt{S}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e_1\ \mathtt{end}$ and $w_1 = \mathtt{fun}_{\mathtt{S}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e_2\ \mathtt{end}$.

We need to consider two cases, one for each of the two rules that could be used to derive (A). It is not necessary to make the same distinction for (G) since it stipulates a clean evaluation. Under the assumption that (A) was derived using **apply/miss** it is straightforward to set up and prove the induction step. We omit the details here.

For the case that (A) was derived using **apply/hit**, we compare that evaluation to a corresponding clean run $\sigma_1, \chi_1, \mathtt{apply}_{\mathtt{S}}(u_1, u_2) \Downarrow^{\mathtt{S}}_{\emptyset} v_0, \sigma_0, \chi_0, \diamond\ \mathtt{T}_0$. Unfortunately, we cannot argue that the induction hypothesis applies based on the length of the trace since there are no restrictions on the length of $\mathtt{T}_1$ or $\mathtt{T}_0$. However, from condition (B) we know that $|\sigma_1| \leq |\sigma_3|$. If $|\sigma_1| < |\sigma_3|$, then we can use the induction hypothesis based on store size, showing that (A) is equivalent to the clean run. This result then lets us fall back to the **apply/miss** case above. If $|\sigma_1| = |\sigma_3|$, then $|\sigma'_1|_{\mathtt{gen}(\mathtt{T}_1)}|$ must be empty, in other words: $\mathtt{gen}(\mathtt{T}_1) = \emptyset$. But then $\mathtt{T}_1$ cannot contain any **read**s, so change propagation is a no-op and induction on the length of the trace works.

• `let`:

Assume that the following hold.

(A) $\sigma_1, \chi_1, \mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \Downarrow^{\mathtt{S}} u, \sigma''_1, \chi''_1, \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2$

(B) $\sigma_3 \sqsupseteq \sigma''_1|_{\mathtt{gen}(\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2)}$. Since $\mathtt{gen}(\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2) = \mathtt{gen}(\mathtt{T}_1) \cup \mathtt{gen}(\mathtt{T}_2)$, we know that $\sigma_3 \sqsupseteq \sigma''_1|_{\mathtt{gen}(\mathtt{T}_1)}$ and $\sigma_3 \sqsupseteq \sigma''_1|_{\mathtt{gen}(\mathtt{T}_2)}$;

$\mathtt{dom}(\sigma_3) \supseteq \mathtt{dom}(\sigma_1)$

(C) $\chi \subseteq \mathtt{dom}(\sigma_1)$

(D) $\mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \uparrow \sigma_1 \cdot \chi = \mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \uparrow \sigma_3 \cdot \chi$; this implies that $e_1 \uparrow \sigma_1 \cdot \chi = e_1 \uparrow \sigma_3 \cdot \chi$ and $e_2 \uparrow \sigma_1 \cdot \chi = e_2 \uparrow \sigma_3 \cdot \chi$.

(E) $\sigma_3, \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2, \chi \overset{\mathtt{S}}{\curvearrowright} \sigma''_3, \mathtt{let}\ \mathtt{T}'_1\ \mathtt{T}'_2, \chi''$.

(F) $\mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \uparrow \sigma_3 = \mathtt{let}\ x\ \mathtt{be}\ e_3\ \mathtt{in}\ e_4\ \mathtt{end} \uparrow \sigma_2$, this implies that $e_1 \uparrow \sigma_3 = e_3 \uparrow \sigma_2$, and $e_2 \uparrow \sigma_3 = e_4 \uparrow \sigma_2$.

(G) $\sigma_2, \chi_2, \mathtt{let}\ x\ \mathtt{be}\ e_3\ \mathtt{in}\ e_4\ \mathtt{end} \Downarrow^{\mathtt{S}}_{\emptyset} v, \sigma''_2, \chi''_2, \mathtt{let}\ \mathtt{T}_3\ \mathtt{T}_4$.

We need to show that the following.

1. $\mathtt{let}\ \mathtt{T}'_1\ \mathtt{T}'_2 \uparrow \sigma''_3 = \mathtt{let}\ \mathtt{T}_3\ \mathtt{T}_4 \uparrow \sigma''_2$
2. $u \uparrow \sigma''_3 = v \uparrow \sigma''_2$,
3. $u \uparrow \sigma''_1 \cdot \chi'' = u \uparrow \sigma''_3 \cdot \chi''$.

To this end, we consider the evaluation and the change-propagation rules for `let` expressions (shown below) and apply the induction hypothesis to the antecedents.

$$\frac{\begin{array}{lcl} \sigma_1, \chi_1, e_1 & \Downarrow^{\mathtt{S}} & u_1, \sigma'_1, \chi'_1, \mathtt{T}_1 \\ \sigma'_1, \chi'_1, [u_1/x]e_2 & \Downarrow^{\mathtt{S}} & u, \sigma''_1, \chi''_1, \mathtt{T}_2 \end{array}}{\sigma, \chi_1, \mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \Downarrow^{\mathtt{S}} u, \sigma''_1, \chi''_1, \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2} \quad \textbf{(let/1)}$$

$$\frac{\begin{array}{lcl} \sigma_2, \chi_2, e_3 & \Downarrow^{\mathtt{S}} & v_1, \sigma'_2, \chi'_2, \mathtt{T}_3 \\ \sigma'_2, \chi'_2, [v_1/x]e_4 & \Downarrow^{\mathtt{S}} & v, \sigma''_2, \chi''_2, \mathtt{T}_4 \end{array}}{\sigma_2, \chi_2, \mathtt{let}\ x\ \mathtt{be}\ e_3\ \mathtt{in}\ e_4\ \mathtt{end} \Downarrow^{\mathtt{S}} v, \sigma''_1, \chi''_2, \mathtt{let}\ \mathtt{T}_3\ \mathtt{T}_4} \quad \textbf{(let/2)}$$

$$\frac{\begin{array}{lcl} \sigma_3, \chi, \mathtt{T}_1 & \overset{\mathtt{S}}{\curvearrowright} & \sigma'_3, \chi', \mathtt{T}'_1 \\ \sigma'_3, \chi', \mathtt{T}_2 & \overset{\mathtt{S}}{\curvearrowright} & \sigma''_3, \chi'', \mathtt{T}'_2 \end{array}}{\sigma_3, \chi, \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2 \overset{\mathtt{S}}{\curvearrowright} \sigma''_3, \chi'', \mathtt{let}\ \mathtt{T}'_1\ \mathtt{T}'_2} \quad \textbf{(let/c.p.)}$$

• **Step I:** We first consider the top antecedents from each rule and show that the following hold.

(AA) $\sigma_1, \chi_1, e_1 \Downarrow^{\mathtt{S}} u_1, \sigma'_1, \chi'_1, \mathtt{T}_1$; this follows by evaluation rule **let/1**.

(BB) $\sigma_3 \sqsupseteq \sigma'_1|_{\mathtt{gen}(\mathtt{T}_1)}$. By property B, we know that $\sigma_3 \sqsupseteq \sigma''_1|_{\mathtt{gen}(\mathtt{T}_1)}$. Since $\sigma''_1 \sqsupseteq \sigma'_1$, $\sigma_3 \sqsupseteq \sigma''_1|_{\mathtt{gen}(\mathtt{T}_1)}$ holds.

$\mathtt{dom}(\sigma_3) \supseteq \mathtt{dom}(\sigma_1)$; this follows directly from B.

(CC) $\chi \subseteq \mathtt{dom}(\sigma_1)$; follows by property C.

(DD) $e_1 \uparrow \sigma_1 \cdot \chi = e_1 \uparrow \sigma_3 \cdot \chi$; follows directly from D.

(EE) $\sigma_3, \mathtt{T}_1, \chi \overset{\mathtt{S}}{\curvearrowright} \sigma'_3, \mathtt{T}'_1, \chi'$; follows by rule **let/c.p.**.

(FF) $e_1 \uparrow \sigma_3 = e_3 \uparrow \sigma_2$; follows directly from F.

(GG) $\sigma_2, \chi_2, e_3 \Downarrow^{\mathtt{S}}_{\emptyset} v_1, \sigma'_2, \chi'_2, \mathtt{T}_3$; follows by evaluation rule **let/2**.

Since the trace $\mathtt{T}_3$ is shorter than the trace $\mathtt{let}\ \mathtt{T}_3\ \mathtt{T}_4$, the induction hypothesis applies and the following are true.

11. $\mathtt{T}'_1 \uparrow \sigma'_3 = \mathtt{T}_3 \uparrow \sigma'_2$

22. $u_1 \uparrow \sigma'_3 = v_1 \uparrow \sigma'_2$

44. $u_1 \uparrow \sigma'_1 \cdot \chi' = u_1 \uparrow \sigma'_3 \cdot \chi'$

• **Step II:** We consider the second (bottom) antecedents from each rule and show that the following hold.

AAA. $\sigma'_1, \chi'_1, [u_1/x]e_1 \Downarrow^{\mathtt{S}} u, \sigma''_1, \chi''_1, \mathtt{T}_2$; this follows by evaluation rule **let/1**.

BBB. $\sigma'_3 \sqsupseteq \sigma''_1|_{\mathtt{gen(T}_2\mathtt{)}}$. By property B, we know that $\sigma_3 \sqsupseteq \sigma''_1|_{\mathtt{gen(T}_2\mathtt{)}}$. Since $\sigma'_3 \sqsupseteq \sigma_3$, $\sigma'_3 \sqsupseteq \sigma''_1|_{\mathtt{gen(T}_2\mathtt{)}}$.

$\mathtt{dom}(\sigma'_3) \supseteq \mathtt{dom}(\sigma'_1)$. By property B, we know that $\mathtt{dom}(\sigma_3) \supseteq \mathtt{dom}(\sigma''_1|_{\mathtt{gen(let\ T_1\ T_2)}}) \cup \mathtt{dom}(\sigma_1)$. This simplies that $\mathtt{dom}(\sigma_3) \supseteq \mathtt{dom}(\sigma''_1|_{\mathtt{gen(T}_1\mathtt{)}}) \cup \mathtt{dom}(\sigma_1) = \mathtt{gen(T}_1\mathtt{)} \cup \mathtt{dom}(\sigma_1)$. We know that $\mathtt{dom}(\sigma'_1) = \mathtt{dom}(\sigma_1) \cup \mathtt{gen(T}_1\mathtt{)}$. Therefore, $\mathtt{dom}(\sigma_3) \supseteq \mathtt{dom}(s'_1)$, since $\mathtt{dom}(\sigma'_3) \supseteq \mathtt{dom}(\sigma_3)$, $\mathtt{dom}(\sigma'_3) \supseteq \mathtt{dom}(\sigma'_1)$ is true

CCC. $\chi' \subseteq \mathtt{dom}(\sigma'_1)$. We know by property C that $\chi \subseteq \mathtt{dom}(\sigma_1)$. We know that $\chi' \subseteq \chi \cup \mathtt{gen(T}_1\mathtt{)}$. It follows that $\chi' \subseteq \mathtt{dom}(\sigma_1) \cup \mathtt{gen(T}_1\mathtt{)}$. Since $\mathtt{dom}(\sigma'_1) = \mathtt{dom}(\sigma_1) \cup \mathtt{gen(T}_1\mathtt{)}$, it follows that $\chi' \subseteq \mathtt{dom}(\sigma'_1)$.

DDD. $[u_1/x]e_2 \uparrow \sigma'_1 \cdot \chi' = [u_1/x]e_2 \uparrow \sigma'_3 \cdot \chi'$. We know by property 44, that $u_1 \uparrow \sigma'_1 \cdot \chi' = u_1 \uparrow \sigma'_3 \cdot \chi'$. Thus all we need to show is that $e_2 \uparrow \sigma'_1 \cdot \chi' = e_2 \uparrow \sigma'_3 \cdot \chi'$. We know, by property D, that $e_2 \uparrow \sigma_1 \cdot \chi = e_2 \uparrow \sigma_3 \cdot \chi$. By This follows by properties of lifting.

EEE. $\sigma'_3, \mathtt{T}_2, \chi' \overset{\mathtt{S}}{\curvearrowright} \sigma''_3, \mathtt{T}'_2, \chi''$; this follows by evaluation rule **let/c.p.**.

FFF. $[u_1/x]e_2 \uparrow \sigma'_3 = [v_1/x]e_4 \uparrow \sigma'_2$, We know, by property 22, that $u_1 \uparrow \sigma'_3 = v_1 \uparrow \sigma'_2$. Thus all we need to show is that $e_2 \uparrow \sigma'_3 = e_4 \uparrow \sigma'_2$. By property F and properties of lifting, we know that $e_2 \uparrow \sigma_3 = e_4 \uparrow \sigma_2$.

GGG. $\sigma'_2, \chi'_2, [v_1/x]e_4 \Downarrow^{\mathtt{S}}_{\emptyset} v, \sigma''_2, \chi''_2, \mathtt{T}_4$; this follows by evaluation rule **let/2**.

By induction hypothesis, we therefore conclude that the following hold.

111. $\mathtt{T}'_2 \uparrow \sigma''_3 = \mathtt{T}_4 \uparrow \sigma''_2$,

222. $u \uparrow \sigma''_3 = v \uparrow \sigma''_2$,

444. $u \uparrow \sigma''_1 \cdot \chi'' = u \uparrow \sigma''_3 \cdot \chi''$.

By using properties 11,22,33,44 and 111,222,333,444, we will now show the We need to show that the following.

1. $\mathtt{let}\ \mathtt{T}'_1\ \mathtt{T}'_2 \uparrow \sigma''_3 = \mathtt{let}\ \mathtt{T}_3\ \mathtt{T}_4 \uparrow \sigma''_2$.

   We know, by property 111, that $\mathtt{T}'_2 \uparrow \sigma''_3 = \mathtt{T}_4 \uparrow \sigma''_2$. Thus all we have to show is that $\mathtt{T}'_1 \uparrow \sigma''_3 = \mathtt{T}_3 \uparrow \sigma''_2$. We know, by property 11, that $\mathtt{T}'_1 \uparrow \sigma'_3 = \mathtt{T}_3 \uparrow \sigma'_2$. Since $\sigma''_2 \sqsupseteq \sigma'_2$ and since $\mathtt{dom}(\sigma''_2) \setminus \mathtt{dom}(\sigma'_2) \cap \mathtt{dom}(\mathtt{T}_3) = \emptyset$, we know that $\mathtt{T}_3 \uparrow \sigma'_2 = \mathtt{T}_3 \uparrow \sigma''_2$. Similarly since the difference between σ'_3 and σ''_3 does not intersect with that the domain of $\mathtt{T}_1$ $\mathtt{T}'_1 \uparrow \sigma'_3 = \mathtt{T}'_1 \uparrow \sigma''_3$. [2].

2. $u \uparrow \sigma''_3 = v \uparrow \sigma''_2$. This directly follows by 222.

3. $u \uparrow \sigma''_1 \cdot \chi'' = u \uparrow \sigma''_3 \cdot \chi''$. This directly follows by 444.

■

[2]This must be made precise