

A Library for Self-Adjusting Computation

Umut Acar¹ Guy Blelloch² Matthias Blume¹
Robert Harper² Kanat Tangwongsan²

Abstract

We present a Standard ML library for writing programs that automatically adjust to changes to their data. The library combines modifiable references and memoization to achieve efficient updates. We describe an implementation of the library and apply it to the problem of maintaining the convex hull of a dynamically changing set of points. Our experiments show that the overhead of the library is small, and that self-adjusting programs can adjust to small changes three-orders of magnitude faster than recomputing from scratch. The implementation relies on invariants that could be enforced by a modal type system. We show, using an existing language, abstract interfaces for modifiable references and for memoization that ensure the same safety properties without the use of modal types. The interface for memoization, however, does not scale well, suggesting a language-based approach to be preferable after all.

Key words: incremental computation, selective memoization, change propagation, computational geometry, convex hulls, quickhull

In many application domains, such as simulations systems, robotics, language environments, it is important for computations to adjust to external changes to their data. This ability can enable a whole new class of computations that has not been previously possible [7,8], and enable performing complex computations in real-time by taking advantage of the fact that most changes are small.

The problem of processing a small change to the input of a program efficiently has been studied extensively in the programming-languages community. This is known as incremental computation. The goal has been to devise general-purpose techniques for writing programs that can adjust to changes. The two key techniques are dynamic dependence graphs and memoization. Dynamic dependence graphs (DDGs) [2,3] generalize static dependence graphs [10], by enabling change propagation to update the dependences. Memoization [20,14,13,3] relies on remembering the results of function calls and re-using them when possible.

Although both of these techniques are general purpose, they yield efficient updates only for certain kinds of input changes and certain kinds of computations.

¹ {umut,blume}@tti-c.org. Toyota Technological Institute at Chicago.

² {blelloch,rwh,ktangwon}@cs.cmu.edu. Carnegie Mellon University.

In previous work [2,3], we point out the limitations of each technique and claim that they can be combined to improve performance substantially. In this paper, we present an SML library that combines DDGs and memoization. Using the library, the programmer can transform an ordinary purely functional program into a *self-adjusting* program by making small changes to the code.

Self-adjusting programs adjust to any external change whatsoever to their data. As a self-adjusting program executes, the run-time system builds a *dynamic dependence graph (or DDG)* that represents the relationship between computation and data. The nodes represent data and the edges represent the relationship between them. Each edge is tagged with a closure that determines how its destination is computed from its source. After a self-adjusting program completes its execution, the user can change any computation data (*e.g.*, the inputs) and update the output by performing a change propagation. This change-and-propagate step can be repeated. The change-propagation algorithm updates the computation by mimicking a from-scratch execution by re-executing the closures of edges whose sources have changed. When re-executing a closure, the algorithm can re-use, via memoization, edges and nodes that the previous execution of that closure created. When an edge is re-executed, the algorithm discards the unused edges and nodes belonging to the prior execution, because these elements no longer belong to the computation.

To demonstrate the effectiveness of our approach, we consider an application from computational geometry [9]. We implement the quick hull algorithm for computing convex hulls and perform an experimental evaluation. Our experiments show that the overhead the library is small and that the self-adjusting programs can adjust to changes significantly faster than recomputing from scratch. For quick-hull algorithm, our experiments show a near linear-time gap between change propagation and recomputing from scratch. For the input sizes we consider, change propagation can be three orders of magnitude faster than recomputing from scratch.

1 The Library

We present a library that combines modifiable references and memoization. The library ensures safe use of modifiable references but it does not ensure safe use of memoization primitives. In Section 4, we present combinators that ensure safe use of memoization. In principle, the combinators for memoization can be incorporated with the library considered in this section. We do not do this, however, because of scalability issues discussed in Section 4.

We first present a high-level description of the underlying model (a full description is out of the scope of this paper). We then describe the interface for the library and present the full code for the core of the implementation.

1.1 The underlying model

Change propagation must detect changes to data so that the computations that process those data can be re-executed. The frames of reference with respect to which

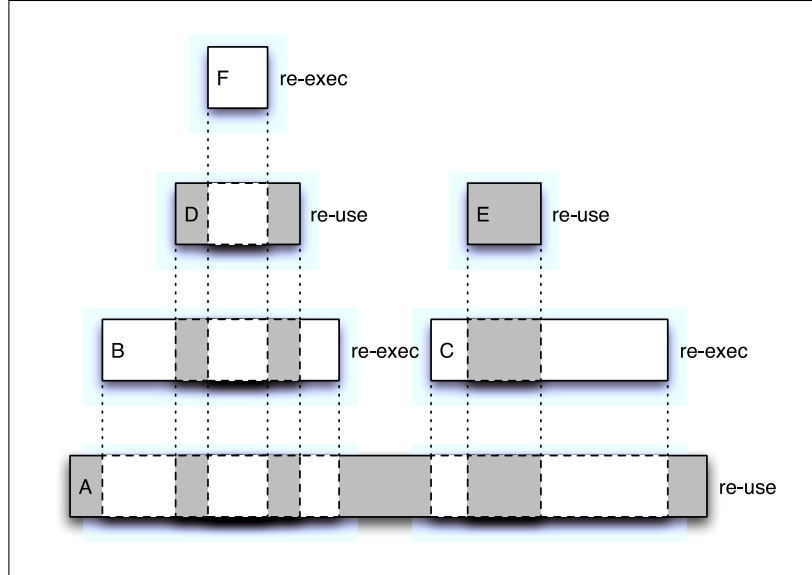


Fig. 1. Intervals of alternating re-use and re-execution

changes are tracked are called *modifiable references*. We will often abbreviate and refer to modifiable references as *modifiabes*. A modifiable can be thought of as a memory location with a fixed address but variable contents. We call computations that are guaranteed to yield fixed results *stable* and other computations—those that may produce variable data—*changeable*. The result of each changeable computation is recorded in a modifiable, and only changeable computations may read the contents of modifiabes.

During propagation, when there are pending changes, the implementation “fast-forwards” to the earliest affected read operation and begins re-executing the corresponding changeable computation from that point on. The result of this computation ultimately writes to some modifiable r . Depending on the original value of r , a change (to the value contained in r) might get recorded. Notice that r itself, *i.e.*, the address of the location that contains the potentially changed value, stays fixed.

Re-executing the entire remainder of a changeable computation starting at an affected read can be very inefficient as significant portions of it might not truly depend on the changed value. We use *memoization of computations* to detect such situations dynamically: when a function is called with the exact same arguments as during the previous execution, then the part of the computation corresponding to this function call can be re-used. However, since a memoized computation might contain read operations that are affected by pending changes, one must propagate these changes into the result of a successful memo lookup.

Thus, there is a form of duality between change propagation and memoization: change propagation takes a part of a computation to be skipped and “carves out” the sub-intervals that need to be re-executed. Memoization takes a part of a computation to be re-executed and “carves out” intervals to be skipped. A simple example for this is shown in Figure 1: Change propagation tries to re-use as much as possible of the overall computation (interval A) but has to re-execute two of its parts

(intervals B and C), because these parts are controlled by affected read operations. During the re-execution of B a re-usable part (interval D) is discovered via memoization. Similarly, E can be re-used during the re-execution of C. However, there is a change that affects a read operation within D which forces the re-execution of sub-interval F which is nested within D.

By synergistically combining memoization and change propagation into *adaptive memoization*, we can also relax the re-use rule for memoization. This relaxation enables re-using a computation even when the values that the computation depends on differ. Consider a function call $f(X, y)$ where X represents all arguments except y . If we arrange for y to be read from a modifiable r_X (where r_X itself depends only on X) instead of being passed as an argument, then the value of y would not figure into the memo-lookup. Instead, a previous execution of the function call—even for a different value of y —can be reused and adjusted to the new value of y by change propagation. Note that this differs from partial evaluation, because change propagation will take advantage of the similarity between the old and the new values for y .

1.2 The library interface

Figure 2 shows the interface to the library.

The `BOXED.VALUE` module supplies functions for operating on boxed values. We used boxes (or tagged values) to support constant-time equality tests, which are necessary for efficient memoization [3]. The `new` function creates a boxed value by associating a unique integer *index* with a given value. The `eq` function compares two boxed values by comparing their indices. The `valueOf` and `indexOf` functions return the value and the index of a boxed value, respectively. The boxed value module may be extended with functions for creating boxed values for a type. Such type-specific functions must be consistent with the equality of the underlying types. For example, the function `fromInt` may assign the index i to integer i .

The `COMBINATORS` module defines modifiable references 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 [6]. Any non-trivial changeable computation reads one or more other modifiables and performs calculations based on the values read.

Values of type α `cc` representing changeable computations are constructed using `write`, `read`, and `mkLiftCC`. The `modref` function executes a given computation on a freshly generated modifiable before returning that modifiable as its result. The `write` function creates a trivial computation which merely writes the given value into the underlying modifiable. To avoid unnecessary propagation, old and new values are compared for equality at the time of `write` using the equality function provided. The `read` combinator, which we will often render as $\square \rightarrow$ in infix

```

signature BOXED_VALUE = sig
  type index
  type  $\alpha$  t

  val init: unit  $\rightarrow$  unit
  val new:  $\alpha \rightarrow \alpha$  t
  val eq:  $\alpha$  t *  $\alpha$  t  $\rightarrow$  bool
  val valueOf:  $\alpha$  t  $\rightarrow$   $\alpha$ 
  val indexOf:  $\alpha$  t  $\rightarrow$  index
  val fromInt: int  $\rightarrow$  int t
end
structure Box: BOXED_VALUE = struct ... end

signature COMBINATORS = sig
  eqtype  $\alpha$  modref
  type  $\alpha$  cc

  val modref:  $\alpha$  cc  $\rightarrow$   $\alpha$  modref
  val write: ( $\alpha$  *  $\alpha \rightarrow$  bool)  $\rightarrow$   $\alpha \rightarrow \alpha$  cc
  val read:  $\beta$  modref * ( $\beta \rightarrow \alpha$  cc)  $\rightarrow$   $\alpha$  cc

  val mkLift: ( $\alpha$  *  $\alpha \rightarrow$  bool)  $\rightarrow$ 
    (Box.index list *  $\alpha$ )  $\rightarrow$  ( $\alpha$  modref  $\rightarrow$   $\beta$ )  $\rightarrow$   $\beta$ 
  val mkLiftCC: (( $\alpha$  *  $\alpha \rightarrow$  bool) * ( $\beta$  *  $\beta \rightarrow$  bool))  $\rightarrow$ 
    (Box.index list *  $\alpha$ )  $\rightarrow$  ( $\alpha$  modref  $\rightarrow$   $\beta$  cc)  $\rightarrow$   $\beta$  cc

  (** Meta Operations **)
  val init: unit  $\rightarrow$  unit
  val change: ( $\alpha$  *  $\alpha \rightarrow$  bool)  $\rightarrow$   $\alpha$  modref  $\rightarrow$   $\alpha \rightarrow$  unit
  val deref:  $\alpha$  modref  $\rightarrow$   $\alpha$ 
  val propagate: unit  $\rightarrow$  unit
end
structure C: COMBINATORS = struct ... end

```

Fig. 2. Signatures for boxed values and combinators.

notation, takes an existing modifiable reference together with a receiver for the value read. The result of the `read` combinator is a computation that encompasses the process of reading from the modifiable, a calculation that is based on the resulting value, and a continuation represented by another changeable computation. Calculation and continuation are given by body and result of the receiver.

Functions `mkLift` and `mkLiftCC` handle adaptive memoization, *i.e.*, memoization based on partially matching function arguments. With ordinary memoization, a memo table lookup for a function call will fail whenever there is no precise match for the entire argument list. As explained above, the idea behind adaptive memoization is to distinguish between strict and non-strict arguments and base memoization on strict arguments only. By storing *computations* (as opposed to mere return values) in the memo table, a successful lookup can then be adjusted to any changes in non-strict arguments using the change-propagation machinery. Since change propagation relies on read operations on modifiables, the memoized function has to access its non-strict arguments via such modifiables. The memo table, indexed by just the strict part of the original argument list, remembers the modifiables set aside

for non-strict arguments as well as the memoized computation.

Given the strict part of the argument list, a *lift operation* maps a function of type $\alpha \text{ modref} \rightarrow \beta$ to a function of type $\alpha \rightarrow \beta$ where α is the type of the non-strict argument.³ Our `mkLift` and `mkLiftCC` combinators create lift operations for ordinary and changeable computations from appropriately chosen equality predicates for the types involved. The strict part of the argument list is represented by an index list, assuming a 1 – 1 mapping between values and indices, *e.g.*, the one provided by the `BOXED_VALUE` interface. Not shown here, our library also contains `mkLift2`, `mkLiftCC2`, `mkLift3`, `mkLiftCC3` and so on to support more than one non-strict argument.

When its memo table lookup fails, a lifted function creates fresh modifiabls containing its non-strict arguments, executes its body, and stores both the computation and the modifiabls into the memo table. Computations are memoized by recording their return value and a representation of their dynamic dependence graph (DDG) using time stamps [2]. A successful memo lookup finds modifiabls and a computation. The lifted function then writes its current non-strict arguments into their respective modifiabls, and lets change propagation adjust the computation to the resulting changes.

It is the responsibility of the programmer to ensure that all applications of lift functions specify the strict and non-strict variables accurately. For correctness, it is required that all free variables of a memoized expression are specified as strict or non-strict. The classification of a variable as strict or non-strict does not affect correctness but just performance. In previous work [3], we proposed *selective memoization* techniques for memoizing expressions in a safe manner based on *branches*.

The `COMBINATORS` module also supplies *meta operations* for inspecting and changing the values stored in modifiabls and performing change propagation. The `change` function is similar to `write` function. It changes the underlying value of the modifiable to a new value—this is implemented as a side effect. The `propagate` function runs the change-propagation algorithm. Change propagation updates a computation based on the changes issued since the last execution or the last change propagation. The meta operations should only be used at the top level—the library guarantees correct behavior only in the cases that meta operations are not used inside the program.

1.3 Implementation

Appendix A gives the code for an implementation of the library. The implementation consists of the following modules: `boxes`, `combinators`, `memo tables`, `modifiabls`, `order-maintenance data structure`, and `priority queues`. We give the complete code for the core part of the library consisting of the `modifiabls`, and the `combinators`.

³ In the actual library, arguments appear in a different order to make idiomatic usage of the interface by actual code more convenient.

2 Self-Adjusting Programming

This section describes how to write self-adjusting programs using the library presented in Section 1. As examples, we consider a function for filtering elements out of a list and a function for combining the values in a list with a binary operator. Section 3 describes how to implement a self-adjusting version of the quick-hull algorithm using these functions.

2.1 Modifiable Lists

Figure 3 shows signature and code for a module implementing modifiable lists. Modifiable lists are similar to ordinary lists—the difference is that the tail of a modifiable lists cell is stored inside of a modifiable. Modifiable lists enable the user to change their contents by changing the values stored in the tail elements through the `change` meta operation.

The modifiable list library provides the functions `write`, `eq`, `lengthLessThan`, `filter`, and `combine`. The `eq` function tests *shallow equality* of the two lists. The `write` function specializes the `write` function of the `COMBINATOR` module for modifiable lists. The `lengthLessThan` function tests if the list is shorter than the given value. This function is straightforward to implement; we therefore omit the code. The `filter` and the `combine` functions are discussed in the following sections.

2.2 The Transformation

The programmer can transform an ordinary program into a self-adjusting program in two steps. The first step makes the program *adaptive* by placing values into modifiables; this is described in detail in previous work [2]. The second step adds memoization. To maximize result reuse, memoization should be applied at as fine a granularity as possible. Performance, however, suffers if many memoized computations share the same key. The programmer can avoid this problem by making sure that each memoized function call is uniquely identified by the values of its strict arguments.

As an example, consider the `filter` function. Figure 4 shows the code for ordinary `filter` (left) and its self-adjusting version (right). The function takes a test function `f` and a list `l` and returns the list of the elements for which `f` is true. The first step of the transformation involves changing the input list to a modifiable list and placing `modref`, `read`, `write` functions appropriately [2]. In the second step, we memoize `filter` by considering its two branches. Since the `NIL` branch performs trivial work it need not be memoized. For the `CONS` branch we create a lift operator using `mkLift` and memoize the branch, treating `h` as strict and the contents of `t` (bound to `ct`) as non-strict. Since the lift operator places `ct` into another modifiable, an extra `read` is required.

```

signature MOD_LIST = sig
  datatype  $\alpha$  modcell = NIL | CONS of ( $\alpha$  *  $\alpha$  modcell C.modref)
  type  $\alpha$  t =  $\alpha$  modcell C.modref

  val eq:  $\alpha$  Box.t modcell *  $\alpha$  Box.t modcell  $\rightarrow$  bool
  val write:  $\alpha$  Box.t modcell  $\rightarrow$   $\alpha$  Box.t modcell C.cc
  val lengthLessThan : int  $\rightarrow$  ( $\alpha$  t)  $\rightarrow$  bool C.modref
  val filter: ( $\alpha$  Box.t  $\rightarrow$  bool)  $\rightarrow$   $\alpha$  Box.t t  $\rightarrow$  ( $\alpha$  Box.t t)
  val combine:( $\alpha$  Box.t* $\alpha$  Box.t $\rightarrow$  $\alpha$  Box.t) $\rightarrow$ ( $\alpha$  Box.t t $\rightarrow$  $\alpha$  Box.t C.modref)
end

structure ML:MOD_LIST = struct
  datatype  $\alpha$  modcell = NIL | CONS of ( $\alpha$  *  $\alpha$  modcell C.modref)
  type  $\alpha$  t =  $\alpha$  modcell C.modref

  infix  $\square\rightarrow$  val op  $\square\rightarrow$  = C.read

  fun eq (a,b) =
    case (a,b) of
      (NIL,NIL)  $\Rightarrow$  true
    | (CONS(ha,ta), CONS(hb,tb))  $\Rightarrow$  Box.eq(ha,hb) andalso (ta=tb)
    | _  $\Rightarrow$  false

  fun write c = C.write eq c
  fun lengthLessThan n l = ...
  fun filter f l = ... (* See Figure 4 *)
  fun combine binOp l = ... (* See Figure 5 *)
end

```

Fig. 3. The signature for modifiable lists and an implementation.

<pre> filter: (α Box.t \rightarrow bool) \rightarrow (α Box.t list) \rightarrow (α Box.t list) fun filter f l = let fun filterM c = case c of nil \Rightarrow nil h::t \Rightarrow if (f h) then (CONS(h,filterM t)) else filterM t in filterM l end </pre>	<pre> filter: (α Box.t \rightarrow bool) \rightarrow (α Box.t ML.t) \rightarrow (α Box.t list) fun filter f l = let val lift = C.mkLift ML.eq fun filterM c = case c of ML.NIL \Rightarrow <u>ML.write</u> ML.NIL ML.CONS(h,t) \Rightarrow t $\square\rightarrow$ (<u>fn ct \Rightarrow lift ([Box.indexOf h],ct) (fn t \Rightarrow if (f h) then <u>ML.write</u> (ML.CONS(h,C.modref (t $\square\rightarrow$ filterM))) else t $\square\rightarrow$ filterM)) in <u>C.modref</u> (l $\square\rightarrow$ filterM) end </u></pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 4. The self-adjusting filter function.

2.3 Program Stability

Even though any purely functional program can be methodically transformed into a self-adjusting program, not all such programs perform efficiently under changes. The performance is determined by how *stable* the program is under that change.

In other work [1,4], we formalize stability and prove stability bounds for a number of algorithms. A precise treatment of stability is beyond the scope of this paper, but we would like to give some intuition. The stability of a program can

```

fun combine binOp l =
let fun halfList l =
    let val hash = Hash.new ()
        fun pairEqual ((b1,c1),(b2,c2)) = Box.eq(b1,b2) andalso ML.eq(c1,c2)
        val writePair = C.write' pairEqual
        val lift = C.mkLiftCC (ML.eq,ML.eq)

        fun half c =
        let fun sumRun(v,ML.NIL) = writePair (v,c)
            | sumRun(v,ML.CONS(h,t)) =
                t □→ (fn ct ⇒
                    if hash(Box.indexOf h) = 0 then writePair (binOp(v,h),ct)
                    else sumRun(binOp(v,h),ct))
        in case c of
            ML.NIL ⇒ ML.write ML.NIL
          | ML.CONS(h,t) ⇒ t □→ (fn ct ⇒
                lift ([Box.indexOf h],ct) (fn t ⇒ t □→ (fn ct ⇒
                    let val p = C.modref (sumRun (h,ct))
                    in p □→ (fn (v,ct') ⇒ ML.write (ML.CONS(v, C.modref (half ct'))))
                    end)))
        end
    in C.modref (l □→ (fn c ⇒ half c)) end

fun comb l = ML.lengthLessThan 2 l □→ (fn b ⇒
    if b then l □→ (fn c ⇒
        case c of ML.NIL ⇒ raise EmptyList
        | ML.CONS(h,_) ⇒ C.write h
    else comb (halfList l))
in C.modref (comb l) end

```

Fig. 5. Self-adjusting list combine.

be determined by comparing the executions of the program on inputs before and after the change takes place. An easy stability criterion is to look at the difference between all data computed by two executions—a large difference means that the program is not stable. A more precise analysis is based on computing the distance between the (execution) traces instead of just data. The *trace* is defined as the set of executed lift functions tagged with their strict arguments. The distance between two traces is total execution time of the function calls in the symmetric set difference of the two traces—we refer to such function calls as affected. A call is *affected* if there isn't a call in the other trace that has the same strict arguments. Under certain conditions, it can be shown that the trace distance and the time for adjusting to a change are equal [1]. For example, it can be shown that the `filter` function adjusts to a deletion/insertion in constant time by showing that the trace distance under this change is constant.

As an example, consider summing the values in a list. The straightforward technique of traversing the list while maintaining the partial sum of all elements visited is not stable, because deleting/inserting a key will change all the partial sums that include that key. We now provide a stable solution to this problem in a more general setting.

A fundamental primitive for computing with lists is a `combine` (*a.k.a.*, `fold` or

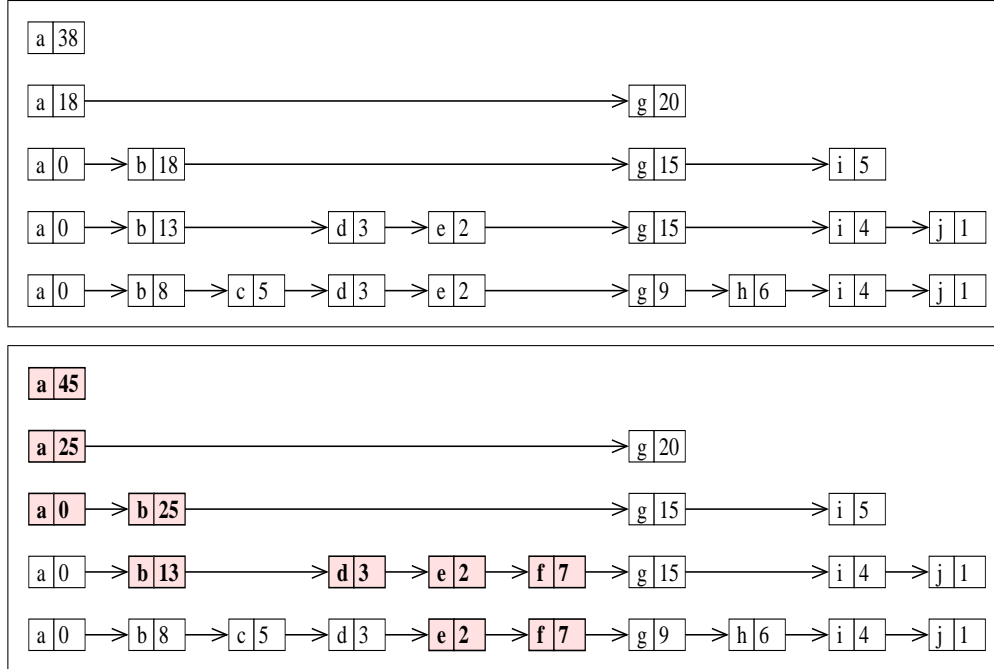


Fig. 6. The lists at each round before and after inserting the key f with value 7.

reduce) primitive that takes a binary operation and a list, and combines the values in the list by applying the binary operation. By choosing the binary operation to be applied, **combine** can perform various operations on lists such as computing the sum of all elements, finding the minimum or maximum element, or finding the longest increasing sequence in a list. To provide a stable solution to this problem, we adopt a randomized approach. The idea is to halve the input list into smaller and smaller lists until the list is a singleton. To halve a list, choose a randomly selected subset of the list and delete the chosen elements from the list. Deleting an element incorporates its value to the closest surviving element to the left. Note that a deterministic approach, where, for example, every other element is deleted is not stable, because deleting/inserting an element can cause a large change by shifting the positions of many elements by one. Figure 5 shows the code for the approach. We assume that the given binary operator is associative—commutativity is not required. For randomization, we use a random hash function [22] that returns zero or one with probability a half.

As an example, Figure 6 shows an execution of **combine** that computes the sum of the values in the lists $[(a,0), (b,8), (c,5), (d,3), (e,2), (g,9), (h,6), (i,4), (j,1)]$ and $[(a,0), (b,8), (c,5), (d,3), (e,2), (\mathbf{f},7), (g,9), (h,6), (i,4), (j,1)]$, which differ by the key f , with value 7. The keys are a, \dots, k . The second execution can be obtained from the first by inserting f and performing a change propagation. During change propagation, only the shaded cells will be recomputed. Based on this intuition and by establishing an isomorphism to Skip List [19], it can be shown that a deletion/insertion requires logarithmic time in the size of the list [1].

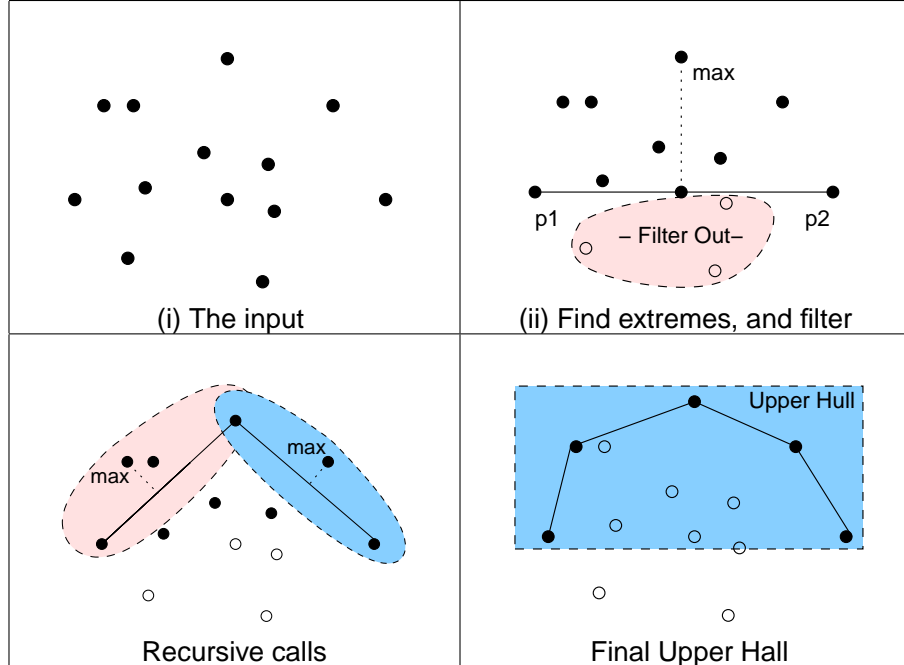


Fig. 7. The Quick-Hull Algorithm

3 Self-Adjusting Quick Hull

The *convex hull* of a set of points is the smallest polygon enclosing these points. We describe a self-adjusting version of the quick hull algorithm for computing convex hulls and present an experimental evaluation. Convex hulls have been studied both in the ordinary and the dynamic (incremental) settings [12,18,17,16,9,5].

Figure 8 shows the code for the quick hull algorithm, as given by the transformation technique described in Section 2.2. As is standard in the literature, we only compute the upper hull—the same algorithm can compute the complete hull by using a slightly different distance function. The code relies on modifiable lists and a module that supplies the geometric primitives specified by the POINT signature—it is straightforward to give an implementation for this signature.

Figure 7 illustrates how the algorithm works. The algorithm first computes the leftmost (*min*) and the rightmost (*max*) points in the input, and calls `split` with the line (*min*,*max*). The `split` function takes the line (*p1*,*p2*) and filters out the points below that line. The function then finds the point, *max*, that is farthest away from the line and performs two recursive calls with the lines (*p1*,*max*) and (*max*,*p2*), respectively. The algorithm uses the `combine` function to find the leftmost and the rightmost points, as well as the point farthest away from a line.

To evaluate the effectiveness our approach, we measure the following quantities for a given input I .

- Time for running the non-self-adjusting quick hull, denoted $T_{\text{verifier}}(I)$.
- Time for running the self-adjusting quick hull, $T_{\text{initialRun}}(I)$.
- Average time for a deletion/insertion, denoted $T_{\text{avgDelIns}}(I)$: This quantity is mea-

```

signature POINT =
sig
  type t
  val toLeft : t * t → bool
  val toRight : t * t → bool
  val leftTurn : t * t * t → bool
  val distToLine: t * t → t → real
  val fromCoordinates : real*real → t
  val toCoordinates : t → real*real
end
structure P: POINT = struct ... end

structure QuickHull =
struct
  structure C = Comb
  infix □→ val op □→ = C.read

  fun split (rp1, rp2, ps, hull) =
    let val lift = C.mkLiftCC2 (ML.eq,ML.eq,ML.eq)

        fun splitM (p1, p2, ps, hull) =
          let fun select p =
              P.distToLine (Box.valueOf p1,Box.valueOf p2) (Box.valueOf p) > 0.0
              val l = ML.filter select ps
              in l □→ (fn cl ⇒
                  case cl of
                    ML.NIL ⇒ ML.write (ML.CONST(p1,hull))
                  | ML.CONST(h,t) ⇒ hull □→ (fn chull ⇒
                      lift ([Box.indexOf p1,Box.indexOf p2],cl,chull) (fn (l,hull) ⇒
                          let val (v1,v2) = (Box.valueOf p1,Box.valueOf p2)
                              fun select (a,b) =
                                  if P.distToLine (v1, v2) (Box.valueOf a) >
                                      P.distToLine (v1, v2) (Box.valueOf b))
                                  then a
                                  else b
                              val rmax = ML.combine select l
                              val rest = C.modref (rmax □→ (fn max ⇒ splitM (max,p2,l,hull)))
                              in rmax □→ (fn max ⇒ splitM (p1,max,l,rest)) end)))
                          end
                    end
              in rp1 □→ (fn p1 ⇒ rp2 □→ (fn p2 ⇒ splitM (p1,p2,ps,hull))) end

  fun qhull l =
    C.modref ((ML.lengthLessThan 2 l) □→ (fn b ⇒
      if b then ML.write ML.NIL
      else let fun select f (a,b) =
          if f (Box.valueOf a, Box.valueOf b) then a
          else b

          val min = ML.combine (select P.toLeft) l
          val max = ML.combine (select P.toRight) l
          val h = C.modref (max □→ (fn m ⇒
              ML.write (ML.CONST(m,C.modref (ML.write ML.NIL))))))
          in split (min, max, l, h) end))
    end
end

```

Fig. 8. Self-adjusting Quick Hull.

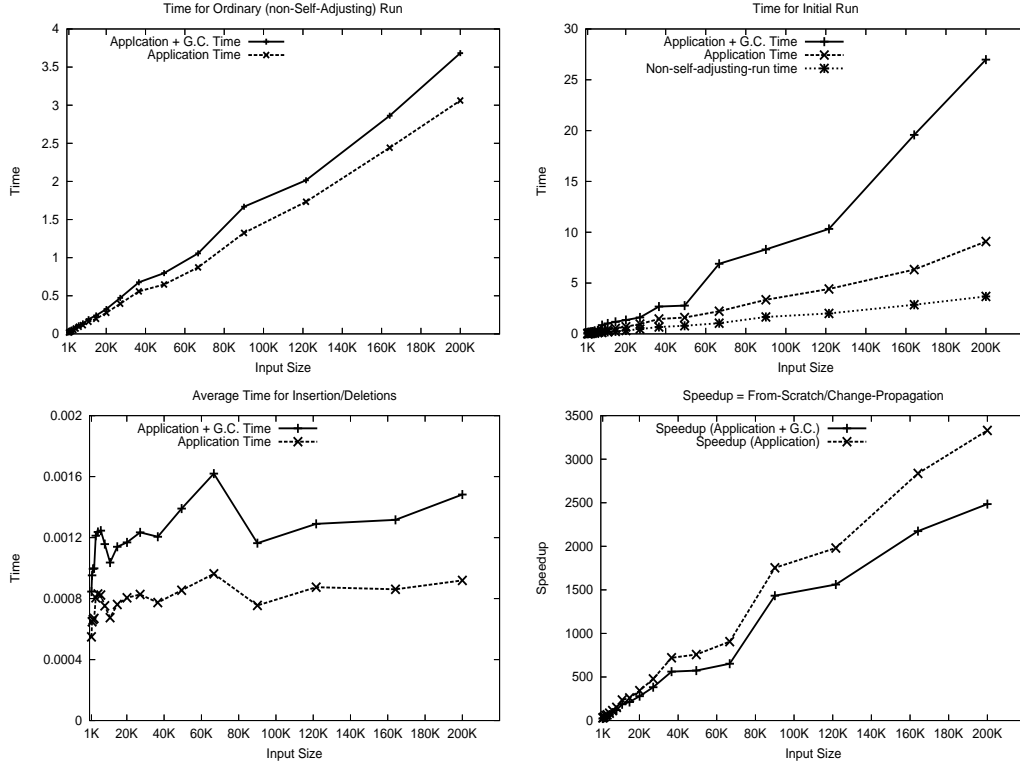


Fig. 9. Timings for quick hull.

sured by running a delete-propagate-insert-propagate step for each element and computing the average time. Each step deletes an element, runs change propagation, inserts the element back, and runs change propagation.

Figure 9 shows the timings for input sizes up to to 200K (*i.e.*, 200000). The inputs are generated randomly. An input of size n is produced by randomly selecting n points from a $10n \times 10n$ square. The experiments are run on a 2.7GHz Power Mac G5 with 4GB of memory; the machine runs Mac OS X. We use the MLton [21] compiler with option `ram-slop 1`. This option directs the compiler to use all the available memory available on the system (MLton, however, can only allocate a maximum of two Gigabytes).

The experiments show that the overhead $T_{\text{initialRun}}(I)/T_{\text{verifier}}(I)$ of our library is no more than six, including time for garbage collection. When the time for garbage collection is excluded the overhead is about a factor of two. During initial run, the ratio of garbage-collection time to the total execution time increases as the input size increases. At its peak, garbage-collection time constitutes 67% of the total execution time. One reason for this can be that self-adjusting programs build and store the dynamic dependence graphs and the memo tables during the initial execution. Although the ratio for garbage collection can be relatively high, this is less of a problem for the initial run, because an initial run takes place only once—the computation adjusts to changes via change propagation.

The bottom left figure shows the average time for a deletion/insertion via change propagation. This quantity increases very slowly with the input size. The time for

garbage collection constitute no more than 40% of the total execution time even for large inputs. It peaks for the input size of approximately 64K.

To get a sense of how fast change propagation is, we compare the time for change propagation to the time for recomputing from scratch (by running an ordinary, non-adaptive, implementation of the quick-hull algorithm). We define the *speedup* obtained by change propagation as the time for recomputing from scratch divided by the time for performing a change propagation after an insertion or deletion. Figure 9 shows the average speedup computed as $T_{\text{verifier}}(I)/T_{\text{avgDelIns}}(I)$ with and without the time for garbage collection. The experiments show that the speedup increases to more than a factor of two thousand as the input size increases.

4 Emulating a language using a library

So far we have considered a setting where the keys used for memo lookups are completely explicit, leaving the programmer in full control—and with the full burden. This burden can become cumbersome when one wants to exploit the fact that only parts of a given input contribute to the result of the computation. Therefore, a different (and complementary) technique is *selective memoization* using an implementation that automatically tracks the relevant parts of the input. Like self-adjusting computation, selective memoization imposes constraints on programs.

In previous work we discussed the design of a language with modal types that can enforces these constraints at compile time [3]. A tiny program fragment in such a language could be the following:

```
mfun f (x, y, z) = mif x < 0 then let! yv = y in return g(yv) end
                               else let! zv = z in return h(zv) end
```

Keyword `mfun` introduces a memoizing function. Initially, all of its arguments—here `x`, `y`, and `z`—are considered *resources*. During execution, the function keeps track of how it uses its resources, recording this information in a trace called a *branch*. Memoization is based on that branch.

In the example, no call of `f` will depend on both `y` and `z` simultaneously. Moreover, the decision between `y` and `z` is based just on the sign of `x`; `x`'s magnitude is ignored. So after having evaluated `f(-1, 2, 3)` once, a call of `f(-2, 2, 4)` would result in a successful memo table lookup.

The part of the computation that actually gets memoized is represented by the expression under `return`. To facilitate this memoization, each instance of a memoizing function has its own memo table. The table is indexed by the value of the branch at the time of the `return`. A branch describes the execution path through the function and records for every decision point which direction was taken. Moreover, it records the values that were obtained by reading resources (using `let!`).

For this to work, the code must have the property that by the time execution reaches `return`, the function must “be done” with all its resources. To enforce this, the expression under `return` must not mention (and therefore cannot depend

on) any resources. With a language specifically designed with this in mind, the compiler can easily check such a constraint. In contrast, a library has to rely on the existing type system of its host language, in our case SML. It is tempting to try using an abstract constructor for resource types, providing only a limited set of operations for its instances. Here is an attempt at such a design:

```

type  $\alpha$  resource
val bang:  $\alpha$  resource  $\rightarrow$   $\alpha$ 
val mless: int resource * int  $\rightarrow$  bool resource
val mif: bool resource * (unit  $\rightarrow$   $\beta$ ) * (unit  $\rightarrow$   $\beta$ )  $\rightarrow$   $\beta$ 
...
val return: (unit  $\rightarrow$   $\alpha$ )  $\rightarrow$   $\alpha$ 

```

The body of the above example could be rendered as:

```

mif (mless (x, 0), fn ()  $\Rightarrow$  let yv = bang y in return (fn ()  $\Rightarrow$  g(yv) end)
      fn ()  $\Rightarrow$  let zv = bang z in return (fn ()  $\Rightarrow$  h(zv) end))

```

Unfortunately, this is a false start. The ML type system can restrict the set of operations that are applicable, but it does nothing to prevent the mere mention of a variable. Thus, it does not stop anyone from incorrectly “simplifying” the code by applying bang to a resource under a return:

```

mif (mless (x, 0), fn ()  $\Rightarrow$  return (fn ()  $\Rightarrow$  g(bang y))
      fn ()  $\Rightarrow$  return (fn ()  $\Rightarrow$  h(bang z))

```

Still, it is possible to construct a library which enforces the required invariant. The trick is to avoid ever naming resources, again using a technique that is similar to monadic programming [6]. Just like the state monad which manipulates a mapping from locations to values without naming this mapping, we will manipulate resources without naming them.

Selective memoization could easily be combined with the ideas described earlier: modifiable references and change propagation. For simplicity, however, this section focuses on the memoization part to demonstrate that supporting it using a library is possible but awkward, and that a better basis would require direct linguistic support (e.g., in form of the modal type system described in previous work).

4.1 Memo-computations

The library consists of combinators for constructing *memo-computations* (see Figure 10). A memo-computation is an abstract value c of type (α, ν, ρ) mc. It represents a program point within a memoizing function—which can be viewed as a delimited continuation up to the corresponding return. The continuation has access to resources of type α and to a non-resource “accumulator” of type ν . It eventually produces a result of type ρ . When it is time to return a value, a memoizing function simply discards its leftover resources and produces the current

```

type ( $\alpha$ ,  $\nu$ ,  $\rho$ ) mc
val return: ( $\alpha$ ,  $\nu$ ,  $\nu$ ) mc
val Do: ( $\nu \rightarrow \mu$ ) * ( $\alpha$ ,  $\mu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\alpha$ ,  $\nu$ ,  $\rho$ ) mc

val pair: (( $\alpha$  *  $\beta$ ) *  $\gamma$ ,  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\alpha$  * ( $\beta$  *  $\gamma$ ),  $\nu$ ,  $\rho$ ) mc
val unpair: ( $\alpha$  * ( $\beta$  *  $\gamma$ ),  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  (( $\alpha$  *  $\beta$ ) *  $\gamma$ ,  $\nu$ ,  $\rho$ ) mc
val dup: ( $\alpha$  * ( $\alpha$  *  $\beta$ ),  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\alpha$  *  $\beta$ ,  $\nu$ ,  $\rho$ ) mc
val swap: ( $\alpha$  * ( $\beta$  *  $\gamma$ ),  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\beta$  * ( $\alpha$  *  $\gamma$ ),  $\nu$ ,  $\rho$ ) mc
val swap2: ( $\alpha$  * ( $\beta$  * ( $\gamma$  *  $\delta$ )),  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\alpha$  * ( $\gamma$  * ( $\beta$  *  $\delta$ )),  $\nu$ ,  $\rho$ ) mc
val drop: ( $\beta$ ,  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\alpha$  *  $\beta$ ,  $\nu$ ,  $\rho$ ) mc

val Bang: ( $\beta$ , int *  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  (int *  $\beta$ ,  $\nu$ ,  $\rho$ ) mc
val If: ( $\beta$ ,  $\nu$ ,  $\rho$ ) mc * ( $\beta$ ,  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  (bool *  $\beta$ ,  $\nu$ ,  $\rho$ ) mc
val Case: ( $\beta$ ,  $\nu$ ,  $\rho$ ) mc * (( $\alpha$  *  $\alpha$  list) *  $\beta$ ,  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\alpha$  list *  $\beta$ ,  $\nu$ ,  $\rho$ ) mc
val memo: ( $\alpha$  * unit, unit,  $\rho$ ) mc  $\rightarrow$   $\alpha \rightarrow \rho$ 

```

Fig. 10. A combinator interface for memo-computations

accumulator as the answer.

The accumulator starts out with some constant value (here: unit). Any impact that resources have on its value is monitored and recorded as part of the branch. If we ignore the possibility of side effects, then all other values that could affect the state of the accumulator are invariant with respect to any given instance of a memoizing function. Therefore, this value does not need to be taken into account when performing memo lookups. Depending on the computations being performed, ν can be instantiated to arbitrary types. The programmer uses the Do combinator to specify computations as SML functions taking an accumulator to a new accumulator. Sub-computations specified using Do are subject to memoization.

We cannot use ordinary SML functions for operating on resources as doing so would “leak” resources, making it possible to violate the desired invariants. The alternative is to let the library provide combinators for operating on resources. Since we want to be able to deal with more than one resource at a time, we organize multiple resources into a *stack of resources* and impose the corresponding constraint that α is to be instantiated to a *resource stack type*. A resource stack type is either unit (representing no resources) or τ * σ where τ is a *resource type* and σ is a resource stack type. Possible resource types are boxed types, pairs of resource types, the type bool, and τ list where τ is another resource type. For simplicity of exposition, we assume that int is the only boxed type.

The first group of resource-related combinators is for manipulating the resource stack by constructing and deconstructing pairs (pair, unpair), making it possible to shuffle an arbitrary entry to the top, dropping resources that are no longer needed (drop), duplicating resources (dup), and so on. Notice that these operations do not have to be recorded as part of the branch; the fact that they participated in some computation is recoverable from the path through the flow-graph⁴, and the path is being recorded.

⁴ This observation holds for all non-branching computations that do not transfer data between the resources and the accumulator.

The second group (consisting of `Bang`, `If`, and `Case`) provides combinators for reading resources while transferring their values to the accumulator, for branching on boolean resources, and for doing case analysis on list resources. These operations extend the branch.

Finally, the `memo` combinator caps off a memo-computation and turns it into a memoized function. The function’s argument appears as the only element on the initial resource stack; the initial accumulator is empty.

4.2 Examples

Assuming a combinator `lt` for comparing an integer resource to a constant, thereby turning it into a boolean resource,⁵ we can write our introductory example as follows:

```

fun g (x:int):int = ...      fun h (x:int):int = ...
val f: int * (int * int) → int =
  memo (unpair (swap (unpair (swap2 (swap (
    lt (0, If (swap (drop (Bang (Do (fn (y, ()) ⇒ g(y), return))))),
    drop (Bang (Do (fn (z, ()) ⇒ h(z), return))))))))))

```

Notice the “reverse polish” feel of this code. The type constraint on `f` is actually unnecessary as the ML compiler can infer it.

A slightly more elaborate example is that of a function which takes a list of integers and produces a sorted list consisting of every other element of its argument. Memoization will ignore those elements that do not contribute to the answer. We must construct a loop using the internal fixpoint combinator `fix` (see Section 4.4) to be able to iterate over a list. The bindings of memo-computations to SML variables `sortit` and `loop` act as labels, the mention of those labels in place of a memo-computation act as unconditional control transfers similar to a **goto**:

```

val sortodd: int list → int list =
  let val sortit = Do (ListMergeSort.sort op >, return)
  in memo (Do (fn () ⇒ [],
    fix (fn loop ⇒ Case (sortit,
      unpair (Bang (Do (op ::, Case (sortit, unpair (drop loop))))))))))
  end

```

Programming with these combinators definitely feels a bit unusual—except, perhaps, to Forth programmers [15]. The biggest problem, however, is something we only mentioned in passing so far: *All* supported operations on resources such as our `lt` above would have to be built into the library from the beginning. To avoid leakage of resources, the set of such operations could not be extended (at least not safely) using ML code. In the end it seems that aside from imposing a strange programming style, a safe library incurs serious problems with scalabil-

⁵ The type of `lt` is `int * (bool * α , ν , ρ) mc → (int * α , ν , ρ) mc.`

```

type branch = hashvalue list
type ( $\alpha$ ,  $\nu$ ,  $\rho$ ) mc =  $\alpha$  * (unit  $\rightarrow$   $\nu$ ) * branch *  $\rho$  memotable  $\rightarrow$   $\rho$ 

fun return (_, a, br, mt) =
  case lookup (mt, br) of
    SOME ans  $\Rightarrow$  ans
  | NONE  $\Rightarrow$  let val ans = a () in insert (mt, br, ans); ans end

fun memo c = let val mt = newtable ()
             in fn x  $\Rightarrow$  c ((x, ()), fn ()  $\Rightarrow$  (), [], mt)
             end

fun pair c ((x, (y, z)), a, br, mt) = c (((x, y), z), a, br, mt)
fun unpair c (((x, y), z), a, br, mt) = c ((x, (y, z)), a, br, mt)
fun dup c ((x, y), a, br, mt) = c ((x, (x, y)), a, br, mt)
fun swap c ((x, (y, z)), a, br, mt) = c ((y, (x, z)), a, br, mt)
fun swap2 c ((x, (y, (z, w))), a, br, mt) = c ((x, (z, (y, w))), a, br, mt)
fun drop c ((x, y), a, br, mt) = c (y, a, br, mt)

fun Do (f, c) (x, a, br, mt) = c (x, f o a, br, mt)

fun lt (k, c) ((x, y), a, br, mt) = c ((x < k, y), a, br, mt)

fun Bang c ((x, y), a, br, mt) = c (y, fn ()  $\Rightarrow$  (x, a ()), hash x::br, mt)

fun If (ct, _) ((true, x), a, br, mt) = ct (x, a, hash 0::br, mt)
  | If (_, cf) ((false, x), a, br, mt) = cf (x, a, hash 1::br, mt)
fun Case (cn, _) (([], y), a, br, mt) = cn (y, a, hash 0::br, mt)
  | Case (_, cc) ((x::l, y), a, br, mt) = cc ((x, l), y), a, hash 1::br, mt)

```

Fig. 11. Implementation of branch library

ity. Therefore, providing language- and compiler support ultimately looks like the better proposition.

4.3 Under the hood

Given a working implementation of memo tables and hashing, a simple implementation of the combinator library, shown in Figure 11, is actually very short. We represent memo-computations as functions taking four arguments: the resources, the accumulator suspended as a thunk, the branch (represented as a list of hash values), and the memo-table. Upon return, the accumulator is forced if and only if there is no memo-match. Function memo allocates a new memo-table and returns a function which passes this table to the computation—along with an empty branch and an empty accumulator.

Notice that the Do combinator causes its argument f to be composed into the current non-resource thunk. As a result, this function will not be invoked until return, and in case of a successful memo lookup it will not be called at all.

The Bang combinator extends the branch with the hash of the resource being read. If and Case indicate in the branch which of their respective two continuations was selected.

```

val fix: (( $\alpha$ ,  $\nu$ ,  $\rho$ ) mc  $\rightarrow$  ( $\alpha$ ,  $\nu$ ,  $\rho$ ) mc)  $\rightarrow$  ( $\alpha$ ,  $\nu$ ,  $\rho$ ) mc
fun fix mkc = let fun c (x, a, br, mt) = mkc c (x, a, br, mt) in c end

val memofix: (( $\alpha \rightarrow \rho$ )  $\rightarrow$  ( $\alpha$  * unit, unit,  $\rho$ ) mc)  $\rightarrow$   $\alpha \rightarrow \rho$ 
fun memofix mkc = let val mt = newtable ()
                    fun f x = mkc f ((x, ()), fn ()  $\Rightarrow$  (), [], mt)
                    in f end

```

Fig. 12. Recursion for memo-computations

4.4 Recursion

We distinguish between two kinds of recursion involving memo computations: *internal* and *external*. Internal recursion is always tail-recursion and corresponds to loops within the control flow graph representing the body of a memoized function. Recursive calls do not involve separate memo-table lookups but simply pass along the current branch. As shown in Figure 12, internal recursion is implemented using a fixpoint combinator `fix` for memo-computations which, internally, is nothing more than an ordinary fixpoint combinator.

External recursion occurs when computations specified using `Do` call the memoized function itself as a subroutine. For this we need `memofix`, a version of `memo` which has an integrated fixpoint combinator. The `memofix` combinator is not expressible directly in terms of `memo` plus some ordinary fixpoint combinator as this would require abstracting over the memoized function itself. That, in turn, causes the call of `memo` to be suspended, and since every call of `memo` generates a fresh memo table, the desired reuse of earlier results would not take place. (A similar issue arises in Erkök and Launchbury’s work on recursive monadic bindings [11].)

5 Conclusions

In this paper we present a library for writing programs that adjust to changes to their data automatically—we call such programs self-adjusting. The library makes writing self-adjusting programs nearly as easy as writing ordinary programs by enabling the programmer to transform an ordinary program into a self-adjusting program. Our experiments show that the library accepts an efficient implementation, and that self-adjusting programs can adjust to changes significantly faster than recomputing from scratch.

We also show some techniques that help enforce the numerous invariants required for self-adjusting programs to work correctly. We conclude, however, that restricting oneself to the tools provided by existing programming languages (even relatively sophisticated ones such as ML) does not make this easy. Fully general and scalable solutions probably require direct linguistic support.

References

- [1] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.
- [2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.
- [3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.
- [4] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vitter, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.
- [5] Gerth Stolting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 617–626, 2002.
- [6] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 26–35. ACM Press, 2002.
- [7] Frederica Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In *Proceedings of Thirteenth International Symposium on Algorithms and Computation (ISAAC)*, volume 3038 of *Lecture Notes in Computer Science*, pages 662 – 669. Springer, 2004.
- [8] Frederica Darema. Dynamic data driven applications systems: New capabilities for application simulations and measurements. In *Proceedings of Thirteenth International Symposium on Algorithms and Computation (ISAAC)*, volume 3515 of *Lecture Notes in Computer Science*, pages 610 – 615. Springer, 2005.
- [9] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.
- [10] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.
- [11] Levent Erkök and John Launchbury. Recursive monadic bindings. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 174–185, New York, NY, USA, 2000. ACM Press.
- [12] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.
- [13] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on PLDI*, pages 311–320, May 2000.

- [14] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.
- [15] Charles H. Moore and G. C. Leach. FORTH – A language for interactive computing. 1970. Amsterdam, NY: Mohasco Industries Inc. (internal pub.).
- [16] Ketan 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.
- [17] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.
- [18] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.
- [19] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
- [20] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.
- [21] Stephen Weeks, Matthew Fluet, Henry Cejtin, and Suresh Jagannathan. Mlton. <http://mlton.org/>, 2005.
- [22] Mark N. Wegman and Larry Carter. New classes and applications of hash functions. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 175–182, 1979.

A The Implementation

The code below assumes the existence of the following structures or functors: MemoTable, Box, PriorityQueue, and TimeStamps.

```
signature COMBINATORS =
sig
  type 'a modref = 'a Modref.t
  type 'a cc

  val modref : 'a cc -> 'a modref
  val write : ('a * 'a -> bool) -> 'a -> 'a cc
  val read : 'b Modref.t * ('b -> 'a cc) -> 'a cc

  val mkLift : ('b * 'b -> bool) -> (int list * 'b) -> ('b Modref.t -> 'd) -> 'd
  val mkLiftCC : (('b * 'b -> bool) * ('d * 'd -> bool)) ->
    int list * 'b -> ('b Modref.t -> 'd cc) -> 'd cc
  val change : ('a * 'a -> bool) -> 'a modref -> 'a -> unit
  val deref : 'a modref -> 'a
  val propagate : unit -> unit
end

structure Comb :> COMBINATORS =
struct

  type 'a modref = 'a Modref.t
  type 'a cc = 'a modref -> Modref.changeable

  type ('b, 'g) liftpad = ('b -> 'g) MemoTable.t * 'g MemoTable.t
  type ('b, 'g) liftpadCC = ('b -> 'g cc) MemoTable.t * 'g Modref.t MemoTable.t

  fun liftPad () = (MemoTable.new (), MemoTable.new ())

  fun write eq x d = Modref.write eq d x

  fun read (r, recv) d = Modref.read r (fn x => recv x d)
  val modref = Modref.modref

  val change = Modref.change
  val deref = Modref.deref
  val propagate = Modref.propagate

  fun memoize (pad:'a MemoTable.t) key (f: unit -> 'a) =
    let
      fun run_memoized (f,r) =
        let
          val t1 = !(Modref.now)
          val v = f()
          val t2 = !(Modref.now)
          val nt1o = TimeStamps.getNext t1
          val _ =
            case nt1o of
              NONE => r:=SOME(v,NONE)
            | SOME(nt1) =>
              if (TimeStamps.compare (nt1,t2)=LESS) then
                (r := SOME(v,SOME(nt1,t2)))
              else
                (r := SOME(v,NONE))
          in v end
        end
    in v end

  fun reuse_result (t1,t2) =
    let
      val _ = TimeStamps.spliceOut (!(Modref.now),t1)
      val _ = Modref.propagateUntil t2
    end
end
```

```

    in ()end

fun memoize' (r:'a MemoTable.entry) =
  case !r of
  NONE => run_memoized (f,r)
  | SOME(v,to) =>
    case to of
    NONE => v
    | SOME(t1,t2) => (reuse_result (t1,t2); v)
  in
  memoize' (MemoTable.find(pad,key,!Modref.now))
end

fun lift (p1,p2) eqb (key,b) f =
  let fun f' () = let val r = Modref.empty ()
                in fn b => let val _ = change eqb r b
                in memoize p2 key (fn _ => f (r)) end
    end
  in memoize p1 key f' b end

fun mkLift eqb = lift (liftPad ()) eqb

fun mkLiftCC (eqb,eqd) =
  let
  fun lifted arg f =
    let fun f' (b) = let val r = modref (f b) in read (r, write eqd) end
    in lift (liftPad ()) eqb arg f' end
  in
  lifted
end

end

signature MODIFIABLE =
sig

  type 'a modref
  type 'a t = 'a modref
  type changeable
  type time = TimeStamps.t

  val init : unit -> unit

  (*****
  ** Standard operations with modifiabiles.
  *****)
  val empty: unit -> 'a modref
  val modref: ('a modref -> changeable) -> 'a modref
  val read : 'a modref -> ('a -> changeable) -> changeable
  val write : ('a * 'a -> bool) -> 'a modref -> 'a -> changeable

  (*****
  ** Meta-machine operations
  *****)
  val change: ('a * 'a -> bool) -> 'a modref -> 'a -> unit
  val deref : 'a modref -> 'a
  val propagate : unit -> unit
  val propagateUntil : time -> unit

end

structure Modref : MODIFIABLE=
struct
  type time = TimeStamps.t
  type changeable = unit

  exception UnsetMod

```

```

(*****
** Time Stamps
*****)
val now = ref (TimeStamps.add (TimeStamps.init ()))
val frameStart = ref (!now)
val finger = ref (!now)

fun insertTime () =
  let val t = TimeStamps.add (!now)
  in now := t; t
  end

(*****
** Priority Queue
*****)
structure Closure =
struct
  type t = ((unit -> unit) * time * time)
  fun compare (a as (ca,sa,ea), b as (cb,sb,eb)) = TimeStamps.compare(sa,sb)
  fun isValid (c,s,e) = not (TimeStamps.isSplicedOut s)
end
structure PQueue = PriorityQueue (structure Element=Closure)

type pq = PQueue.t
val PQ = ref PQueue.empty
fun initQ() = PQ := PQueue.empty
fun insertQ e = PQ := PQueue.insert (e,!PQ)
fun findMinQ () =
  let val (m,q) = PQueue.findMin (!PQ)
  val _ = PQ := q
  in m end
fun deleteMinQ () =
  let val (m,q) = PQueue.deleteMin (!PQ)
  val _ = PQ := q
  in m end

(*****
** Modifiables
*****)
type 'a reader = ('a -> unit) * time * time
datatype 'a readers = NIL | FUN of ('a reader * 'a readers)
datatype 'a modval = EMPTY | WRITE of 'a * 'a readers
type 'a modref = 'a modval ref
type 'a t = 'a modref

fun empty () = ref EMPTY
fun modref f =
  let val r = (ref EMPTY)
  in (f (r); r) end
fun read modr f =
  case !modr of
  EMPTY => raise UnsetMod
  | WRITE (v,_) =>
    let val t1 = insertTime()
    val _ = f(v)
    val t2 = insertTime ()
    val WRITE (v,rs) = !modr
    val rs' = FUN((f,t1,t2),rs)
    in modr := WRITE(v,rs')
    end
fun readAtTime(modr,r as (f,_,_)) =
  case !modr of
  EMPTY => raise UnsetMod
  | WRITE(v,rs) => (modr := WRITE(v,FUN (r,rs))); f v
fun addReadersToQ (rs: 'a readers, modr : 'a modref) =

```

```

let fun addReader (r as (f,t1,t2)) =
  if TimeStamps.isSplicedOut(t1) then ()
  else insertQ(fn () => readAtTime(modr,r),t1,t2)
  fun addReaderList rlist =
    case rlist of
      NIL => ()
    | FUN(r,rest) => (addReader (r); addReaderList rest)
  in addReaderList rs
end
fun write comp modr v =
  case !modr of
    EMPTY => modr := WRITE (v,NIL)
  | WRITE(v',rs) =>
    if comp (v,v') then ()
    else let val _ = modr := WRITE(v,NIL)
         in addReadersToQ (rs,modr)
        end
end

fun deref modr =
  case !modr of
    EMPTY => raise UnsetMod
  | WRITE (v,_) => v

(*****
** Change propagation
*****)
fun propagateUntil (endTime) =
  let fun loop () =
        case (findMinQ ()) of
          NONE => ()
        | SOME(f,start,stop) =>
          if (TimeStamps.isSplicedOut start) then loop ()
          else if (TimeStamps.compare(endTime,stop) = LESS) then ()
          else let val _ = deleteMinQ ()
               val finger' = (!finger)
               val _ = now := start
               val _ = finger := stop
               val _ = f()
               val _ = finger := finger'
               val _ = TimeStamps.spliceOut (!now,stop) handle e => raise e
            in loop ()
            end
        end
  in (loop (); now := endTime)
  end

fun propagate () =
  let fun loop () =
        case (findMinQ ()) of
          NONE => ()
        | SOME(f,start,stop) =>
          let val _ = deleteMinQ ()
              val finger' = (!finger)
              val _ = now := start
              val _ = finger := stop
              val _ = f()
              val _ = finger := finger'
              val _ = TimeStamps.spliceOut (!now,stop) handle e => raise e
            in loop ()
            end
        end
  in loop ()
  end

fun init () = (now := TimeStamps.init(); initQ())
fun change comp l v = write comp l v
end

```