Self-Adjusting Computation with Delta ML

Umut A. Acar[1] and Ruy Ley-Wild[2]

[1] Toyota Technological Institute
Chicago, IL, USA
umut@tti-c.org
[2] Carnegie Mellon University
Pittsburgh, PA, USA
rleywild@cs.cmu.edu

Abstract. In self-adjusting computation, programs respond automatically and efficiently to modifications to their data by tracking the dynamic data dependences of the computation and incrementally updating the output as needed. In this tutorial, we describe the self-adjusting-computation model and present the language ΔML (Delta ML) for writing self-adjusting programs.

1 Introduction

Since the early years of computer science, researchers realized that many uses of computer applications are *incremental* by nature. We start an application with some initial input to obtain some initial output. We then observe the output, make some small modifications to the input and re-compute the output. We often repeat this process of modifying the input incrementally and re-computing the output. In many applications, incremental modifications to the input cause only incremental modifications to the output, raising the question of whether it would be possible to update the output faster than recomputing from scratch.

Examples of this phenomena abound. For example, applications that interact with or model the physical world (e.g., robots, traffic control systems, scheduling systems) observe the world evolve slowly over time and must respond to those changes efficiently. Similarly in applications that interact with the user, application-data changes incrementally over time as a result of user commands. For example, in software development, the compiler is invoked repeatedly after the user makes small changes to the program code. Other example application areas include databases, scientific computing (e.g., physical simulations), graphics, etc.

In many of the aforementioned examples, modifications to the computation data or input are external (e.g., the user modifies some data). In others, incremental modifications are inherent. For example, in motion simulation, objects move continuously over time causing the property being computed to change continuously as well. In particular, if we wish to simulate the flow of a fluid by modeling its constituent particles, then we need to compute certain properties of moving objects, e.g., we may want to triangulate the particles to compute the

forces exerted between particles, and update those properties as the points move. Since the combinatorial structure of the computed properties change slowly over time, we can often view continuous motion as an incremental modification; this makes it possible to compute the output more efficiently than re-computing it from-scratch at fixed intervals.

Although incremental applications abound, no effective general-purpose technique or language was known for developing incremental applications until recently (see Section 10 for the discussion of the earlier work on the subject). Many problems required designing specific techniques or data structures for remembering and re-using results to ensure that computed properties may be updated efficiently under incremental modifications to data. Recent advances on self-adjusting computation (Section 10.2) offer an alternative by proposing general-purpose techniques for automatically adapting computations to data modifications by selectively re-executing the parts of the computation that depend on the modifications and re-using unaffected parts. Applications of the technique to problems from a reasonably diverse set of areas show that the approach can be effective both in theory and practice.

We present a tutorial on a language for self-adjusting computation, called ΔML (Delta ML), that extends the Standard ML (SML) language with primitives for self-adjusting computation.

In self-adjusting computation, programs consist of two components: a self-adjusting *core* and a top- or meta-level *mutator*. The self-adjusting core is a purely functional program that performs a single run of the intended application. The mutator drives the self-adjusting core by supplying the initial input and by subsequently modifying data based on the application. The mutator can modify the computation data in a variety of forms depending on the application. For example, in a physical simulation, the mutator can insert a new object into the set of objects being considered. In motion simulation, the mutator changes the outcomes of comparisons performed between objects as the relationship between objects change because of motion. After modifying computation data, the mutator can update the output and the computation by requesting *change propagation* to be performed. Change propagation is at the core of self-adjusting computation: it an automatic mechanism for propagating the data modifications through the computation to update the output.

To support efficient change propagation, we represent a computation with a *trace* that records the data and control dependences in the computation. Change propagation uses the trace to identify and re-execute the parts of the computation that depend on the modified data while re-using the parts unaffected by the changes. The structure and the representation of the trace is critical to the effectiveness of the change propagation. Techniques have been developed for implementing both tracing and change propagation efficiently (Section 10.2).

The ΔML language provides linguistic facilities for writing self-adjusting programs consisting of a core and a mutator. To this end, the language distinguishes between two kinds of function spaces: conventional and self-adjusting. The mutator consists solely of conventional functions. The self-adjusting core consists

of self-adjusting functions and all pure (self-adjusting or conventional) functions that they call directly or indirectly (transitively).

ΔML enables the programmer to mark the computation data that is expected to change across runs (or over time) by placing them into *modifiable references* or *modifiables* for short. For implementing a self-adjusting core, ΔML provides facilities for creating and reading modifiables within a self-adjusting function. In this tutorial, we do not include the update operation on modifiables in the core—modifiables are write-once within the self-adjusting core. [3] ΔML also provides facilities for defining self-adjusting functions to be memoized if so desired. For implementing a mutator, ΔML provides meta-level facilities to create, read, and update modifiables, and to perform change propagation. The mutator can use the update operation to modify destructively the contents of modifiables—this is how mutators modify the inputs of the self-adjusting core. After such modifications are performed, the mutator can use change propagation to update the result of the core.

Writing a self-adjusting program is very similar to writing a conventional, purely functional program. Using the techniques described in this tutorial, it is not hard to take an existing purely functional SML program and make it self-adjusting by annotating the code with ΔML primitives. Annotated code is guaranteed to respond to modifications to its data correctly: the result of an updated run is equivalent to a (from-scratch) run. Guaranteeing efficient change propagation, however, may require some additional effort: we sometimes need to modify the algorithm or use a different algorithm to achieve the optimal update times.

When an algorithm does not yield to efficient change propagation, it is sometimes possible to change it slightly to regain efficiency, often by eliminating unnecessary dependences between computation data and control. For example, the effectiveness of a self-adjusting mergesort algorithm can be improved by employing a divide-and-conquer strategy that divides the input into two sublists randomly instead of deterministically dividing in the middle. Using randomization eliminates the dependence between the length of the list and the computation, making the computation less sensitive to modiciations to the input (e.g. when a new element is inserted the input length changes, causing the divide-and-conquer algorithm to create different splits than before the insertion, ultimately preventing re-use). Sometimes, such small changes to the algorithm do not suffice to improve its efficiency and we need to consider an entirely different algorithm. For example, the quicksort algorithm is inherently more sensitive to input modifications than the mergesort algorithm, because it is sensitive to values of the pivots, whereas the mergesort algorithm is not. Similarly, an algorithm that sums a list of numbers by performing a traversal of the list and maintaining an accumulator will not yield to efficient change propagation, because inserting an element can change the value of the accumulator at every recursive call. No

[3] The actual ΔML language places no such restrictions on how many time modifiables can be written in the core.

small modification will improve this algorithm as we would like. Considering a different, random sampling algorithm addresses the problem (Section 8).

The structure of the rest of the tutorial is as follows. In Section 2 we describe how incremental modifications arise and why they can lead to improved efficiency and why having general-purpose techniques and languages can help take advantage of this potential. In Section 3 we describe the self-adjusting computation model and the core and the meta primitives for writing self-adjusting programs. In Section 4 we describe an example self-adjusting application, called CIRCLES, and how the user can interact with such a program. In the rest of the tutorial, we use this example to illustrate how the ΔML language may be used to implement self-adjusting programs.

2 Motivation

We consider the two kinds of modifications, discrete and continuous, that arise in incremental applications via simple examples and describe how we may take advantage of them to improve efficiency. We then describe how and why language-based techniques are critical for scalability.

2.1 Discrete and Continuous Modifications

Close inspection of incremental applications reveal that two kinds of modification arise naturally: discrete/dynamic and continuous/kinetic.

Discrete/Dynamic: A discrete or dynamic modification is a *combinatorial* modification to the input of the program that modifies the set of objects in the input.

Continuous/Kinetic: A continuous or kinetic modification affects the relationship between the input objects but does not affect the set of objects itself. By a relationship we broadly mean any function mapping objects into a discrete set. For example, comparisons between objects are relationships because the co-domain contains `true` and `false`.

As an example, consider sorting a list of numbers and how discrete and continuous modifications may arise in this application.

Suppose that we sort a list of numbers, e.g., `[30,10,20,0]`, and then we insert a new number to the list, e.g. `[7,30,10,20,0]`. Since the set of objects itself is modified this is a discrete modification. Note that inserting/deleting an element from the input only causes an in incremental modification to the output, that of inserting/deleting the new element into/from the output (at the right position). Thus we can expect to be able to update the output significantly faster than recomputing from scratch. In fact, we can update the output for a sorting application in optimal logarithmic time, instead of the $O(n \log n)$ time a re-computation would require.

As an example of continuous modifications, consider a set of numbers that change as a function of time $a(t) = 40.0 - 0.3t$, $b(t) = 20.0 - 0.3t$, $c(t) = 10$, and

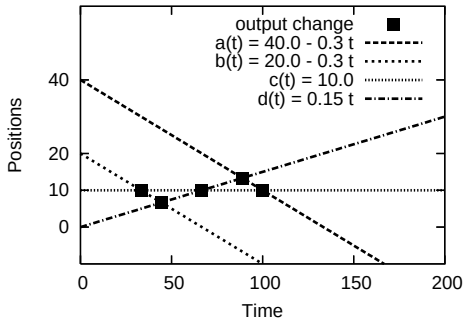

Fig. 1. Numbers varying with time.

$d = 0.15t$. Suppose that we want to keep the numbers sorted as they change over time. More specifically, we want to start at time zero ($t = 0$), and then update the output whenever the ordering changes. Since time changes continuously, the values of the functions also change continuously. But the sorting changes discretely—only when the outcomes of comparisons between the moving numbers change.

For example, at time 0, sorted output is $[d(t), c(t), b(t), a(t)]$, which remains the same until the time becomes $33.\dot{3}$, when $b(t)$ falls below $c(t)$ changing the output to $[d(t), b(t), c(t), a(t)]$. The output then remains the same until $44.\dot{4}$, when $b(t)$ falls below $d(t)$, and the output becomes $[b(t), d(t), c(t), a(t)]$. The output then changes at $66.\dot{6}$ to $[b(t), c(t), d(t), a(t)]$, and at $88.\dot{8}$ to $[b(t), c(t), a(t), d(t)]$. The final change takes place at time 100.0, when the output becomes $[b(t), a(t), c(t), d(t)]$. Although the output changes continuously, we only need to update the computation at these times, i.e., when the output changes combinatorially.

Note that when the outcome of the comparison changes, the change in the output is small—it is simply a swap of two adjacent numbers. This property enables treating motion as a form of incremental modification. In fact, in this example, we can model continuous modifications as discrete modifications that affect the outcomes of comparisons. In general, if the computed property only depends on relationships between data whose values range over a discrete domain, then we can perform motion simulation by changing the values of these relationships as they take different values discretely.

2.2 Taking Advantage of Incrementality

When an incremental modification to computation data causes an incremental modification to the output, we can expect to update the output faster than by re-computing from scratch. In this tutorial, we propose language-based general purpose techniques for taking advantage of incremental modifications to update outputs faster. An alternative approach would be to design and implement ad

hoc data structures, called *dynamic data structures*, on a per-problem basis. In this section, we briefly overview the design of several dynamic data structures for some relatively simple problems and point out some difficulties with the approach.

As a simple example, consider mapping a list to another list. Here is the interface for a `DynamicMap` data structure:

```
signature DynamicMap =
sig
   type (α,β) t
   val map: α list -> (α -> β) -> β list * (α,β) t
   val insert: α * int * (α,β) t -> β list * (α,β) t
end
```

The `map` function performs an "initial map" of the input and generates the output list as well as a data structure (of abstract type `(α,β) t`) that can be used to speedup the subsequent insertions. After `map` is executed, we can modify the input by inserting new elements using the `insert` operation. This operation takes the new element with the position at which is should be inserted and the data structure, and returns the updated output list and data structure.

Having designed the interface, let's consider two possible choices for the auxiliary data structure that is used to speed up the insertions.

1. The auxiliary data structure is the input, the `map` function simply returns the input. The `insert` function inserts the element into the list at the specified position. To update the output we have no choice but to re-execute, which offers no performance gain.
2. The auxiliary data structure represents the input and the output along with pointers linking the input elements to the corresponding output elements. These pointers help in finding the location of the element corresponding to an input element in the output. The `map` function constructs and returns this data structure. The `insert` function inserts the element in the input at the specified position, maps the element to an output element, and using the pointers to the output, finds the position for the output element, and inserts it. Using this approach, we can insert a new element and update the output with expected constant time overhead over the time it takes to map the input element to an output. Note that we can update the input in (expected) constant time by maintaining a mapping between locations and the pointers to update.

As our second example, consider a dynamic data structure for sorting with the following interface.

```
signature DynamicSort =
sig
   type α t
   val sort: α list -> (α * α -> bool) -> α list * α t
   val insert: (α * int) -> α t -> α list * α t
```

```
end
```

The sort operation performs an "initial sort" of the input and generates the output list as well as the auxiliary data structure (of abstract type α `t`). After `sort` is executed, we can change the input by inserting new elements using the `insert` operation. This operation takes the new element with the position at which it should be inserted and the data structure, and returns the updated output list and data structure.

As before, there are choices for the auxiliary data structure.

1. The `sort` function returns the input as the auxiliary data structure. The `insert` operation simply modifies the input and sorts from scratch to update the output. This is equivalent to a from-scratch execution.
2. The `sort` function returns both input and the output as the auxiliary data structure. The `insert` operation inserts the new element into the input and into the output. In this case, the `insert` operation can be performed in $O(n)$ time.
3. The `sort` function builds a balanced binary-tree representation of the input list and returns it as the auxiliary data structure. The nodes of the balanced binary search tree point to their place in the output list. The `insert` operation performs an insertion into the binary search tree and updates the output by splicing the inserted element into the output at the right place. To find the right place in the output, we find the previous and the next elements in the output by performing a search. These operations can be performed in $O(\log n)$ time by careful use of references and data structures.

The first two approaches are not interesting. They improve performance by no more than a logarithmic factor, which would not worth the additional complexity. The third approach, however, improves performance by a linear factor, which is very significant.[4]

As this example suggests, designing an efficient incremental data structure can be difficult even for a relatively simple problem like sorting. In particular, to support efficient incremental updates, we need to use balanced binary search trees, whose design and analysis is significantly more complex than that of a sorting algorithm. This suggests that there is a design-complexity gap between conventional static problems (where computation data is not subject to modifications) and their dynamic version that can respond efficiently to incremental modifications. Indeed, the design and analysis of such dynamic data structures has been an active field in the algorithms community (Section 10.3). We give more examples of this complexity gap in Section 10.3.

In addition to the design-complexity gap, there are difficulties in using dynamic data structures for building large software systems, because they are not composable. To see why this is important note that conventional algorithms are directly composable. For example if we have some function $f : \alpha \to \beta$ and

[4] Note that we can support modifications to the input in (expected) constant time by maintaining a mapping between locations and the pointers to update.

$g : \beta \to \gamma$, then we can compose these two algorithms, $g \circ f$, is well defined and has type $\alpha \to \gamma$. Concretely, if we want to sort a list of objects and then apply a map function to the sorted list, then we can achieve this by composing the functions for sorting and mapping. Not so with dynamic data structures. As an example suppose that we wish to compose `DynamicSort` and `DynamicMap` so that we can maintain a sorted and mapped list under incremental modifications to the input. For example, we may want sort a set of (one-dimensional) points on a line from left to right and then project them onto some parabola by using a list-map, while we maintain this relationship as the input list is modified with insertions and deletions. Call the dynamic data structure for this purpose `DynamicSortMap`.

If dynamic data structures were composable, we could easily implement `DynamicSortMap` by composing `DynamicSort` and `DynamicMap`. The difficulty arises when composing the `insert` functions. When we insert a new element, we want it to be mapped on to the parabola and inserted at the right place in the output. We could update the initial sorting of the points on the line by invoking the `insert` of `DynamicSort` but to apply the insertion to `DynamicMap` we also need to know where in the output the new point appears so that we can modify the input to `DynamicMap`. In other words, we don't need just the updated output; we also need to know how the output has changed, which `DynamicSort` does not provide.

In general, for dynamic data structures to be composable, data structures must return a description of how their output has changed as a result of a modification to their input. This description must be in some universal "language" so that any two dynamic data structures can be combined (when possible). We must also develop some way to apply such output-change descriptions to modify the input. This can require converting arbitrary modifications to the output into input-modifications that are acceptable by the composed data structure. For example, a change that replaces the value of an element in the output must be converted into a deletion and an insertion, if the composed data structures do not support replacement directly. The authors are not aware of any concrete proposals for facilitating such composition of dynamic data structures. Another approach could be to compute the difference between the outputs of a consecutive applications of a function and use this difference as an incremental modification. This approach, however, may be difficult to make work optimally. For example, it can require solving the graph isomorphism problem (when comparing two graphs), which is NP hard.

The approach proposed in this tutorial is different: it enables writing programs that can respond to arbitrary modifications to their data. The programs are written in a style similar to conventional programs; in particular, functions are composable.

3 The Programming Model: An Overview

In self-adjusting computation programs are stratified into two components: a *self-adjusting core*, or a *core* for short, and a meta-level *mutator*.

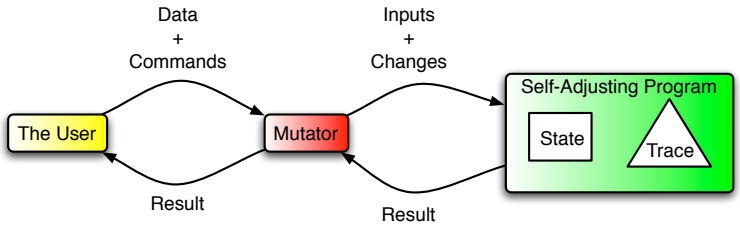

Fig. 2. The self-adjusting computation model.

A (self-adjusting) core is written like a conventional program: it takes an input and produces an output, while also performing some effects (i.e., writing to the screen). Like a conventional program, we can execute the core with an input, performing a a *from-scratch* or an *initial* run. Typically, we execute a core from-scratch only once, hence the name initial run.

After an initial run, the input data or intermediate data, which is generated during the initial run, can be modified and the core can be asked to update its output by performing change propagation. This process of modifying data and propagating the modifications can be repeated as many times as desired. *Change propagation* updates the computation by identifying the parts that are affected by the modifications and re-executing them, while re-using the parts that are unaffected by the changes. Although change propagation re-executes only some parts of the core, it is semantically equivalent to a (from-scratch) run: it is guaranteed to yield the same result as running the core from scratch with the modified input. The asymptotic complexity of change propagation is never slower than a from-scratch run and can be dramatically faster. In a typical use, an initial run is followed by numerous iterations of changing the input data and performing change propagation. We therefore wish change propagation to be fast even if this comes at the expense of slowing down the initial run.

The interaction between the cores initial output and its subsequent inputs may be complex. We therefore embed a core program in a *meta-level mutator* program—the mutator drives the feedback loop between the self-adjusting core and its data. One common kind of core is a program that interacts with the user to obtain the initial input for the core, runs the core with that input to obtain an output, inspects the output, and continues interacting with the user by modifying the data as directed and performing change propagation as necessary. Another common kind of mutator used in motion simulation combines user-interactions with event scheduling. While interacting with the user, such a

mutator also maintains an event queue consisting of objects that indicate the comparisons whose outcomes need to be modified and at which time. The mutator performs motion simulation by changing the outcomes of comparisons and performing change propagation. Change propagation updates the event queue and the output. In general, mutators can change computation data in arbitrarily complex ways.

4 An Example: CIRCLES

As a running example, consider an application, called CIRCLES, that displays a set of circles and identifies the pair of circles that are furthest away from each other, i.e., the *diameter*. The user enters the initial set of circles and subsequently modifies them by inserting new circles, deleting existing circles, changing the properties of existing circles, etc.

Figure 3 shows an example interaction between a user and CIRCLES. Initially the user enters some circles, labeled "a" through "g", specifies their color, the position of their center (the top left corner is the origin) and their radii. CIRCLES computes the diameter of this set, namely "c" and "e", and renders these circles as shown on the right. The user is then allowed to modify the set, which s/he does by changing the color of "a" from yellow to blue. CIRCLES responds to this change by re-rendering circle "a". The user then inserts a new circle "h", to which CIRCLES responds by displaying the new circle and updating the new diameter, namely "c" and "h." Finally, the user deletes the new circle "h", to which CIRCLES responds by blanking out "h", and updating and displaying the diameter.

Note that CIRCLES only re-draws the modified circles. Although it is not apparent in this discussion it also updates the diameter efficiently, much faster than re-computing the diameter from scratch.

How can we write such a program? One way would be to design and develop a program that carefully updates the display and the diameter by using efficient algorithms and data structures. We encourage the reader to think about how this could be achieved. Some thought should convince the reader that it is a complex task. Rendering only the affected circles is reasonably simple: we can do this by keeping track of the circle being modified, or inserted/deleted, and rendering only that circle. Updating the diameter efficiently, however, is more difficult. For example, how can we compute the new diameter, when the user deletes one of the circles that is currently part of the diameter? Can this be done more efficiently than recomputing from scratch? This turns out to be a non-trivial question (see Section 10 for some related work). Next, we describe the ΔML language and how we can implement CIRCLES in ΔML.

5 The Delta ML Language

The ΔML language extends the Standard ML language with primitives for self-adjusting computation. The language distinguishes between two function spaces:

```
> Welcome to circles.
> Enter the circles, type "done" when finished:
  a = yellow (0.88, 0.70) 0.37
  b = red (0.36, 2.22) 0.22
  c = green (0.94,3.24) 0.34
  d = black (2.50,2.61) 0.25
  e = orange (3.01, 1.01) 0.25
  f = yellow (2.26, 1.30) 0.22
  g = brown (1.45, 1.75) 0.25
  done
> Rendering circle a.
> Rendering circle b.
> Rendering circle c.
> Rendering circle d.
> Rendering circle e.
> Rendering circle f.
> Rendering circle g.
> The diameter is (c,h)
> Enter the circle to modify or type "quit":
  a
> Enter circle or type "delete".
  a = blue (0.88, 0.70) 0.75
> Propagating...
>   Rendering circle a.
> Done.
> Enter the circle to modify or type "quit":
  h
> Enter circle or type "delete".
  h = purple (2.26, 0.11) 0.75
> Registered new circle.
> Propagating...
>   Rendering circle h.
>   The diameter is (c,h).
> Done.
> Enter the circle to modify or type "quit":
  h
> Enter circle or type "delete".
  "delete"
> Deleted circle h.
> Propagating...
>   Rendering circle h (blanking out).
>   The diameter is (c,e).
> Done.
```

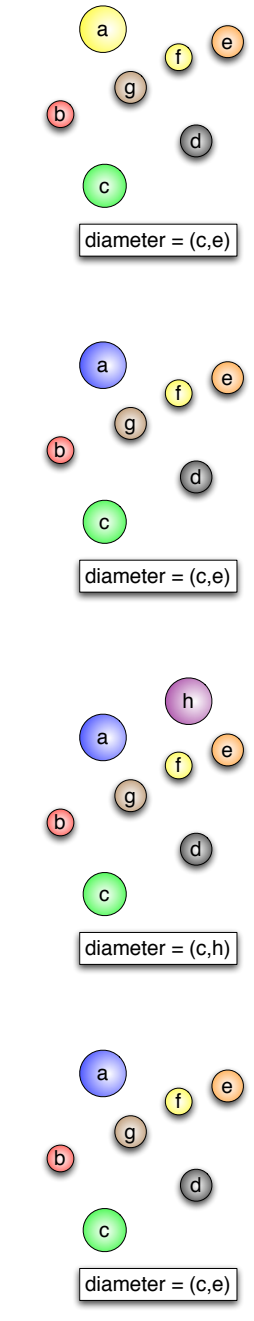

Fig. 3. An example interaction with circles

```
signature ADAPTIVE = sig
 type 'a box
 val put : 'a -$> 'a box
 val get : 'a box -$> 'a

 val mkPut : unit -$> ('k * 'a -$> 'a box)

 (** Meta operations **)
 val new : 'a -> 'a box
 val deref : 'a box -> 'a
 val change : 'a box * 'a -> unit

 datatype 'a res = Value of 'a | Exn of exn
 val call : ('a -$> 'r) * 'a -> 'r res ref
 val propagate : unit -> unit
end
structure Adaptive :> ADAPTIVE = struct ... end
```

Fig. 4. Signature for the `Adaptive` library.

conventional functions and *adaptive functions*. Conventional functions are declared using the conventional ML `fun` syntax for functions. Adaptive functions are declared with the `afun` and `mfun` keywords. The `afun` keyword declares an adaptive, non-memoized function; the `mfun` keyword declares an adaptive, memoized function. Adaptive functions (memoized and non-memoized) have the adaptive-function type τ_1 `-$>` τ_2. We use the infix operator `$` for applying adaptive (memoized or non-memoized) functions. An adaptive function must be written in the pure subset of SML but can operate on modifiables using the core primitives that we describe below.

An adaptive application can appear only in the body of an adaptive function. We can thus partition a self-adjusting program into a set of adaptive functions, A, that call each other and other (pure) conventional functions, and a set of conventional functions, C, that can only call other (possibly imperative) conventional functions. This distinction helps us statically identify the mutator and the self-adjusting core: the mutator consists of the conventional function set C, the core consists of the set of adaptive functions A and the conventional function called by them. [5]

Except for `afun` and `mfun` keywords, all other self-adjusting computation primitives are provided by a library. Figure 4 shows the interface to this library. The *box type* τ `box` is the type of a modifiable reference and serves as a container for changeable data. The `put:` α `-$>` α `box` primitive creates a box and places the specified value into the box, while the `get:` α `box -$>` α primitive returns the contents of a box. Since the primitives have adaptive function types, they can only be used within an adaptive function.

The `put` and `get` function are all we need to operate on modifiable references. For the purposes of improving efficiency the library provides an additional primitive: `mkPut`. The `mkPut` operation returns a *putter* that can be used to perform

[5] This distinction also helps improve the efficiency of the compilation mechanisms: we need to treat/compile only adaptive functions specially, other functions are compiled in the conventional manner.

keyed allocation. The putter takes a key and a value to be boxed and returns a box (associated with the key) holding that value. Section 7 describes the motivation for `mkPut` and how it can be used to improve the efficiency of change propagation.

To facilitate the mutator to perform initial run and to operate on changeable data, we provide several *meta primitives*. These meta primitives are impure and should not be used within adaptive functions; doing otherwise can violate the correctness of change propagation. [6]

To perform an initial run, the interface provides the `call` primitive. At the meta-level the result of an initial run is either an ordinary value or an exception. More concretely, the `call` operation takes an adaptive function and an argument to that function, calls the function, and returns its results or the raised exception in an ordinary reference. Note that the `call` operation is the only means of "applying" an adaptive function outside the body of another adaptive function. The result of the `call` operation is a reference cell containing the output (distinguishing between normal and exceptional termination) of the self-adjusting computation.

The mutator uses the `new`, `change`, and `deref` operations to create and modify inputs, and to inspect outputs of a self-adjusting computation. The `new` operation places a value into box—it is the meta-version of the `put` operation. The `deref` operation returns the contents of a box—it is the meta-version of the `get` operation. The `change` operation takes a value and a box and modifies the contents of the box to the given value by a destructive update.

A typical mutator starts by setting up the input and calling an adaptive function to perform an initial run. Interacting with the environment, it then modifies the input of the adaptive function or other data created by the initial run, and performs change propagation by calling the meta primitive `propagate`. When applied, the `propagate` primitive incorporates the effects of the `change` operations executed since the beginning of the computation or the last call to `propagate`.

6 Implementing CIRCLES

We describe a full implementation of CIRCLES in ΔML. Since self-adjusting computation facilitates interaction with modifications to data via change propagation, we only need to develop a program for the static case where the input list of circles does not change. We then use a mutator to drive the interaction of this program with the user. We first describe the interfaces of libraries for modifiable lists and geometric data structures. We then describe the implementation of the mutator and the core itself. To ensure efficiency, the core uses the quick-hull algorithm for computing convex hulls as a subroutine, which we describe last.

[6] Our compiler does not enforce statically this correct-usage principle.

```
signature MOD_LIST =
sig
  datatype α cell = NIL | CONS of α * α modlist
  withtype α modlist = α t
  type α t = α modlist

  val lengthLessThan: int -> α t -$> bool box
  val map: (α -> β) -> α t -$> β t
  val filter: (α -> bool) -> α t -$> α t
  val reduce: (α -> α -> α) -> α t -> α -$> β t
end

signature POINT =
sig
  type t
  val fromXY: real * real -> t
  val toXY: t -> real * real
end

signature GEOMETRY =
sig
  structure Point : POINT
  type point
  type line = point * point

  val toLeft : point * point -> bool
  val toRight : point * point -> bool
  val dist : point * point -> real

  val lineSideTest: line * point -> bool
  val distToLine: point * line -> real
end

structure ModList: MOD_LIST = ... (see Figure 10)
structure Geom: GEOMETRY = ...
structure L = ModList
structure Point = Geom.Point
type circles = Point.t * (string * string * real)
```

Fig. 5. Modifiable lists, points, and the geometry library.

6.1 Lists and Geometric Data Structures

Figure 5 shows the interface of the libraries that we build upon as well as the definitions of some structures implementing them.

Modifiable Lists. We use a list data structure to represent the set of circles. Since we want to insert/delete elements from this list, we want the list to be modifiable, which we achieve by defining

```
datatype α cell = NIL | CONS of α * α modlist
withtype α modlist = α cell box
```

A modifiable list of type α `modlist` is a linked list of cells of type α `cell`. A cell is either empty (`NIL`) or a `CONS` of an element and a modifiable list. This definition of lists is similar to conventional lists, except that the tail component of a `CONS` cell is boxed. This enables the mutator to change the contents of a modifiable list by updating the tail modifiables.

As with conventional lists, we can define a number of standard operations on lists, as shown in Figure 5. The `lengthLessThan` function takes a number and a list and returns a boolean modifiable that indicates whether the length of the list is less than the specified number. The `map` function takes as arguments a function that can map an element of the list to a new value and a list; it returns the list obtained by applying the function to each element in the list. The `filter` function takes a predicate and a list and returns the list of elements of the input list that satisfies the predicate. The `reduce` function takes as arguments an associative binary operation defined on the elements of the list, a list, and a value to return when the list is empty; it returns the value obtained by combining all elements of the list using the operation.

In our implementation we assume a structure called `ModList` that implements the modifiable lists interface, i.e., `structure ModList:MOD_LIST`. For brevity, we abbreviate the name simply as `L`, i.e., `structure L = ModList`. Section 8 presents and implementation of `ModList`.

Geometric Data Structures and Operations. To operate on circles, we use some geometric data structures. Figure 5 shows the interface for a point data structure and a library of geometric operations. A point supports operations for translation from points into x-y coordinates. The geometry library defines a line as a pair of points and provides some operations on points and lines. The `toLeft` (`toRight`) operation returns `true` if and only if the first point is to the left (right) of the second, i.e., the former's x-coordinate is less (greater) than the latter's. The `dist` operation returns the distance between two points. The `lineSideTest` returns true if the point is above the line, that is the point lies in the upper half-plane defined by the line. The `distToLine` returns the distance between a point and a line.

In our implementation, we assume a structure `Geom` that implement the geometry primitives. For brevity, we define a point structure `Point` as `structure Point = Geom.Point`.

We define a circle type as a tuple consisting of a point (center), and a triple of auxilary information, i.e., a `string` id, a `string` color, and a floating point radius, i.e., `type circle = Point.t * (string * string * real)`.

6.2 Implementing the Mutator

Suppose that we have a self-adjusting function `processCircles` that takes a list of circles, finds their diameter, renders them, and returns the name of the circles on the diameter. The signature of `processCircles` can be written as:
`processCircles: ModList.t -> (string * string) box`.

```
fun mutator () =
let
  fun realFromString (str) = ...

  fun prompt target str =
  let val _ = print str
      val tokens = String.tokens Char.isSpace (TextIO.input TextIO.stdIn)
  in case tokens of
       t::nil => if t = target then NONE else SOME tokens
     | _ => SOME tokens
  end

  fun mkCircle tokens =
  let val [id, color,xs,ys,rs] = tokens
      val (x,y,r) = (realFromString xs,realFromString ys, realFromString rs)
  in (Point.fromXY (x,y),(id,color,r)) end

  fun readCs cs =
    case (prompt "done" ("Enter circle or type 'done' : /n")) of
       NONE => cs
     | SOME tk => let val c = mkCircle tk in readCs (new (L.CONS (c,cs))) end

  fun findAndModify cs id c =
  let fun isId (_,(id',_,_)) = id = id'
      fun find f l =
        case deref l of
          L.NIL => raise BadInput
        | L.CONS (h,t) => if f h then SOME l else find f t
      val SOME m = find isId cs
      val L.CONS (h,t) = deref m
  in change (m,L.CONS (c,t)) end

  fun modify cs =
    case (prompt "quit" "Enter the circle to modify or type 'quit'./n") of
      NONE => ()
    | SOME [id] =>
      let val SOME tokens = prompt "" ("Enter circle: /n")
          val c = mkCircle tokens
          val _ = findAndModify cs id c
          val _ = (print ``Propagating.../n''; propagate (); print ``Done./n'')
      in modify cs end

  fun main () =
  let val _ = init ()
      val _ = print "Welcome to circles!/n"
      val cs = readCs (new L.NIL)
      val _ = call (processCircles, cs)
      val _ = modify cs
  in () end

in main () end
```

Fig. 6. A mutator for circles.

Figure 6 shows a mutator that drives this self-adjusting program by interacting with the user in a style as shown in Figure 3. For brevity, we show here a simplified mutator that only allows the user to modify the properties of an existing circles—it does not support insertion or deletion (the complete mutator is provided in the source distribution). The mutator performs mostly standard tasks, such as prompting the user, reading data from the user, etc.; the self-adjusting computation primitives are underlined to assist with the discussion.

Let's dissect the mutator in a top down style. The `main` function is the interaction loop. It starts by initializing the self-adjusting computation system, prints a welcome message, and reads the circles from the user. It then calls the adaptive `processCircles` function with the user input to compute the diameter and render the circles on the screen. The computation then proceeds into a modify-propagate loop with function `modify`. The function `modify` asks the user to modify a circle, applies the modifications by calling `findAndModify`, and performs change propagation, which updates the computation. The function `findAndModify` traverses the input list using the meta operation `deref` (used to access the contents of modifiables) and updates the circle by using the `change` meta operation when it finds the circle to modify. [7] The `change` operation destructively updates the contents of the modifiable holding the circle being modified. The upper half of the mutator code, i.e., the functions `realFromString`, `prompt`, `mkCircle`, `readCs` are written using standard techniques. For brevity the code for `realFromString` is omitted.

As this example should make clear, the treatment of modifiables and changeable data at the meta-level (i.e., in the mutator code) is identical to that of references. We process modifiable lists as though modifiables are simple references. We start a self-adjusting computation by calling an adaptive function with the `call` meta operation, and perform change propagation with the `propagate` meta operation when the user changes the computation data.

6.3 Implementing the Core

Figure 7 shows the code for the self-adjusting core of CIRCLES. For the time being, the reader should ignore `mkPut` and read "`putM $ (_, v)`" as "`put $ v`"; the use of `putM` and `mkPut` is explained in Section 7.

The `circle` function takes the list of circles, renders them (with the `renderCircle` function), and computes their diameter (with the function `findDiameter`).

The `renderCircle` function traverses the list of circles and prints them on the screen. In this implementation, we simply print the properties of the circle—rendering a real image would be structurally identical. Two points about `renderCircle` differ from the same function operating on an ordinary list. First, being a modifiable list, the input is stored in a modifiable, and thus, we use the

[7] For improved efficiency in finding for the circle to be modified, we may want to use an auxilary search structure such as a hash-table that maps the circle id's to the cons cell that holds that circle.

```
fun map = ... (* See Figure 10 *)
fun reduce = ... (* See Figure 11 *)
fun quick_hull l = ... (* See Figure 9 *)

mfun renderCircles l =
let fun printC (c as (p,(id,color,r))) =
    let val (x,y) = Point.toXY p
        val s = id  ^ " : "  ^ color
                    ^ " ("  ^ Real.toString x  ^ ", "  ^ Real.toString y  ^ ")"
                      ^ " "  ^ Real.toString r  ^ "/n"
    in print ("Rendering circle = "  ^ s  ^ "/n"); end

in case get $ l of
      L.NIL => ()
    | L.CONS(h, t) => (printC h ; renderCircles t)
end

afun findDiameter l =
let val putM = mkPut $ ()
    fun maxDist (da as (_,va),db as (_,vb)) =
      if (Real.> (va,vb)) then da
      else db

    fun dist (a as (o_a, (id_a,_,r_a))) (b as (o_b, (id_b,_,r_b))) =
        Geom.dist (o_a,o_b) - r_a - r_b end

    mfun farthestFrom (c,l) =>
      let val dist = map (dist c) $ l
          val max = reduce maxDist $ dist
      in get $ max end

    mfun findAllDist l =
      let val putM = mkPut $ ()
      in case get $ l of
          L.NIL => putM $ (NONE, L.NIL)
        | L.CONS(h,t) => case get $ t of
                           L.NIL => putM $ (NONE,L.NIL)
                         | _ => let val m = farthestFrom $ (h,t)
                                 in putM $ (SOME m, L.CONS(m, findAllDist $ t))
                                end
      end

    val hull = quick_hull $ l
    val dist = findAllDist $ hull
    val max = reduce maxDist $ dist
    val ((ida,idb),_) = get $ max
    val _ = print ("diameter = : "  ^ ida  ^ " x "  ^ idb  ^ "/n")
in putM $ (NONE,(ida,idb)) end

afun processCircles l = (renderCircles l ; findDiameter l)
```

Fig. 7. The code for the core.

self-adjusting `get` primitive to access its contents. Second, the function is adaptive function instead of an ordinary function, because it uses a self-adjusting primitive (`get`). Since the function takes more than constant time (it takes linear time in the size of the list), we decided to make it memoized by declaring it with `mfun`.

When deciding what functions to memoize, we observe the following principle: if an adaptive function performs more than constant work excluding the work performed by memoized functions called within its body, then we memoize it.

The function `findDiameter` computes the diameter of the set of circles and prints it on the screen. We define the diameter as the maximum distance between any two circles, where the distance between two circles is the minimum distance between any points belonging to them (we can compute the distance between two circles by as the distance between their center points minus the sum of their radii). The function `dist` computes the distance between two circles. To compute the diameter, we need some auxilary functions.

The function `farthestFrom` takes a circle `c` and a list of circles `l` and returns the circle that is farthest away from `c`. The function computes its result by first calculating the distance between each circle in the list and `c` (by using `map` over the list of circles) and then selecting the maximum of the list (by using `reduce` on the list). Since the function performs calls to adaptive functions, it itself is an adaptive function. Since it takes more than linear time, we declare it memoized.

The function `findAllDist` computes for each circle in the input, the circle that is farthest away from it among the circles that come after it in the input list. Even `findAllDist` computes only half of all possible distances, it correctly finds the maximum distance because distance function is symmetric. Since the function takes more than constant time (more precisely, $O(m^2)$ time in the size of its input), we declare it memoized.

One way to compute the diameter is to compute the pairwise distances of all circles, using `findAllDist`, and then pick the maximum of these distance. This, however, would not be asymptotically efficient because it requires $\Theta(n^2)$ time in the size of the input list. Instead, we first compute the convex hull of the centers of the circles. We then compute the pairwise distances of the circles whose centers lie on the hull.

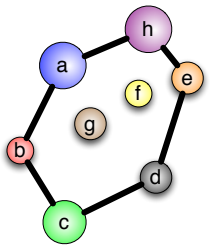

Fig. 8. An example convex hull.

The convex hull of a set of points is the smallest polygon enclosing the points. Figure 8 shows the convex hull for our running example. It is a property of the convex hulls that the pair of circles that are farthest away from each other is on the hull. To find the diameter of the circles efficiently, we can therefore first find the circles whose centers lie on the convex hull of the centers and consider only these circles when computing the farthest-apart pair. With this approach, we improve efficiency: computing convex hulls requires $O(n \log n)$ time and computing finding the diameter requires $O(h^2)$ time where h is the number of circles on the hull, which is generally significantly smaller than the input n. In our implementation, we use `quick-hull` to compute the convex hull. This algorithm is not asymptotically optimal but works well in practice.

```
fun select f (a,b) = if f (a, b) then a else b
fun above (cl as (pl,_), cr as (pr,_)) (c as (p,_)) =
  Geom.lineSideTest ((pl, pr), p)
fun distToLine (cl as (pl,_), cr as (pr,_)) (c as (p,_)) =
  Geom.distToLine (p, (pl,pr))

afun split (bcl, bcr, l, hull) =
  let val putM = mkPut $ ()
      mfun splitM (cl, cr, l, hull) =
        let val lf = L.filter (above (cl, cr)) $ l
        in if get $ (L.lengthLessThan 1 $ lf) then
             putM $ (cl, L.CONS (cl, hull))
           else let val dist = distToLine (cl,cr)
                    val selectMax = select (fn (c1,c2) => dist c1 > dist c2)
                    val max = get $ (combine selectMax $ lf)
                in splitM $ (cl, max, lf, splitM $ (max, cr, lf, hull)) end
        end
      val (cl,cr) = (get $ bcl, get $ bcr)
  in splitM $ (cl, cr, l, hull) end

afun quick_hull l =
  if get $ (L.lengthLessThan 2 $ l) then l
  else let fun isMin (c1 as (p1,_), c2 as (p2,_)) = Geom.toLeft (p1,p2)
           fun isMax (c1 as (p1,_), c2 as (p2,_)) = Geom.toRight (p1,p2)
           val min = combine (select isMin) $ l
           val max = combine (select isMax) $ l
           val lower = split $ (max, min, l, put $ L.NIL)
           val hull = split $ (min, max, l, lower)
       in hull end
```

Fig. 9. The code for quick-hull.

6.4 Implementing Quickhull

Many algorithms, some optimal, have been proposed for computing convex hulls. In this example, we use the quick-hull algorithm, which is known to perform well in practice, except with some distributions of points. Figure 9 shows the complete code for `quick-hull`. The algorithm uses several geometric primitives. The function `above` returns `true` if and only if the center of the circle `c` above the line defined by the centers of the circle `cl` and `cr`. The function `distToLine` returns the distance from the center of the circle `c` to the line defined by the centers of the circles `cl` and `cr`. These functions are standard—they perform no self-adjusting computation.

The `split` function takes two boxed circles, which define the *split-line*, a list of circles, and the partial hull constructed thus far, and it uses the auxiliary `splitM` function to extend the hull. The `splitM` function first eliminates the list of points below the split-line. If the remaining list is empty, then the hull consists of the left circle (`cl`) alone. If the remaining list is non-empty, then the function finds the circle, `max`, whose center is furthest away from the line defined by the split-line. The algorithm then recurses with the two split-lines defined by the left circle and the max, and the max and the right circle.

The function `quick-hull` computes the initial split-line to be the line whose endpoints are the left-most and the right-most circles (based on their centers). Based on this split-line, it then uses `split` to compute convex hull in two passes. The first pass computes the lower hull, the second pass computes the complete hull.

7 Performance

Self-adjusting computation enables writing programs that can respond to modifications to their data in the style of conventional/non-incremental programs. In this section, we describe how we may analyze the running time of self-adjusting programs for both from-scratch executions and change propagation, and how we may improve the performance for change propagation. Our approach is informal; a precise treatment is out of the scope of this tutorial but the interested reader can find more discussion in Section 10.2 and the papers referenced there.

For the purposes of this tutorial, we consider so-called *monotone* computations only. We define a computation to be monotone if 1) no function is called with the same arguments more than once, and 2) the modifications being considered do not affect the execution order and the ancestor-descendant relationships between function calls. Not all programs are naturally monotone (for interesting classes of input modifications) but all programs can be made so by adding (extra) arguments to function calls.

When analyzing the performance of self-adjusting programs, we consider two different performance metrics: running time of from-scratch execution and running time of change propagation.

7.1 From-scratch runs

Analyzing the from-scratch running time of self-adjusting programs is essentially the same as analyzing conventional programs: we treat the program as a conventional program where all (meta and core) self-adjusting computation primitives take expected constant time (the expectation is over internal randomization). More precisely, we can ignore self-adjusting computation primitives when analyzing the from-scratch running time, the actual running time is only slower by an expected constant factor. Thus, analysis of the from-scratch runnning time of self-adjusting programs is standard.

7.2 Stability and Change Propagation

To analyze the running time of change propagation, we first fix the input modification that we want change propagation to handle. For example, we can analyze the running time for change propagation in CIRCLES after inserting (or deleting) a new circle. Considering unit-size modifications (e.g., a single insertion, a single deletion) often suffices, because it is guaranteed that batch modifications never take more time than the sum of their parts. For example, if a the propagation caused by a single application of a modification (e.g., an insertion) requires $O(m)$ time then propagating k applications of that modification (e.g., k insertions) takes no more than $O(k \cdot m)$ time. Additionally, change propagation never takes asymptotically more than a from-scratch run (re-running the program from scratch).

To analyze the time for change propagation, we define a notion of *distance* between computations. Suppose that we run our program with some input I, then we modify the input to I' and perform change propagation to update the output. Intuitively, the time for change propagation will be proportional to the distance between the two runs.

Before defining distance between runs, we need to identify similar subcomputations. We define two calls of the same function to be *similar* if their arguments are the equal and their results are equal. We compare values (arguments and return values) by using structural equality up to modifiables: two values are equal if they are the same non-modifiable value or they are the same modifiable, e.g., the booleans `true` are `true` equal, the modifiable l is equal to itself (regardless of the contents), but the (distinct) modifiables l and l' are distinct (regardless of their contents), therefore the tuple `(true,`l`)` is equal to itself but distinct from `(true,`l'`)`. If the values are functions (closures) themselves, then we conservatively assume that they are never equal. Note that when computing the distance, we view modifiables as conventional references.

Next, we define the *difference* $X_1 \setminus X_2$ between executions X_1 and X_2 as the set of function calls in X_1 for which no similar call exists in X_2. We define *distance* between two monotone executions X_1 and X_2 as the size of the symmetric difference between two executions, i.e., $|(X_1 \setminus X_2) \cup (X_2 \setminus X_1)|$. We say that a program is $O(f(n))$-*stable* for some input change, if the distance between two executions of the program with any two inputs related by the change is

bounded $O(f(n))$, where n is the maximum input size. Informally, we say that a program is *stable* for some input change, if it is poly-logarithmically stable, i.e., $O(\log^c n)$-stable where c is some constant.

7.3 Programming for Stability

When calculating the stability of a program, we compare modifiables by their identity (i.e., physical location in memory). This makes stability measurements sensitive to the whims of non-deterministic memory allocation: if a program allocates memory in a way that is similar to that of the previous run, then it can be stable, if not, then it will not be stable [8]

To address this unpredictability of non-deterministic memory allocation, we label each modifiable with a unique key and define two modifiables to be equal if they have the same key. We support this memory model by performing memory allocation associatively. In particular, we maintain a hash table that maps keys to allocated locations. When performing an allocation, we first perform a memo lookup to see if there already is a location (from a previous run) associated with the specified key. If so, we re-use it. Otherwise, we allocate a fresh location and insert it into the memo table for re-use in subsequent runs. The idea is that every execution of the an allocation with the same key will return the same modifiable.

The ΔML language relaxes the requirement about keys being unique: different allocations can share the key. In practice, it often seems to suffice for keys to act as a guide for allocation to help improve sharing between computations, but it is not necessary for them to be unique. Requiring that allocations are uniquely keyed, however, simplifies the stability analysis, which we take advantage of in our discussions of stability.

ΔML supports this *keyed allocation* strategy through the `mkPut` function, which creates a fresh hash table for keyed allocation and returns an associated *putter* function. A putter takes two arguments: a key associated with the modifiable to be allocated and a value to place into the allocated modifiable. We often create putters locally for each function. Since each putter has its own hash function, creating local putters helps eliminate key collision between different functions: allocations performed by different putters can share the same keys without causing a collision.

8 Modifiable Lists

We now turn to the implementation of lists and their asymptotic complexity under change propagation. Figure 10 shows the definition and implementation of modifiable lists and some functions on modifiable lists. A modifiable list of type α `modlist` is a boxed cell where a cell is either empty or a `CONS` cell consisting of an element of type α and a modifiable list.

[8] Since change propagation re-executes only parts of a program, the contrast is less stark in reality. Still, non-determinism in memory allocation can detrimentally affect stability.

```
structure ModList =
struct
  datatype α cell = NIL | CONS of α * α modlist
  withtype α modlist = α cell box
  type α t = α modlist

  afun lengthLessThan (l: 'a modlist) : bool box =
  let putM = mkPut $ g()
      afun len (i,l) =
        if i >= n then false
        else case get $ l of
                 NIL => true
               | CONS(h,t) => len $ (i+1,t)
  in putM $ (NONE, len $ (0,l)) end

  afun map (f: 'a -> 'b) (l: 'a modlist) : 'b modlist =
  let val putM = mkPut $ ()
      mfun m l =
        case get $ l of
          NIL => NIL
        | CONS(h,t) => CONS (f h, putM $ (SOME h, m t))
  in putM $ (NONE, m l) end
end
```

Fig. 10. Some list primitives.

Modifiable lists are defined in a similar way to conventional lists, with the small but crucial difference that the tail of each cell is placed in a modifiable. This enables the mutator to modify the contents of the list by inserting/deleting elements. Thus, we can use modifiable lists to represent a set that changes over time.

As with conventional lists, we can write various primitive list functions. Here we describe how to implement three functions on lists: `lengthLessThan`, `map`, and `reduce`. Omitting the self-adjusting primitives, the first two functions are nearly identical to their conventional counterparts and therefore have comparable asymptotic complexity; the complexity of `reduce` is discussed below.

In our implementation, we observe the following convention. Each list function creates a putter and defines a locally-scoped *work function* to perform the actual work. When using putters, we key the tail modifiable of a `CONS` cell by `SOME h` where `h` is the head element. This identifies the modifiables by the head item in the same `CONS` cell. In our analysis, we assume that lists contain no duplicates—thus, the tail modifiables are uniquely identified by the head elements. For boxing the head of a list we use the key `NONE`.

The function `lengthLessThan` takes an integer and returns a boxed boolean indicating whether the length of the list is less than the supplied integer. Many self-adjusting programs use this function. For example, in `quick_hull` we check whether the input list has two or fewer elements and return the list directly if

so. This is often preferable to simply returning the length of the list, because some computations do not directly depend on the length of the list, but rather on whether the length is less than some value. Having this function allows us to express such a dependence more precisely, allowing for more efficient change propagation. The implementation of `lengthLessThan` follows our description: the outer function allocates a modifiable with `putM` for the result and calls the work function `len` with the argument.

The `map` function takes a function and a list and produces another list by applying the function to the elements of the input list. The outer function creates a putter by using `mkPut`. The work function `m` uses the head `h` of each `CONS` cell to key the tail modifiable with `SOME h`. The top-level call to the work function is boxed using the key `NONE`. This version of `map` is $O(1)$-stable with respect to single insertions or deletions.

Consider running `map` with the identity function on lists `[...,1,3,...]` and `[...,1,2,3,...]` where each list has no repetitions and both lists are identical except for the element `2` (which occurs at an arbitrary position). Recall that the head of a `CONS` cell is used to key the tail cell. For example, in the first list the element `1` is used as a key to allocate box l_1, which contains cell `CONS(3,`l_3`)`. Similarly, the result of `map` on the first list will have box l'_1 holding cell `CONS(3,`l'_3`)`. Therefore the work function on the first list will include calls:

$$\ldots, \texttt{m } l_0 \texttt{ = CONS(1,}l_1\texttt{)}, \texttt{m } l_1 \texttt{ = CONS(3,}l_3\texttt{)}, \qquad \texttt{m } l_3 \texttt{ = } \cdots, \ldots$$

and the work function on second list will include calls:

$$\ldots, \texttt{m } l_0 \texttt{ = CONS(1,}l_1\texttt{)}, \texttt{m } l_1 \texttt{ = CONS(2,}l_2\texttt{)}, \texttt{m } l_2 \texttt{ = CONS(3,}l_3\texttt{)}, \texttt{m } l_3 \texttt{ = } \cdots, \ldots$$

Therefore, the two runs differ only by three calls: `m` l_1 `= CONS(3,`l_3`)`, `m` l_1 `= CONS(2,`l_2`)`, and `m` l_2 `= CONS(3,`l_3`)`, so `map` is $O(1)$ stable for single insertions or deletions.

Many programs, e.g., most of those considered in this tutorial, are naturally stable or can be made stable with relatively small changes. When the program is not stable, we often need to choose a different algorithm.

As an example of a program that is not naturally, consider the list function `reduce` takes a list and an associative binary operator, and produces a single value by applying the operator to the elements of the list, e.g., applying `reduce [1,2,3,4]` with addition operation yields `10`. A typical implementation of `reduce` (shown below) traverses the list from left to right while maintaining an accumulator of the partial results for the visited prefix.

```
afun reduce f base l =
let mfun red (l,a) =
      case get $ l of NIL => put a
                    | CONS(h,t) => red $ (t,f(h,a))
in red $ (l,base) end
```

```
fun reduce mkRandSplit f base = afn l =>
let afun halfList l =
    let val putM = mkPut $ ()
        val randSplit = mkRandSplit ()

        afun redRun (v,l) =
          case get $ l of
            NIL => (v, l)
          | CONS (h, t) =>
              if randSplit h then (f (v,h), t)
              else redRun $ (f(v,h), t)

        mfun half l =
          case get $ l of
            NIL => NIL
          | CONS (h, t) =>
              let val (v, tt) = redRun $ (h,t)
                  val ttt = half $ tt
              in CONS(v, putM $ (SOME h, ttt)) end
      in putM $ (NONE, half $ l) end

  mfun mreduce l =
  let val putM = mkPut $ ()
    if get $ ((lengthLessThan 2) $ l) then
      case get $ l of
        NIL => base
      | CONS(h,_) => h
    else mreduce $ (halfList $ l)

in putM $ (NONE, mreduce $ l) end
```

Fig. 11. The code for list reduce.

This implementation of `reduce` is not stable. To see why let's consider adding integer elements of a list with inputs that differ by a single key. More precisely consider the case when one list has one extra element at the beginning, e.g., `[1,2,3,4,...]` and `[2,3,4,...]`. The function calls performed have the form, `red` $(l_i,\ a_i)$ where l_i is the modifiable pointing to the i^{th} cons cell and a_i is the sum of the elements in the first $i-1$ cells. With the first input, the accumulators are $a_i =$ (0, 1, 3, 6, 10, ...). With the second input, the accumulators are one more than the corresponding accumulator in the first case, $a_i =$ (0, 2, 5, 9, ...). Thus, no two prefix sums are the same and a linear number of operations (in the length of the input) will differ. Consequently, the algorithm is linear stable, i.e., change propagation performs just as good re-computing from scratch.

Figure 11 shows a logarithmic stable solution for `reduce`. This implementation uses the classic technique of random-sampling to compute the result. The idea is to "halve" the input list into smaller and smaller lists until only a single element remains. To halve the list (`halfList`), we choose a randomly selected

subset of the list and combine the chosen elements to their closest element to the left (`redRun`). Note that a deterministic approach, where, for example the elements are combined in pairs, is not stable, because deleting/inserting an element can cause a large change by shifting the positions of many elements by one. Note that we do not require commutativity—associativity alone suffices. For randomization, we use a random hash function that returns 0 or 1 with probability 1/2.

The approach requires expected linear time because each application of `halfList` reduces the size of the input by a factor of two (in expectation). Thus, we perform a logarithmic number of calls to `halfList` with inputs whose size decreases exponentially.

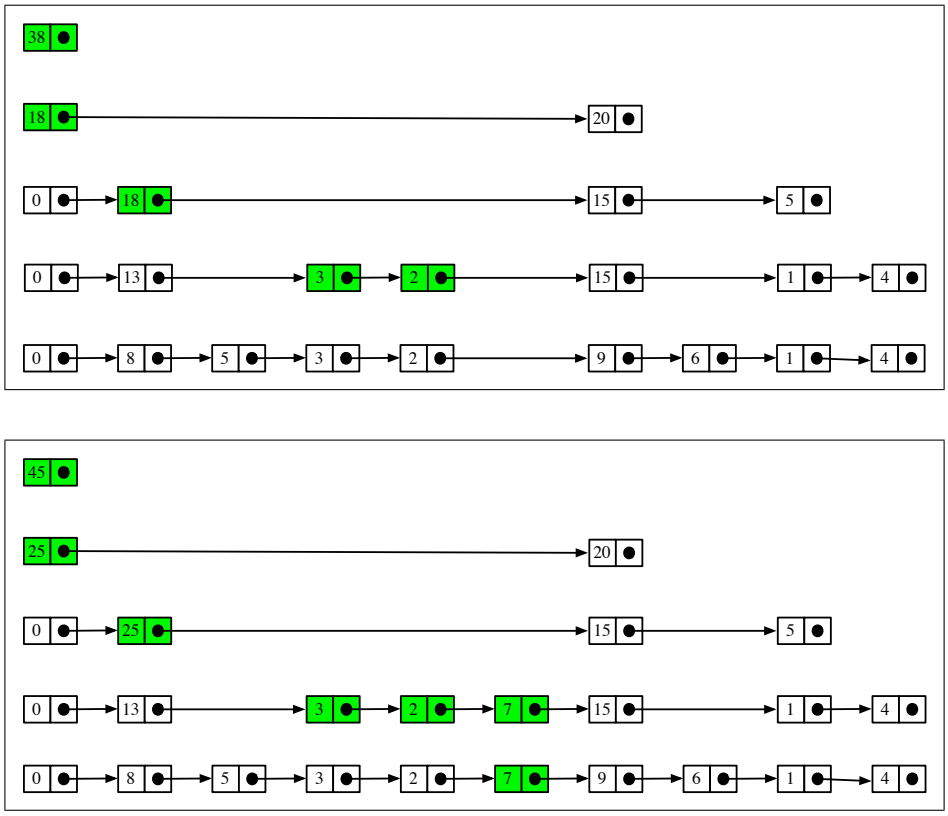

Fig. 12. Example stable list-reduce.

The approach is $O(\log n)$-stable in expectation (over internal randomization of `mkRandomSplit`). We do not prove this bound here but give some intuition by considering an example. Figure 12 shows executions of `reduce` with lists

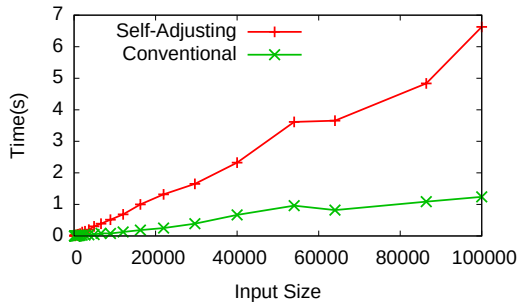

Fig. 13. Time for a from-scratch runs conventional and self-adjusting.

`[0,8,5,3,2,9,6,4,1)]` and `[0,8,5,3,2,7,9,6,4,1)]` to compute the sum of the integers in the lists. The two lists differ by box `7`. Comparing two executions, only the function calls that touch the highlighted cells differ. It is not difficult to show that there are a constant number of such cells in each level and, based on this, to prove the $O(\log n)$-stability bound.

9 Experimental Results

We present a simple experimental evaluation of CIRCLES. In our evaluations, we compare two versions of CIRCLES: conventional and self-adjusting. The code for self-adjusting version has been presented in this tutorial. The code for the conventional version is automatically derived from the self-adjusting version by removing the primitives on modifiable references and replacing (memoized and non-memoized) adaptive functions with conventional functions.

All of the experiments were performed on a 2.66Ghz dual-core Xeon machine, with 8 GB of memory, running Ubuntu Linux 7.10. We compiled the applications with our compiler with the option "-runtime ram-slop 0.75," directing the run-time system to allocate at most 75% of system memory. Our timings measure the wall-clock time (in seconds).

Figure 13 shows the run-time for executing the conventional and from-scratch versions of CIRCLES with up to 100,000 circles. We generate the input to the these executions randomly by picking the center of the circle uniformly randomly from within a unit square and picking a uniformly random radius between 0.0 and 1.0. The measurement shows that the self-adjusting version is about 5 times slower than the conventional version. A significant portion of this overhead is garbage collection: when excluding garbage-collection time self-adjusting version is about 3.5 times slower than the conventional. Although the overhead may be improved significantly by more careful compiler optimization techniques directed to self-adjusting programs (which the current implementation of our compiler does not employ), we consider it to be acceptable, because, in self-adjusting

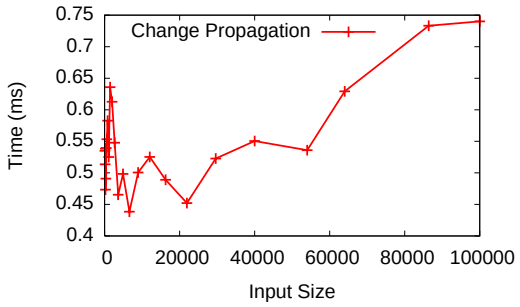

Fig. 14. Time for change propagation.

computation, from-scratch executions are infrequent. In typical usage, we execute a self-adjusting program from scratch once and then perform all subsequent updates to the input by utilizing change propagation.

Figure 14 shows the average time for change propagation for a single insertion/deletion for varying input sizes. The measurement is performed by repeating an update-step for each circle in order. In each step, we remove the circle from the input and perform change propagation. We then re-insert the circle and perform change propagation. We compute the average time for an insertion/deletion as the total time it takes to apply the update step to each element in the list divided by the number of steps, i.e., $2n$ where n is the number of circles in the input. As the figure shows the time for change propagation is rather uneven but increases very slowly over time (less than doubles between the smallest and the largest input sizes). The reason for the unevenness is the input-sensitive nature of CIRCLES: the time to update the diameter critically depends on the number of circles that are on the hull, which also determines the update time for quick-hull.

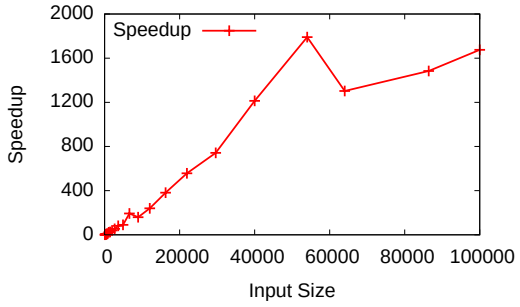

Fig. 15. Speedup of change propagation

Figure 15 shows the speedup measured as the time for a from-scratch execution of the conventional version of CIRCLES divided by the time for change propagation. As can be seen, change propagation appears to be faster by an asymptotic linear factor, delivering increasingly significant speedups. The speedups for larger inputs exceed three orders of magnitude. The speedups are significant even with relatively small input sizes.

10 Related Work

The problem of having computation respond to small modifications to their data has been studied extensively in several communities. Earlier work in the programming-languages community, broadly referred to as *incremental computation*, focused on developing techniques for translating static/conventional programs into incremental programs that can respond automatically to modifications to their input. Recent advances on self-adjusting computation generalized these approaches and dramatically improved their effectiveness. In the algorithms community, researchers proposed so called dynamic and kinetic data structures for addressing incremental problems. In this section, we briefly overview the earlier work on incremental computation (Section 10.1), recent work on self-adjusting computation (Section 10.2), and some of the work on algorithms community (Section 10.3).

10.1 Incremental Computation

Incremental computation offers language-centric techniques for developing programs that can automatically respond to modifications to their data. The most effective techniques are based on dependence graphs, memoization, and partial memoization.

Dependence-graph techniques record the dependences between data in a computation and rely on a change-propagation algorithm to update the computation when the input is modified. Demers, Reps, and Teitelbaum [27] and Reps [57] introduced the idea of *static dependence graphs* and presented a change-propagation algorithm for them. Hoover generalized the approach outside the domain of attribute grammars [45]. Yellin and Strom used the dependence graph ideas within the INC language [69], and extended it by having incremental computations within each of its array primitives. Static dependence graphs have been shown to be effective in some applications, e.g., syntax-directed computations. But they are not general-purpose, because they do not permit the change-propagation algorithm to update the dependence structure. For example, the INC language [69], which uses static dependence graphs for incremental updates, does not permit recursion.

The limitations of static dependence graphs motivated researchers to consider alternative approaches. Pugh and Teitelbaum [55] applied memoization (also called function caching) to incremental computation. Memoization, a classic idea that goes back to the late fifties [21, 50, 51], applies to any purely functional

program and therefore is more broadly applicable then static dependence graphs. Since the work of Pugh and Teitelbaum, others have investigated applications of various forms of memoization to incremental computation [1, 49, 43, 61, 6]. The idea behind memoization is to remember function calls and their results and re-use them when possible. In the context of incremental computation, memoization can improve efficiency when executions of a program with similar inputs perform similar function calls. Although the reader may expect this to be intuitively the case, it turns out that the effectiveness of memoization critically depends on the structure of the program and the kind of the input modification. For a given computation, it is often possible to find input modifications that prevent a large fraction of function calls from being re-used. Intuitively, the problem is that with memoization all function calls that consume a modified data and all their ancestors in the function call tree need to be re-executed (because these functions will have modified arguments).

Other approaches to incremental computation are based on partial evaluation [65, 32]. Sundaresh and Hudak's approach [65] requires the user to fix the partition of the input that the program will be specialized on. The program is then partially evaluated with respect to this partition and the input outside the partition can be modified incrementally. The main limitation of this approach is that it allows input modifications only within a predetermined partition. Field [33], and Field and Teitelbaum [32] present techniques for incremental computation in the context of lambda calculus. Their approach is similar to Hudak and Sundaresh's, but they present formal reduction systems that optimally use partially evaluated results.

10.2 Self-Adjusting Computation

Foundations. The first work on self-adjusting computation [5], called Adaptive Functional Programming (AFP), generalized dependence-graph approaches by introducing *dynamic dependence graphs* (*DDGs*), by providing a change propagation algorithm for DDGs, and by offering a technique for constructing DDGs from program executions. Change propagation with DDGs is able to update the dependence structure of the DDG by inserting and deleting dependences as necessary. This makes it possible to apply the approach to any purely functional program. Type-safe linguistic facilities for writing adaptive programs guarantee safety and correctness of change propagation. A prototype implementation in SML was provided but the implementation did not enforce the safety properties statically. Carlsson gave a safe implementation of the proposed linguistic facilities in Haskell [23].

Although DDGs are general purpose, their effectiveness is limited: certain modifications can require as much time as re-computing from scratch. Subsequent work identified a duality between DDGs and memoization and provided linguistic and algorithmic techniques for combining them [4]. The linguistic techniques enable annotating adaptive programs [5] with particular memoization constructs. The algorithms provide efficient re-use of computations (via change

propagation) without significantly slowing down a from-scratch run. An experimental evaluation showed that the approach can speedup computations by orders of magnitude (increasing linearly with the input size) while causing the initial run to slowdown by moderate amounts. Follow-up work gave a formal semantics for combining memoization and DDGs and proved it correct with mechanically verified proofs [12].

The aforementioned work on self-adjusting computation assumes a form of purely functional programming. Although modifiables are in fact a form of references, they cannot be updated destructively by self-adjusting programs—only the mutator is allowed to update modifiables destructively. This can limit the applicability of the approach to problems that are suitable for purely functional programs only. Recent work [3] showed that self-adjusting computation may be extended to programs that update memory imperatively by proposing updateable modifiable references, which ΔML supports (we do not discuss updateable references in this tutorial).

The aforementioned approaches to self-adjusting computation rely on specialized linguistic primitives that require a monadic programming style. These primitives enable tracking dependences selectively, i.e., only the dependences on data that can change over time are tracked. If selective dependence tracking is not needed, then it is possible to track all dependences without requiring programmer annotations [2]. The monadic primitives make it cumbersome to write self-adjusting program and also require substantial restructuring of existing code. Recent work developed direct language support self-adjusting computation and provided a compiler for the language [48], which ΔML is based on. Having direct language and compiler support not only simplifies developing self-adjusting programs but also makes it possible to give a precise cost-semantics that enables programmer to analyze the time for change propagation [47].

All of the aforementioned work on self-adjusting computation extends type-safe, high level languages such as Standard ML and Haskell. In an orthogonal line of work, we develop techniques for supporting self-adjusting programs with low-level languages such as C [39]. Low level language present some challenges, because self-adjusting-computation primitives are higher order and they require tracking side effects. They do, however, offer more explicit cost model and some interesting opportunities for improving performance. For example memory management and change propagation may be integrated to support garbage collection without traversing memory [38].

Applications. Self-adjusting computation has been applied to a number of problems from a broad set of application domains such as motion simulation, machine learning, and incremental invariant checking.

Some of these applications are developed and implemented using the linguistic techniques described in earlier work and in this paper. One such class of applications is motion simulators for various geometric properties. In these applications, we employ a mutator that starts with an initial run of a self-adjusting program. The mutator then performs motion simulation by maintaining a sim-

ulation time and modifying the outcomes of comparisons performed between moving objects (in accordance with the specified motion plans), updating the computation via change propagation. Using this approach, we obtain motion simulators from purely functional programs by translating them into self-adjusting programs and by combining them with the mutator for motion simulation. Since change propagation is fully general and can handle arbitrary modifications to the input, the approach enables processing a broad range of modifications during motion simulation. It also helps address efficiently some critical robustness issues that arise in motion simulation. Previous work applies the approach to a number of problems from two [10] and three dimensions [8, 9]. The solutions on three-dimensional problems made progress on well-known open problems.

Other applications use self-adjusting computation to obtain dynamic algorithms for specific problems. One such application is the tree contraction algorithm of Miller and Reif [52], whose self-adjusting version provides an efficient solution to the problem of dynamic trees [7, 11]. We applied the same approach to statistical inference, a classic problem in machine learning, to show that statistical inference can be performed under incremental modifications efficiently [13, 14]. These results made progress on open problems related to incremental updating of inferred statistical properties as the underlying models change. In all these applications, the mutators typically perform discrete modifications such as insertions/deletions of edges and nodes into/from a tree, a graph, or a statistical model.

Shankar and Bodik [62] adapted self-adjusting computation techniques (more specifically the approach presented in an earlier paper [4]) for the purposes of incremental invariant checking in Java. In their approach, the Java program acts as a mutator by modifying the contents of memory. Such modifications trigger reevaluation of program invariants, via change propagation, expressed in a separate purely functional language. Instead of tracking dependences selectively, they track all dependences by treating all memory cells as modifiables. This treatment of memory is similar to a formulation of self-adjusting computation proposed in the first author's thesis [2]. They show that the approach can dramatically speedup invariant checking during evaluation.

10.3 Dynamic Algorithms

In the algorithms community, researchers approach the problem of incremental computation from a different perspective. Rather than developing general-purpose techniques for transforming static programs to incremental programs that can respond to modifications to their data, they develop so called *dynamic algorithms* or *dynamic data structures* (e.g., [64, 25, 30]). Dynamic data structures facilitate the user to modify the data by making small modifications, e.g., inserting/deleting elements. For example a dynamic data structure for computing the diameter (points furthest away from each other) allows the user to insert/delete points into/from a set of points while updating the diameter accordingly. We considered other example dynamic algorithms in Section 2.

By taking advantage of the structure of the particular problem being considered, the algorithmic approach facilitates designing efficient, often optimal algorithms. In fact, previous work shows that there is often a linear-time gap between a dynamic algorithm and its static version in terms of responding to incremental modifications to data. Taking advantage of incrementality, however, often comes at an increased complexity of design, analysis, and implementation. Dynamic algorithms can be significantly more difficult to design, analyze, and implement than their static counterparts. For example, efficient algorithms for computing the convex hull of a set of points in the plane are relatively straightforward. Efficient dynamic algorithms for convex hulls that can respond to incremental modifications (e.g., insertion/deletion of a point), however, are significantly more complicated. In fact this problem has been researched since the late 70's (e.g., [54, 53, 42, 19, 24, 22, 16]). Similarly, computing efficiently the diameter of a point set as the point set changes requires sophisticated algorithms (e.g., [46, 28, 29, 58]). Convex hulls and diameters are not the exception. Another example is Minimum Spanning Trees (MST), whose dynamic version has required more than a decade of research to solve efficiently [34, 31, 41, 40, 44], while its static/conventional version is straightforward. Other examples include the problem of dynamic trees, whose various flavors have been studied extensively [63, 64, 26, 56, 40, 68, 17, 35, 18, 67, 66].

Because dynamic algorithms are designed to support a particular set of modifications, they are highly specialized (an algorithm may be efficient for some modifications to data but not others), naturally more complex than their static versions, and are not composable (Section 2). These properties make them difficult to adapt to different problems, implement, and use in practice.

Algorithms researchers also study a closely related class of data structures, called *kinetic data structures*, for performing motion simulations efficiently [20]. These data structures take advantage of the incremental nature of continuous motion (Section 2) by updating computed properties efficiently. Many kinetic data structures have been proposed and some have also been implemented (e.g., [15, 36] for surveys). These data structures share many characteristics of dynamic data structures. They also pose additional implementation challenges [15, 37, 60, 59], due to difficulties with motion modeling and handling of numerical errors.

11 Conclusion

This tutorial presents a gentle introduction to the ΔML (Delta ML) language. The compiler and the source code for the examples may be reached via the authors web pages.

References

1. Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and Caching of Dependencies. In *Proceedings of the International Conference on Functional Programming*, pages 83–91, 1996.

2. Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.
3. Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, 2008.
4. Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.
5. 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.
6. 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.
7. Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms*, 2004.
8. Umut A. Acar, Guy E. Blelloch, and Kanat Tangwongsan. Kinetic 3D Convex Hulls via Self-Adjusting Computation (An Illustration). In *Proceedings of the 23rd ACM Symposium on Computational Geometry (SCG)*, 2007.
9. Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, September 2008.
10. Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic Algorithms via Self-Adjusting Computation. In *Proceedings of the 14th Annual European Symposium on Algorithms*, pages 636–647, September 2006.
11. Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*, 2005.
12. Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. In *Proceedings of the 16th Annual European Symposium on Programming (ESOP)*, 2007.
13. Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian Inference. In *Neural Information Processing Systems (NIPS)*, 2007.
14. Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Inference on General Graphical Models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008.
15. Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.
16. Giora Alexandron, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. In *9th Workshop on Algorithms and Data Structures (WADS). Lecture Notes in Computer Science*, volume 3608, pages 269—281, aug 2005.
17. Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming*, pages 270–280, 1997.

18. Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully-dynamic trees with top trees, 2003. The Computing Research Repository (CoRR)[cs.DS/0310065].
19. Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 747–756. Society for Industrial and Applied Mathematics, 1997.
20. Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.
21. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
22. 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.
23. Magnus Carlsson. Monads for Incremental Computing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming*, pages 26–35. ACM Press, 2002.
24. Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proceedings of the the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 92–99, 1999.
25. Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.
26. 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.
27. 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.
28. David Eppstein. Average case analysis of dynamic geometric optimization. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 77–86, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.
29. David Eppstein. Incremental and decremental maintenance of planar width. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 899–900, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.
30. David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.
31. David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.
32. 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.
33. John Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems*. PhD thesis, Department of Computer Science, Cornell University, November 1991.
34. Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14:781–798, 1985.
35. Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997.

36. L. Guibas. Modeling motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004.
37. Leonidas Guibas and Daniel Russel. An empirical comparison of techniques for updating delaunay triangulations. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 170–179, New York, NY, USA, 2004. ACM Press.
38. Matthew A. Hammer and Umut A. Acar. Memory management for self-adjusting computation. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 51–60, 2008.
39. Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.
40. Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.
41. Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 594–604. Springer-Verlag, 1997.
42. John Hershberger and Subhash Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32(2):249–267, 1992.
43. Allan Heydon, Roy Levin, and Yuan Yu. Caching Function Calls Using Precise Dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 311–320, 2000.
44. Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.
45. Roger Hoover. *Incremental Graph Evaluation*. PhD thesis, Department of Computer Science, Cornell University, May 1987.
46. Ravi Janardan. On maintaining the width and diameter of a planar point-set online. In *ISA '91: Proceedings of the 2nd International Symposium on Algorithms*, pages 137–149, London, UK, 1991. Springer-Verlag.
47. Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.
48. Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming*, 2008.
49. Yanhong A. Liu, Scott Stoller, and Tim Teitelbaum. Static Caching for Incremental Computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.
50. John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.
51. D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, 1968.
52. Gary L. Miller and John 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.
53. Mark H. Overmars and Ja van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

54. F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Commun. ACM*, 22(7):402–405, 1979.
55. 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.
56. Tomasz Radzik. Implementation of dynamic trees with in-subtree operations. *ACM Journal of Experimental Algorithms*, 3:9, 1998.
57. Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 169–176, 1982.
58. G. Rote, C. Schwarz, and J. Snoeyink. Maintaining the approximate width of a set of points in the plane. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 258–263, 1993.
59. Daniel Russel. *Kinetic Data Structures in Practice*. PhD thesis, Department of Computer Science, Stanford University, March 2007.
60. Daniel Russel, Menelaos I. Karavelas, and Leonidas J. Guibas. A package for exact kinetic data structures and sweepline algorithms. *Comput. Geom. Theory Appl.*, 38(1-2):111–127, 2007.
61. João Saraiva, S. Doaitse Swierstra, and Matthijs F. Kuiper. Functional incremental attribute evaluation. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 279–294, London, UK, 2000. Springer-Verlag.
62. Ajeet Shankar and Rastislav Bodik. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming language Design and Implementation*, 2007.
63. Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
64. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
65. R. S. Sundaresh and Paul Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, 1991.
66. Robert Tarjan and Renato Werneck. Dynamic trees in practice. In *Proceeding of the 6th Workshop on Experimental Algorithms (WEA 2007)*, pages 80—93, 2005.
67. Robert Tarjan and Renato Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2005.
68. Robert E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:167–177, 1997.
69. D. M. Yellin and R. E. Strom. INC: A Language for Incremental Computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991.