An Experimental Analysis of Self-Adjusting Computation

UMUT A. ACAR
Toyota Technological Institute at Chicago
GUY E. BLELLOCH
Carnegie Mellon University
MATTHIAS BLUME
Toyota Technological Institute at Chicago
and
ROBERT HARPER, and KANAT TANGWONGSAN
Carnegie Mellon University

Recent work on adaptive functional programming (AFP) developed techniques for writing programs that can respond to modifications to their data by performing *change propagation*. To achieve this, executions of programs are represented with *dynamic dependence graphs* (DDGs) that record data dependences and control dependences in a way that a change-propagation algorithm can update the computation as if the program were from scratch, by re-executing only the parts of the computation affected by the changes. Since change-propagation only re-executes parts of the computation, it can respond to certain incremental modifications asymptotically faster than recomputing from scratch, potentially offering significant speedups. Such asymptotic speedups, however, are rare: for many computations and modifications, change propagation is no faster than recomputing from scratch.

In this article, we realize a duality between dynamic dependence graphs and memoization, and combine them to give a change-propagation algorithm that can dramatically increase computation reuse. The key idea is to use DDGs to identify and re-execute the parts of the computation that are affected by modifications, while using memoization to identify the parts of the computation that remain unaffected by the changes. We refer to this approach as self-adjusting computation. Since DDGs are imperative, but (traditional) memoization requires purely functional computation, reusing computation correctly via memoization becomes a challenge. We overcome this challenge with a technique for remembering and reusing not just the results of function calls (as in conventional memoization), but their executions represented with DDGs. We show that the proposed approach is realistic by describing a library for self-adjusting computation, presenting efficient algorithms for realizing the library, and describing and evaluating an implementation. Our experimental evaluation with a variety of applications, ranging from simple list primitives to more sophisticated computational geometry algorithms, shows that the approach is effective in practice:

The work of U. A. Acar was supported in part by the Intel Corporation.
Contact author's email address: ktangwon@cs.cmu.edu

DOI 10.1145/1596527.1596530 http://doi.acm.org/10.1145/1596527.1596530

compared to recomputing from-scratch; self-adjusting programs respond to small modifications to their data orders of magnitude faster.

Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages, Performance, Algorithms

Additional Key Words and Phrases: Computational geometry, dynamic algorithms, dynamic dependence graphs, memoization, performance, self-adjusting computation

ACM Reference Format:

Acar, U. A., Blelloch, G. E., Blume, M., Harper, R., and Tangwongsan, K. 2009. An experimental analysis of self-adjusting computation. ACM Trans. Program. Lang. Syst. 32, 1, Article 3 (October 2009), 53 pages.

DOI = 10.1145/1596527.1596530 http://doi.acm.org/10.1145/1596527.1596530

1. INTRODUCTION

Many applications must interact and respond to data that incrementally changes over time: 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. Applications that interact with the user observe their application data change incrementally as a result of user issued modifications (e.g., in software development, programmers make many small changes to the source code, typically recompiling the program after each revision). Applications that perform motion simulation, where objects move continuously over time, causing incremental, continuous modifications to the property being computed, must respond to such modifications correctly and efficiently. All these applications naturally arise in a number of domains, including software systems, graphics, robotics, databases, and scientific computing.

In these applications, it has been observed that incremental modifications to data often require only small modifications to the output. Therefore, if we have techniques for quickly identifying the parts of the output that are affected by the modifications and updating them while reusing the rest of the output, we will be able to respond to these changes significantly faster than recomputing the output entirely from scratch. Devising such techniques has been a subject of active research in the algorithms and the programming-languages communities.

In the algorithms community, researchers design algorithms that efficiently maintain the output for a specific problem as the input undergoes small modifications. These algorithms, known as *dynamic data structures* or *dynamic algorithms*, have been studied extensively over the course of the last several decades. A closely related class of data structures, called *kinetic data structures*, aims to efficiently compute properties of continuously moving objects. Dynamic and kinetic data structures are designed on a per-problem basis, by taking advantage of the particular structure of that problem. Thus, they are usually highly efficient, even optimal. They can, however, be difficult to design and implement, even for problems that are trivial in the conventional setting. For a history of work on dynamic and kinetic algorithms, and for some examples

of dynamic problems whose conventional versions are straightforward, we refer the readers to Section 10.2.

In the programming-languages community, researchers have made significant progress on run-time and compile-time approaches for automatically dynamizing or incrementalizing conventional, static algorithms. These approaches, broadly called *incremental computation*, record and maintain certain information during an execution of a static algorithm so that the output can be updated efficiently as the input undergoes small changes. When incremental computation is made to work, it has some important advantages over dynamic and kinetic algorithms: for example, there is no need to design, implement, debug, document, and maintain separate static and dynamic algorithms for every problem—dynamic problems can simply be reduced to a static problem by using the proposed language-based techniques.

Most effective approaches to incremental computation are based on dependence graphs and memoization (see Section 10.1 for a more detailed discussion). The idea behind dependence-graph-based approaches is to record the data and control dependences in a computation so that a *change-propagation* algorithm can update the computation by identifying the parts that are affected by the modifications and rebuilding only those parts. Previous work on dependence graphs include the work of Demers et al. [1981] on *static dependence graphs*, which can be used to incrementalize certain computations only, and the work of Acar et al. [2006] on *dynamic dependence graphs*, which can be used to incrementalize any purely functional program. The idea behind memoization (also called *function caching*), a classic technique that dates back to the 60's [Bellman 1957; McCarthy 1963; Michie 1968], is to remember the results of function calls and reuse them when possible, instead of re-evaluating the functions. Pugh and Teitelbaum [1989] were the first to apply memoization to incremental computation, making it the only general-purpose technique that can incrementalize any purely functional program at that time.

Dynamic dependence graphs and memoization offer two different ways to incrementalize purely functional programs. Neither approach on its own, however, is effective in general. For example, consider the simple list-map primitive, which applies an operation to each list's element to generate a new list. Both approaches take linear time on average to update the output list after a new element is inserted at a random position; this update time is asymptotically the same as re-executing the whole list-map computation from scratch.

The primary reason that dynamic dependence graphs and memoization remain ineffective in general has to do with the granularity at which they operate on computations: both techniques reuse or re-execute computations at the granularity of *function-call trees*. More precisely, with memoization, we reuse the result of a function call in place of another identical call, effectively reusing the work of that function and all the function calls that it transitively performs, which collectively form a function-call tree. With dynamic dependence graphs, change propagation repeatedly identifies a function call that is affected by modifications and re-executes it until no such functions remain. When a function call is re-executed, all the calls that it transitively invokes are removed and replaced. Thus, for both approaches to be effective, a computation must yield

similar function-call trees on similar input. This, however, is not what typically happens. For instance, in the list-map example we just discussed, two executions of the list primitive on lists that differ by one element in the middle can have a linear (in the input size) number of function call trees that differ. For this reason, neither approach on its own performs well with such examples. In summary, except in computations that are carefully structured to yield similar function-call trees, many data modifications can cause both approaches to perform poorly.

In this article, we propose techniques for developing *self-adjusting* programs that can efficiently respond to incremental modifications to their data, via a combination of dynamic dependence graphs and memoization (Section 2 presents an overview). To enable efficient response to incremental modification, we observe and exploit an interesting duality between memoization and dynamic dependence graphs: through the combination of the two approaches, we can identify function calls to reuse using memoization, and we can identify function calls to re-execute using dynamic dependence graphs an re-execute them using change propagation.

We describe the self-adjusting-computation model of programming and present language primitives for translating purely functional programs into self-adjusting programs (Sections 3 and 4). These primitives enable the programmer to specify which computation data is *changeable* (i.e., time-varying), enabling the system to selectively track only dependences on changeable data. As examples, we describe how to implement self-adjusting versions of several list primitives in Section 5. Like a conventional program, a self-adjusting program can be executed with an input from-scratch, called *initial execution*, which yields an output and constructs the structures needed to update the output: a DDG and memo tables. After the initial run, the user, or more generally a *mutator* program, can change the input and update the output by performing change propagation, which, as before, re-executes the parts of the computation affected by the change, but reuses previous computations at the level of function-call invocations.

We present algorithms and data structures for supporting self-adjusting computations efficiently (Section 6) and provide an implementation of the proposed library based on these algorithms (Section 7). We evaluate the effectiveness of the approach by considering a number of benchmarks (Section 8). The benchmarks include various list primitives (filter, fold, map, reverse, split), two sorting algorithms (merge-sort, quick-sort), and some more involved computational-geometry algorithms, including several convex-hull algorithms (quick-hull [Barber et al. 1996], an output-sensitive convex hull algorithm [Chan 1996; Bhattacharya and Sen 1997]), and an algorithm for maintaining the diameter of a point set [Shamos 1978]. These benchmarks are chosen to span a number of computing paradigms, including simple iteration (filter, map, split), accumulator passing (reverse, quick-sort), incremental result construction (diameter), random sampling (fold), and divide-and-conquer (quick-sort, merge-sort, quick-hull).

We perform an experimental evaluation (Section 9) by comparing self-adjusting versions of our benchmarks to their standard, uninstrumented

versions. Experiments show that the performance of from-scratch runs of self-adjusting programs are within a constant factor (ranging between 2 and 30) of standard, uninstrumented programs. In general, we observe that the simpler the benchmark, the higher the overheads: for computational geometry benchmarks, the overheads are about a factor of 2, whereas for simple lists primitives they are higher. Since self-adjusting programs construct a graph representation of the execution, we expect the overheads to be moderate for simple programs like list primitives. We measure the effectiveness of change propagation by performing one insertion/deletion into/from the input and updating the output. The results demonstrate performance that greatly improves over rerunning the uninstrumented programs from scratch. Change propagation starts performing better than from-scratch execution (of the uninstrumented code) at relatively small input sizes (often less than 100). For the basic routines, on inputs of size 1,000,000, the time for change propagation is over four orders of magnitude faster than running the uninstrumented routine from scratch. For the computational-geometry benchmarks with inputs of size 100,000, the time for change propagation is more than three orders of magnitude faster than running the uninstrumented code from scratch. We also present an experimental study of the GC times as the size of the available memory changes with respect to the resident-memory size (Section 9.9).

This article combines two preliminary papers, one on an SML library for self-adjusting computation [Acar et al. 2006a] and the other on an efficient implementation of this library [Acar et al. 2006b]. Both of these papers are based on the first author's thesis [Acar 2005]. The results reported here differ slightly from those in the previous papers, because we used a different computer for these experiments and changed the measurements to exclude the time for initializations and input generation, which can constitute a reasonably large fraction of the computation in some cases. This measurement technique gives more accurate results by measuring the property of interest in isolation. The approach and the library described in this article have been applied to a number of applications, including simple and more sophisticated problems in invariant checking, motion simulation, high-dimensional computational geometry, and machine learning. Recent work extends the approach described here to support imperative memory updates [Acar et al. 2008a] and proposes direct language support and compilers for self-adjusting computation [Ley-Wild et al. 2008, 2009]. Section 10 contains more details on these developments and other related work.

2. OVERVIEW

We give an overview of dynamic dependence graphs [Acar et al. 2002] and memoization [Pugh and Teitelbaum 1989], describe their limitations and how they can be combined to make up for each others' limitations.

2.1 Dynamic Dependence Graphs

At a high level, *dynamic dependence graphs* or *DDGs* can be viewed as a representation of the data and control dependences. We can build the DDG of a

program as it executes by tracking the executed operations, and use it to update the computation and the output when the inputs are modified.

Structure of a DDG. The DDG of an execution or computation can be represented as a dynamic (function) call tree augmented with edges between computation data and function calls that depend on them. The nodes of the call tree represent function calls performed in an execution. The edges of the call tree represent caller-callee relationships between function calls. The call tree captures the control dependences at the granularity of a function call. A function call may have a control dependence on any of its ancestors in the call tree: if the ancestor takes a different control branch, then the descendant may cease to exist—the control path taken by a function call determines the descendants. Edges between computation data (e.g., inputs, intermediate results) and function calls represent data dependences: if a function call reads a piece of computation data, then there is an edge from the data to that call.

For a given program, we can build its DDG by tracking memory operations and function calls if the program conforms to the following requirements: (1) all memory locations are written at most once; (2) functions return no values; and (3) functions access nonlocal computation data before performing any function call. Apart from the write-once assumptions, these requirements cause no loss of generality (e.g., the approach applies to all purely functional programs).

In this overview, we will not describe how to enforce these restrictions, but assume that the computations conform to them.

Change Propagation. Given a purely functional program, we can run it with some input, construct its DDG as it executes, modify the input or data generated during the computation, and update the computation and the output by performing change propagation. Such a program is called a *self-adjusting* program. Wrapping around self-adjusting programs is an "outer-loop" program, called the "mutator," which is responsible for inspecting the output, modifying the input, and calling the change-propagation process. The mutator does not have to be purely function and is not subject to the write-once restriction.

At a high level, change propagation mimics a complete re-execution of the program with the modified data, but only re-executes the function calls that depend on the modification. To achieve this, change propagation maintains a work queue of function calls to be re-executed. The work queue initially contains the function calls that *directly* depend on the modified data. When re-executed, a function call can modify other computation data (by writing to memory), whose dependents are then added to the work queue. To ensure correctness, change propagation must re-execute function calls in the work queue in the same order as they were originally executed. This is important because there can be data and control dependences between a function call and other calls that come after it: a later function call can depend on some data updated by an earlier call; similarly, a later function call can have a control dependence on an earlier call. Since it is impossible to know *a priori* what control flow a function call will take when re-executed with the modified data, change propagation deletes all the descendants of the call from the DDG along with their dependences. This

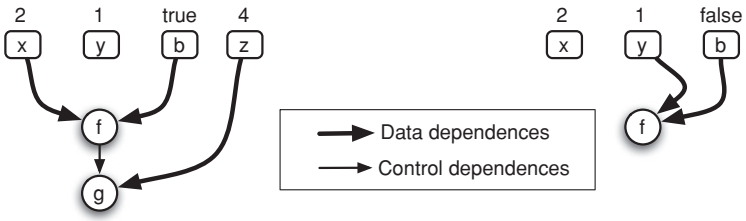


Fig. 1. The DDGs for the simple example.

conservative approach ensures correctness: the resulting DDG is isomorphic to the DDG that would be obtained by re-executing the program from scratch with the modified data.

A Simple Example. Consider a program consisting of the functions `f` and `g` (left unspecified) written as follows:

```
fun f (x,y,b) = if b then g(x*x) else y
fun g (z) = ...
```

The function `f` reads a flag `b` and calls the function `g` with x squared if the flag is true; otherwise, it returns `y`. If `f` is executed with `x=2,y=1,b=true`, then it calls `g(z)` with `z=4`. Thus, the DDG of this computation, shown in Figure 1 (left), consists of `f` and `g` with a call edge from `f` to `g` and edges from `x` and `b` to `f`, and from `z` to `g`. The control dependence between the call to `g` and the conditional inside `f` is represented by the parent-child relationship between `f` and `g`. Suppose that after this execution with `x=2,y=1,b=true`, we modify `y` to `0` and run change propagation. Since `y` is not read, it has no dependences, and thus change propagation terminates immediately, causing no modifications to the computation or output. Suppose now we modify `b=false`. Since `f` reads `b`, change propagation re-executes `f` after throwing away `g` (because `g` is a descendant of `f` in the call tree). When re-executed, `f` returns `y`. Since there are no more function calls that depend on `b`, change propagation completes. Figure 1 (right) shows the DDG for this example.

Effectiveness of DDGs and Change Propagation. In certain applications, DDGs and change propagation can support incremental updates in near optimal time (e.g., Acar et al. [2002, 2004]). In general, however, change propagation can take as much time as recomputing from scratch. Intuitively, this is because change propagation re-executes function calls from-scratch while throwing away all the work performed by their descendants—in other words, change propagation reuses computations at the granularity of function-call trees. In particular, if the modification being performed affects a function call that is shallow in the call tree (i.e., has low depth) that has many descendants, then change propagation will likely take a long time, because such a call is likely to perform significant work. If, however, the modifications affect only deep calls that have few descendants, then change propagation will be fast, because when

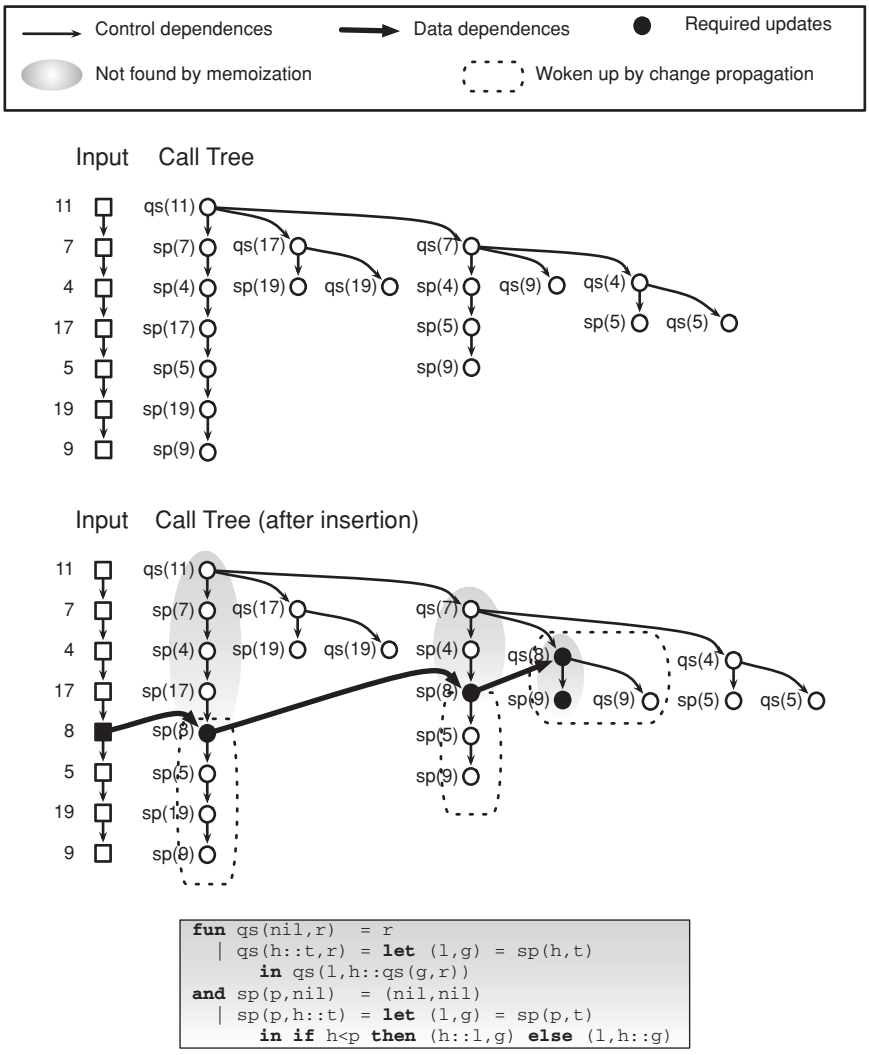


Fig. 2. Memoization versus change propagation for quick-sort (qs). The number in parenthesis is the first element of the argument list for the corresponding function call.

re-executed such function calls will likely perform little work. The performance of DDGs thus critically depends on the nature of the data modifications.

As an example, consider the quick-sort algorithm. Figure 2 shows the function-call trees of quick-sort with the inputs $I = [11, 7, 4, 17, 5, 19, 9]$ (top) and $I' = [11, 7, 4, 17, \mathbf{8}, 5, 19, 9]$ (bottom). Suppose that we start with the DDG of the computation with I, modify the input to I' by side-effecting memory and inserting the new key 8, and perform change propagation. Change propagation will re-execute the function calls that depend on the new key. In the bottom part of Figure 2, we show the function calls executed by change propagation in dotted boxes. By reasoning with call trees as illustrated, we can show that if an

element is inserted into the input list at the end (after the last element), then change propagation takes $O(\log n)$ time in expectation. If, however, the element is inserted in the middle, then change propagation will take $O(n)$ expected time. If the new element is inserted at the beginning (before the first element), then change propagation will require a from-scratch execution.

In summary, even though DDGs can be used to represent executions of any write-once program and to perform change propagation after modifications to data, their performance is highly sensitive to the nature of modifications. In this section, we illustrated this by the quick-sort example. It is not difficult to find other examples that have the same limitation. For example, with standard implementations of various list primitives such as `map` and `reduce`, change propagation requires (asymptotically) linear time for an insertion/deletion into/from the middle of the input list.

2.2 Memoization

Another approach to incremental computation is based on the classic idea of memoization. To see how memoization can help in efficiently updating computations, imagine executing a program with some input and later with a slightly different input. Since the inputs are similar, we may expect function calls from the first execution to be repeated in the second execution (with slightly different input). By memoizing function calls, we can reuse the results of the function calls from the first execution while performing the second execution.

Although it may seem that memoization alone can help reuse a significant portion of the previously computed results, its effectiveness also critically depends on the nature of the input modification. In some cases, it is possible to perform incremental updates optimally. In general, however, memoization can take as much time as recomputing from scratch. Intuitively, this is because with memoization, the ancestors of a function call that operates on modified data (including the call itself) cannot be reused. This is because the ancestors need to be re-executed to pass the modified data to the function call that uses it. Thus, if the modified data is used only by the function calls that are shallow in the call tree, then we may reuse their descendants via memoization. If, however, the modified data is used by deep calls in the call tree, then only few calls may be reused via memoization: we have to perform all the function calls from the root to affected calls.

As an example, consider, again, the executions of quick-sort shown in Figure 2. Even though the second input differs from the first by only one key, many function calls performed with the second input cannot be reused from the first execution. In particular, the calls whose input contain the new key (8) cannot be reused (Figure 2 highlights such calls). Thus, updating the output of quick-sort after one insertion can take asymptotically the same time as re-executing from scratch.

In general, there are many such examples which show that the performance of memoization critically depends on the nature of the input modification. For example, with the standard list primitives such as map and fold, incremental updates with memoization can require (asymptotically) linear time.

2.3 Combining DDGs and Memoization

In the previous two sections, we argued why neither change propagation nor memoization alone are effective in adjusting computations to modifications in the general case. Interestingly, they provide reuse of results in complementary ways: with memoization, we can reuse the parts of a computation that remain unaffected by the modification, while with change propagation, we find and re-execute the parts of the computation that are affected by the modification. This duality can be seen concretely in Figure 2: the function calls not reused by memoization and the function calls executed during change propagation intersect only at the modifications that are essential to update to correct the output. This duality suggests that we can hope to execute only the essential function calls (marked in black in the figure) by combining these two techniques.

In this article, we present a technique that combines change propagation and memoization. When executing a program, we still construct its DDG as usual and additionally we remember function calls and their DDGs in memo tables. Change propagation proceeds as in the DDG case. Given some modifications to computation data, we identify the function calls that depend on the modified data and insert them into a work queue. We then re-execute the function calls in the work queue in their original execution order. When re-executed, a function call can modify other computation data by writing into memory, causing other function calls to be inserted into the work queue. Propagation stops when all function calls that depend on modified data are re-executed. This process is so far identical to the change propagation process for DDGs.

The key difference is how we treat function calls during change propagation: if we perform a function call with the same arguments as before, then we reuse the DDG for that function call instead of executing the call from scratch. It is not possible, however, to reuse the DDG as is, since during change propagation, computation data may have been modified (because executed function calls may write to memory). Therefore, before a reuse, we update the DDG recovered from the memo table by performing a local change propagation through it. We then reuse the updated DDG by incorporating it into the computation. This mechanism of adapting a DDG to side-effected memory can be extended to enable the reuse of DDGs of a function call even when the arguments do not exactly match. We achieve this by allowing the programmer to specify which arguments to match and force the rest of the arguments to match via change propagation, that is, by updating the computation with respect to the unmatched arguments.

Realizing this high-level description efficiently requires addressing a number of challenges. One challenge is in efficiently storing DDGs of function calls in memo tables. Since a DDG is a graph, storing it can be expensive. We address this challenge by allowing reuse of only the subgraphs, or sub-DDGs, of the DDG currently being considered for change propagation. This allows us to use time intervals to represent the sub-DDGs as subgraphs of the current DDG. Another challenge is maintaining various invariants of DDGs while allowing their sub-DDGs to be reused. One such DDG invariant requires uniquely representing all executed data and control dependences. To ensure that reuse does not violate

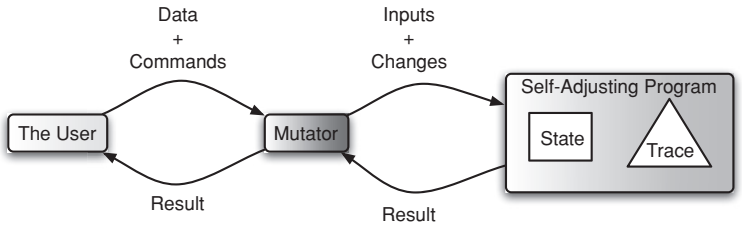


Fig. 3. The self-adjusting computation model.

this invariant, we permit the DDG of a function call to be reused *only* if it is disjoint from the DDG of the current computation. In general, checking that two DDGs are disjoint is expensive. We address this challenge by allowing the reuse of only the function calls that are descendants of the function call currently being re-executed by change propagation.

To illustrate how this approach takes advantage of the duality of change propagation and memoization, let us revisit the quick-sort example (Figure 2). Inserting the new key **8** into the input affects the first read containing the leftmost `sp` function (drawn as a dark solid circle). The change-propagation algorithm therefore re-executes that function. When re-executed, the function immediately calls `sp(5)`, which is part of the current DDG and is a descendant of the call being re-executed and thus can be reused. This process is repeated until all affected `read`'s are re-executed (the re-executed reads and all new function calls are shown as dark circles). The amount of work performed is expected $O(\log n)$, matching up to constants the work necessary to update the computation. Here n is the number of input elements, and the expectation is taken over internal randomization. The key difference between this approach and conventional memoization is the recording and reuse of computations (represented as DDGs) instead of just their results. Reusing DDGs is critical for correctness because it allows us to reuse computations in an imperative setting where computation data is updated via side-effects—conventional memoization requires purely functional programming.

3. THE MODEL

We present a brief overview of the self-adjusting computation model, describe the language primitives for writing self-adjusting programs, and sketch the underlying algorithms.

3.1 Self-Adjusting Program and the Mutator

In self-adjusting computation, programs are divided in two classes according to their roles: a self-adjusting program (core) and a top-level mutator.

A self-adjusting program is written much like a conventional program: it takes some input and produces an output; it may additionally perform some effects (i.e., writing to the screen). Like a conventional program, we can execute the self-adjusting program with some input. We call such an execution

a *from-scratch* or *initial* run. A self-adjusting program is typically executed from-scratch only once, hence the name initial run.

After an initial run, the input data[1] can be modified and the program can be asked to update its output by performing change propagation. This process of modifying data and propagating modifications can be repeated as many times as desired. Change propagation updates the computation by identifying the parts of the program that are affected by the modifications and re-executing them while reusing the parts that are unaffected by the modifications. Although change propagation re-executes only some parts of the program, it is semantically equivalent to a from-scratch run: it is guaranteed to yield the same result as running the program from scratch with the modified input. Asymptotically, 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 iterations of input data changes and change-propagation steps. For this reason, change propagation must be highly efficient even at the expense of slowing down the initial run.

The interaction between a self-adjusting program's initial output and its subsequent inputs may be complex. We therefore embed a self-adjusting core program in a meta-level *mutator* program to drive this feedback loop. One kind of a mutator that is common in various applications is a program that interacts with a user to obtain the initial input for the self-adjusting core, runs the core with that input, and continues interacting with the user, modifying the data as directed and performing change propagation as necessary. We use such a mutator in our experiments in this article. Another kind of mutator that is common in motion simulation is a program that combines user-interaction with event scheduling (e.g., [Acar et al. 2008b]). 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. This mutator perform the 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 modify computation data in arbitrarily complex ways as required by the application.

3.2 The Primitives

We provide two kinds of primitives for writing self-adjusting programs and the mutator: (1) the core primitives and (2) the impure, meta primitives. A self-adjusting program employs the core primitives. A mutator employs the meta primitives.

The interface between a self-adjusting program and the mutator is established by *changeable data* (i.e., data that is modified over time). To mark data as changeable, we provide the notion of *modifiable references* or *modifiables* for short: a modifiable is a memory location with a fixed address and modifiable contents.

[1]In fact, not only the input data but also the data generated during the initial run can be modified.

Core Primitives. Self-adjusting programs operate on changeable data through a set of core primitives: the `mod` operation creates a modifiable, and the `read` and `write` operations provide access to the modifiables. The `read` operation takes a modifiable and a function, called the *reader*, and applies the reader to the contents of the modifiable. A `read` operation cannot return a value but it can write to other modifiables.

To support memoization, we provide a family of *lift* operations. The programmer can memoize any expression (block of code) by specifying each free variable of the expression as *strict* or *nonstrict* and using the appropriate lift operation. During memo matching, the change-propagation algorithm looks for existing computation that matches the strict variables. In this way, the lift operations generalize conventional memoization by allowing computation reuse by only matching some of the free variables (i.e., the strict part). As we will explain in detail later in this section, the nonstrict variables are considered if the lookup succeeds: the retrieved computation is then "adapted" to take into account these variables.

The core primitive may be arranged in a type-safe interface that ensures that modifiable references are used in a way that is consistent with purely functional programming. Section 4 describes such an interface in Standard ML.

Meta Primitives. To enable the mutator to modify the data to a self-adjusting computation, inspect its output and run change propagation, we provide a set of meta-primitives. These primitives create modifiables (`new`), dereference them (`deref`), and write into them (`update`). The `new` operation returns an empty modifiable, the `deref` operation returns the contents of the modifiable, and the `update` operation updates the value stored inside a modifiable with a new value. Note that the meta-primitives treat modifiables imperatively. In fact modifiables and conventional references are isomorphic as far as the meta-primitives are concerned. In addition, we include a `propagate` operation to enable the mutator to perform change propagation.

Alternative Interfaces. It is possible to design a simpler interface for self-adjusting computation by treating all computation data as changeable. It is, however, difficult to provide an implementation for such an interface that performs well in practice, since treating all data as changeable requires tracking all memory operations and all dependences between data and computation. In reality, this would be prohibitively expensive. Such a model can, however, be interesting from a theoretical perspective, as it provides a more direct way to analyze the asymptotic complexity of self-adjusting programs; this subject is discussed in more details in the first authors thesis [Acar 2005].

3.3 Traces and Change Propagation

In self-adjusting computation, we represent computations as execution traces, which record the data dependences between the computations and the control dependences in the executed code. These traces are constructed during the initial run and updated by the change-propagation algorithm.

One key idea behind our approach is selective dependence tracking: we only track data and control dependences that pertain to modifiable references. This suffices because the interface to modifiables ensures that all changeable data is placed in modifiables, and all computations that operate on changeable data take place inside the read operations. To see this, first note that the mutator can perform modifications only by side-effecting the contents of modifiables. Thus, all other changeable data must be computed using modifiables. Since the only way to access the contents of a modifiable is to read it, other changeable data is computed by the read operations. Furthermore, since the read operations cannot return a value, the only way they can affect other data is by writing into other modifiables. Thus, all changeable data, except for those operated inside the read operations, is written into modifiables. This property allows us to track all data dependences between changeable values by tracking modifiables and the read operations. Similarly, we track all control dependences between computations that operate on changeable data by tracking the dynamic containment relationships between read operations. We say that a read r is *contained* within the dynamic scope of another read r' if the execution of r takes place during the execution of r' (the execution-time frame of r' is within that of r).

More concretely, we represent a trace as a tuple consisting of a DDG and a memo table. A DDG consists of a set of modifiables V, a set of reads R, a set of data dependences $D \subseteq V \times R$, and a containment hierarchy $C \subseteq R \times R$. The data dependences represent the dependences between modifiables and reads: $(v, r) \in D$ if r reads v. The containment hierarchy C represents the control dependences: $(r_1, r_2) \in C$ if r_2 is contained within the dynamic scope of r_1. We note that a containment hierarchy naturally corresponds to a function call tree, mentioned in earlier sections. A memo table Σ maps tuples consisting of the function and the strict argument to tuples consisting of a modifiable (allocated for the nonstrict argument) and a DDG (of the executed function call).

We construct DDGs by tracking the `mod`, `read`, and `write` operations. A `mod` operation inserts a new modifiable into `V`, a `read` operation inserts a new read operation into `R`, and a `write` operation sets the value of the given modifiable.

The `lift` operations populate the memo table and reuse existing DDGs. Consider executing a lift operation with a function `f` and strict and nonstrict variables `s` and `n`, respectively (multiple strict and nonstrict variables are handled similarly). We first perform a memo lookup, seeking for a call to `f` with the strict argument `s`. If the lookup fails, then we allocate a modifiable `m` for the nonstrict argument `n` and write `n` into `m`. We then execute the function call and store its tuple, consisting of `m` and the DDG in the memo table indexed by the function and the strict arguments. If the lookup succeeds, then we check if the set of reads of the DDG found and the DDG of the current computation remain disjoint. If so, then we perform change propagation on the found DDG and insert it into that of the current computation. Otherwise, we execute the function as if the lookup had failed. When reusing a DDG, we need to make sure that its read set is disjoint from the current DDG because we require that each executed `read` is represented uniquely in the DDG of the computation.

When a self-adjusting program is executed with an input, we construct its DDGs by tracking the operations on modifiables, as described before. After the initial run is performed, the mutator can modify the contents of modifiables by using the `update` operation and perform change propagation by calling `propagate`. The change-propagation algorithm maintains a queue of affected reads that initially contains the reads of the updated modifiables (a read is represented as a closure consisting of a reader and an environment). The algorithm repeatedly removes the earliest read r from the queue, that is, the read that was executed earliest, and re-executes its reader. Before re-executing the reader, the algorithm removes all the reads contained in r (found using the containment hierarchy). Note that even though they may be removed from the DDG of the current computation, the removed reads may still remain live because they may be stored in the memo tables. This makes it possible to later reuse the parts of the previous execution of the read. When the change propagation algorithm terminates, the result and the DDG of the computation are identical to the result and the DDG obtained from a from-scratch execution.

4. THE SML LIBRARY

We present an SML library that organizes the primitives in a type-safe interface. The interface (Section 4.1) enforces proper usage of modifiables, but still requires additional correct usage rules to ensure the correctness of change propagation. The library facilitates transforming conventional, purely functional SML programs into self-adjusting programs (Section 4.2). Based on this interface, recent work proposes direct language support and compilers for self-adjusting computation [Ley-Wild et al. 2008, 2009] that can statically ensure correctness.

4.1 The Interface

Figure 4 shows the interface for our library. The signature `COMBINATORS` specifies the core library functions. These functions rely on a `BOXED_VALUE` module that supplies functions for operating on boxed values.

A *boxed value* is equipped with a constant-time *index function* that returns a unique integer, called an *index*, and a constant-time equality function. We use boxed values to determine when to stop change propagation (using equality test) and when a computation may be reused (using the indices as a lookup key). Boxed values are implemented as a pair consisting of a value and a unique integer index. The function `new` creates a boxed value by creating a unique index for that value. The `eq` function compares two boxed values by comparing their indices. The `valueOf` and `indexOf` functions return the value and the index of a boxed value, respectively. The `BOXED_VALUE` module may be extended with functions for creating boxed values for a type. Such type-specific functions must be consistent with the equality of the underlying types. For example, the function `fromInt` may assign the index i to the integer i.

The `COMBINATORS` module defines the primitives for modifiable references and for memoization. It consists of two parts: core operations on modifiables for writing self-adjusting programs and meta operations (impure operations)

```
signature BOXED_VALUE = sig
  type index
  type α t

  val init:  unit → unit
  val new:  α → α t
  val eq:  α t * α t → bool
  val valueOf:  α t → α
  val indexOf:  α t → index
  val fromInt:  int → int t
end
structure Box:  BOXED_VALUE = struct ...  end

signature COMBINATORS = sig
  eqtype α modref
  type α cc

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

  val mkLift:  (α * α → bool) →
               (Box.index list * α) → (α modref → β) → β
  val mkLiftCC: ((α * α → bool) * (β * β → bool)) →
                (Box.index list * α) → (α modref → β cc) → β cc

  (** Meta Operations **)
  val init:  unit → unit
  val new:  unit → α modref
  val update:  (α * α → bool) → α modref → α → unit
  val deref:  α modref → α
  val propagate:  unit → unit
end
structure C: COMBINATORS = struct ...  end
```

Fig. 4. Signature for boxed values and combinators.

for inspecting modifiables, changing the contents of modifiables via side effects, and for performing change propagation. A mutator uses the meta-operations to control self-adjusting programs.

The Core Interface. The core primitives ensure that well-typed self-adjusting programs have the following properties:

—all modifiables are written exactly once, and
—no modifiable is read before it is written.

These properties can be ensured statically. To do so, we distinguish between changeable and stable computations. A *changeable computation* of type α cc is a function that ends by writing its result of type α into a *destination*, a modifiable

reference, of type α `modref`. The destination of a changeable computation is implicit.[2]

A `write` primitive introduces a changeable computation: it creates a trivial changeable computation which writes the given value. The `write` primitive takes a comparison function to detect when the value written is equal to the value already stored in the destination modifiable—this avoids unnecessary propagation of changes. Any nontrivial changeable computation reads some modifiables and performs calculations on the values read.

The `read` primitive, which we render as $\ominus\!\!\rightarrow$ in infix notation, takes an existing modifiable reference together with a reader—a changeable computation—for the value read. It returns a changeable computation that encompasses the process of reading from the modifiable and performing the computation specified by the reader with the value read.

The elimination form for changeable computations is the `modref` function; `modref` allocates a fresh modifiable, executes the given changeable computation with that modifiable as the destination (the result is written to the allocated modifiable) and returns the modifiable. A modifiable that holds values of type α has type α `modref`. Thus, every execution of a changeable computation of type α `cc` starts with the creation of a fresh modifiable of type α `modref` and ends by writing to that modifiable. For the duration of the execution, the reference never becomes explicit. Instead, it is carried implicitly in a way that is strongly reminiscent of monadic computation. This approach to enforcing the invariants on modifiable references is similar to that proposed by Carlsson in the context of the Haskell language [Carlsson 2002].

To support computation memoization, we define a notion of *lifting* with *strict* and *nonstrict* arguments. Given a function (e.g., `fn x => e`), a *strict* argument is a free variable of the function body (e.g., `e`) that is required to match for computation reuse, whereas a *nonstrict* argument is a free variable that is not required to match for computation reuse. Informally, we use the strict arguments as the lookup key for retrieving the appropriate computation and adapt the computation to match the remaining (i.e., nonstrict) arguments.

We distinguish between several kinds of lift operations. The simplest lift operation memoizes nonchangeable computations; it takes the indices of the strict arguments as a list of type `Box.index list`, a nonstrict variable of type α and a function of type α `modref` $\rightarrow \beta$ and returns a value of type β. There is also a similar lift operation for changeable computation, where the lift operation takes the indices of the strict arguments as a list of type `Box.index list`, a nonstrict variable of type α and a function of type α `modref` $\rightarrow \beta$ `cc` and returns a value of type β `cc`. For the purposes of memo matching, strict arguments must be boxed values. The `mkLift` and `mkLiftCC` functions create lift operations from supplied equality tests for the nonstrict argument (of type α), and in the case of `mkLiftCC` equality tests for the result of the changeable computation (of type β `cc`). These functions allow one nonstrict argument. Not shown here, our library also contains `mkLift2`, `mkLiftCC2`, `mkLift3`, `mkLiftCC3`, and so on to support more than one nonstrict argument.

[2]In the implementation a changeable computation takes the destination to write as an argument.

A lift operation performs memo lookups based only on strict arguments by using the index list for strict variables as a key—it ignores the nonstrict arguments. If there is no memo match, then it places the nonstrict arguments into modifiables and then applies the function to these modifiables to compute the returned result. If the strict arguments match, then a memo lookup succeeds and returns a computation. By construction, this computation wraps the nonstrict variables inside modifiables. The operation modifies the contents of these modifiables by storing the values of nonstrict arguments inside them and by performing change propagation on the matched computation. The memo table, indexed by just the strict arguments, remembers the modifiables that hold the nonstrict arguments as well as the memoized computation.

Meta-Operations. The meta-(impure) operations consist of the `init` function for initializing the system, and `propagate` function for performing change propagation, the `new` function for creating (empty) modifiables, the `update` function for destructively updating modifiables, and the `deref` function for dereferencing modifiables. The `propagate` function runs the change-propagation algorithm. Note that while these operations can update or access modifiables, they will not be tracked in the DDG because they take place at the meta-operation level.

Correct Usage. We require that a self-adjusting program written with the core primitives satisfy a number of *correct-usage* rules to ensure correctness, that is, that change propagation updates the computation and its output correctly after a modification.

Correct usage requires that the side-effecting meta-operations, that is, all except for `deref` are not used by the self-adjusting program—meta-operations can be used only by the mutator.

The other correct-usage rules concern the lift operations. We require that the programmer declare all free variables of the function being lifted either as a strict or a nonstrict argument. Whether an argument is strict or nonstrict does not impact correctness; what matters is that *all* arguments are declared. Similarly, we require that no two functions share the same lift operation. Each function needs its own lift operation, which can be generated by an appropriate `mkLift` function.

Violation of correct usage can prevent change propagation from correctly updating a computation, as it may generate incorrect memo matches. For example, omitting a free variable from a lifted function can generate an incorrect memo match. Similarly, lifting multiple functions with the same lift operation can cause the one function call to match another, ultimately causing computations to be reused incorrectly.

Although it is possible to eliminate the correct usage rules by designing a richer type system, it seems difficult to embed such a type system within the type system of SML [Acar et al. 2006a]. It is possible, however, to ensure correctness statically by providing direct languages support [Ley-Wild et al. 2008, 2009].

4.2 Transformation

Using our SML library, the programmer can transform a normal, purely functional SML program into a self-adjusting program by annotating the code with the core primitives supplied by the library. This transformation requires only syntactic changes to the code. It consists of two steps.

First, the programmer determines what parts of the input data will be modified over time and places such changeable data into modifiables. This makes the program sensitive to updates to places in the input marked by the modifiables. The programmer then annotates the original program by making the reads of modifiables explicit. Since the reads are changeable computation, this requires creating new modifiables that hold all the changeable data in the computation.

Second, the programmer memoizes selected functions by creating lift operations and applying them to the functions chosen to be memoized. In general, all functions can be memoized, but performance concerns imply that functions that perform nonconstant work should be memoized. To apply a lift operation to a function, the programmer must partition the arguments into strict and nonstrict arguments. The particular partitioning chosen does not affect correctness, but can effect performance of change propagation because only strict arguments are matched during a memo lookup. Therefore, making many arguments strict can decrease reuse via memoization; making many arguments nonstrict would increase memo matches but can also increase change-propagation cost to adapt the computation for reuse.

As guidance for determining the partition, we suggest the following *strictness principle*: make nonstrict those free variables that are only passed as arguments to other memoized functions, make strict all other variables. This ensures that when a memo match takes place, the results computed by the invocation of the memoized function are reused without any need for change propagation and the results computed by other memoized calls will be updated via change propagation. Since a function call invocation takes at least constant time, the principle ensures that the cost of memoization (a constant) can be amortized against the cost of performing the invocation itself.

All the applications that we use in this article are written using this two-step process. We describe several examples of how to write self-adjusting programs in the next two sections. In Section 8, we discuss several applications and show that the transformation process requires reasonably small changes to the existing code.

5. AN EXAMPLE: MODIFIABLE LISTS

As an example of how we can write self-adjusting programs using the presented library, we consider *modifiable lists*, a data structure for lists whose contents can change over time. We describe a representation for modifiable lists and discuss some list operations analogous to ordinary functional lists.

5.1 The Interface

Figure 5 shows the interface (`signature`) and the implementation (`structure`) for a module implementing modifiable lists. We define a modifiable list as a

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

  val eq:  α Box.t modcell * α Box.t modcell → bool
  val write:  α Box.t modcell → α Box.t modcell C.cc
  val lengthLessThan :  int → (α t) → bool C.modref
  val map:  (α Box.t → β Box.t) → α Box.t t → β Box.t t
  val fold:(α Box.t*α Box.t→α Box.t)→(α Box.t t→α Box.t C.modref)
end

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

  infix ⊙→

  val op ⊙→ = C.read

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

  fun write c = C.write eq c
  fun lengthLessThan n l = ...
  fun map f l = ...      (* See Figure 6 *)
  fun reduce binOp l = ... (* See Figure 7 *)
end
```

Fig. 5. The signature for modifiable lists and an implementation.

modifiable reference to a cell, which is either a cons cell or a nil. A cons cell consists of an element and a tail which is a modifiable list. Modifiable lists are defined similarly to conventional lists; the difference is that the tail of a cell is a modifiable itself.

By placing the tail of a cons cell into a modifiable, we enable the mutator to modify the contents of the list by appropriately updating the tail modifiables. For example, the mutator can splice in new elements into a list and splice out existing elements from the list. Thus modifiable lists may be used to represent a set of elements that change over time. Additionally, a mutator can read a list of items from the user (e.g., list of objects to process, list of keystrokes to match against some string) and represent the list as a modifiable list. It can then perform some self-adjusting computation with this list (e.g., compute some property of the objects, perform a string-match on the keystrokes) and return its output to the user. As it continues to interact with the user, the mutator can modify the list as directed by the user (using the meta-operations), and update the output by performing change propagation.

Our modifiable-list library provides the functions `eq`, `write`, `lengthLessThan`, `map`, and `fold`. These functions all require the elements in the lists to be boxed (Section 5.1). Boxing the elements facilitates testing for equality and reusing

```
map:(α Box.t → β Box.t) →
    (α Box.t list) →
    (β Box.t list)
fun map f l =
let

  fun mapM c =
    case c of
      nil ⇒ nil
    | cons(h,t) ⇒

      cons(f h, mapM t)
in mapM l end
```

```
map:(α Box.t → β Box.t) →
    (α Box.t ML.t) →
    (β Box.t list)
fun map f l =
let
  val lift = C.mkLiftCC (ML.eq,ML.eq)
  fun mapM l = C.modref(l ⊙→ (fn c =>
    case c of
      ML.NIL ⇒ ML.write ML.NIL
    | ML.CONS(h,t) ⇒ t ⊙→ (fn ct ⇒
        lift ([Box.indexOf h],ct) (fn t ⇒
            ML.write (ML.CONS(f h,mapM t))))))
in mapM l end
```

Fig. 6. The self-adjusting map function.

computations via memoization. The eq function tests for shallow-equality of the two lists. The `write` function specializes the `write` function of the `COMBINATOR` module for modifiable lists. The `lengthLessThan` function tests if the list is shorter than the given value. This function is straightforward to implement; we therefore omit the code. As example list operations, we discuss the `map` and the `fold` functions. The modifiable-list versions of other conventional list operations can be obtained similarly.

When translating conventional programs into self-adjusting programs, we apply the two-step process described in Section 4.2. Many self-adjusting programs obtained via this translation perform well under change propagation. As an example of such a program, we consider the `map` example. Not all self-adjusting programs obtained via such a simple translation, however, perform well with change propagation. To be efficient under change propagation, the translated programs must be *stable*, that is, their execution should be relatively insensitive to changes to their data. By performing a stability or a sensitivity analysis, we can determine how stable or sensitive a program is for some kind of change. We describe this analysis technique informally (Section 5.3). As an example program whose typical implementation is not stable, we consider the `fold` function and describe a stable implementation for it (Section 5.4).

5.2 The `map` Function

Figure 6 shows the code for a standard implementation of `map` (left) and its self-adjusting version (right). Both functions take a function `f` and a list `l` and return a list obtained by applying `f` to each element of `l`.

We obtain the self-adjusting version of `map` from the conventional version by applying the two-step transformation process (Section 4.2). We start by changing the input list to a modifiable list. In a modifiable list, we need to `read` the modifiable to access the contents. Since a `read` operation is a changeable computation, it can only take place in the context of a `modref`, which is also used to store the result.

Then, we memoize `map`. We do not need to memoize the `NIL` branch, as it performs trivial work. We memoize the `CONS` branch by treating `h` as strict and

the contents of `t` (bound to `ct`) as nonstrict by applying the strictness principle. Finally, we create the `lift` function for the `CONS` branch by using the `mkLiftCC` function by supplying the list equality function. This completes the translation.

5.3 Stability

By applying the translation process, we can derive self-adjusting versions of other conventional list functions such as `reverse`, `filter`, `fold`. How effective are these self-adjusting functions under small changes to their input, for example, an insertion or deletion into or from the input list? As with conventional complexity analysis of run-time, reasoning about the run-time of change propagation under some change often requires rigorous analysis. Developing an intuitive bound, however, is simpler. The idea is to compare the operations performed in from-scratch executions of the program with the inputs before and after a change. If the difference in the set of executed operations is large, then we say that the program is not stable. If the difference is small, then we say that the program is stable.[3]

To analyze the stability of `map`, consider executing it with two input lists that differ by one key. The performed operations consist of the traversal of the lists and the application of the supplied function to the elements. Since the two lists differ by one key, the difference will be constant. Thus, `map` will take constant time to respond to an insertion/deletion into/from the input.

Many programs are naturally stable or can be made stable with relatively small changes. All the examples we consider in this article, except for one, are either naturally stable or made stable with small modifications. Some programs, however, are not stable. Consider, for example, the list primitive `fold`, which takes a list and an associative binary operator, and applies the operator to the elements of the list to produce a single value (e.g., applying `fold` to the list `[1,2,3,4]` with the integer plus operation yields `10`).

The fold function is typically implemented by traversing the list from left to right while maintaining the partial results for the visited prefix in an accumulator. This implementation of `fold`, however, is unstable. To see this, let's consider the operations performed with inputs that differ by one key. More precisely, consider the case when one list has one element at the beginning that the other lacks, for example, `[1,2,3,4,...]` and `[2,3,4,...]`. Suppose now we sum the elements in the list by applying the addition operator to the elements from left to right, keeping the prefix sum in an accumulator and summing each element with the accumulator. The prefix sums with the lists are `1,3,6,10,...` and `2,5,9,...`. The prefixes differ by 1 in every position. Thus, no two prefix sums are the same, so a linear number of operations (in the length of the input) will need to be updated. Consequently, change propagation will take at least linear time. In the section that follows, we describe how we can implement a stable `fold` for lists.

[3]A precise treatment of stability is out of the scope of this article but can be found in the first author's thesis [Acar 2005].

```
fun fold binOp l =
let
  fun halfList l =
  let
    val hash = Hash.new ()
    fun pairEqual ((b1,l1),(b2,l2)) = Box.eq(b1,b2) andalso (l1=l2)
    val writePair = C.write' pairEqual
    val lift = C.mkLiftCC (ML.eq,ML.eq)

    fun half l =
    let
      fun sumRun(v,l) = l ⊙→ (fn c =>
        case c of
          ML.NIL => writePair (v,l)
        | ML.CONS(h,t) =>
            if hash(Box.indexOf h) = 0 then
              writePair (binOp(v,h),t)
            else sumRun(binOp(v,h),t))

    in l ⊙→ (fn c =>
      case c of
        ML.NIL ⇒ ML.write ML.NIL
      | ML.CONS(h,t) ⇒ t ⊙→ (fn ct ⇒
        lift ([Box.indexOf h],ct) (fn t ⇒
          let val p = C.modref (sumRun (h,t))
          in p ⊙→ (fn (v,t') ⇒ ML.write (ML.CONS(v, C.modref (half t'))))
          end)))
    end
  in C.modref (half l) end

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

Fig. 7. Self-adjusting list `fold`.

5.4 A Stable `fold` Function

Figure 7 shows a stable version of `fold`. This implementation uses the classic technique of random-sampling to compute the result. The idea is to shrink the input list into smaller and smaller lists until only one element remains. To shrink the list, we choose a randomly selected subset of the list and combine the chosen elements to their closest element to the right. Note that a deterministic approach, where, for example, every other element is deleted is not stable, since 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—only associativity suffices. For randomization, we use a random hash function [Wegman and Carter 1979] that returns 0 or 1 with probability 1/2.

In the implementation in Figure 7, the `fold` function takes two arguments: an associative binary operator `binOp` and a modifiable list `l`. When `fold` is called, it runs the helper function `halfList`, which shrinks the list by calling the

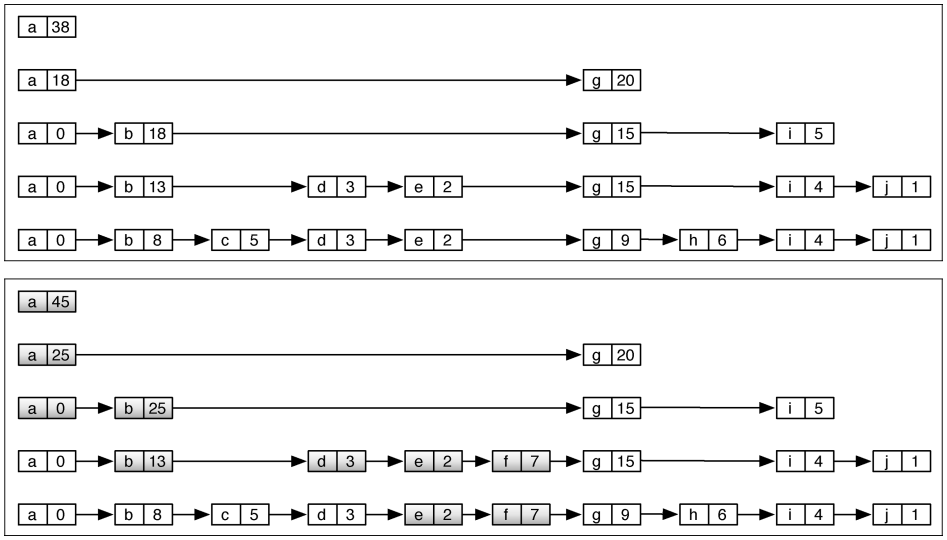


Fig. 8. The lists at each round before and after inserting the key f with value 7.

`sumRun` function and then recursively calls itself. The recursion terminates when only one element remains in the list. As the name suggests, each application of `halfList` shrinks the list approximately by half; this is done by applying the `sumRun` function multiple times. The behavior of the `sumRun` function is of particular note: as it recursively traverses the list, the function "flips" a coin for each element visited until a tails is found. During the traversal, the function stores in the argument `v` the running sum of the values it has seen so far. When a tails is found, the function returns a changeable computation that writes to the designated modifiable the combined value and the remaining list. In this case, the target modifiable is the variable `p`, to which `C.modref` that calls the `sumRun` function is bound: `val p = C.modref (sumRun (h,t))`.

For stability, we memoize the `half` function by applying the same reasoning as in the `map` function: The `NIL` branch is not memoized because it performs trivial work. We apply the strictness principle when memoizing the `CONS` branch, treating `h` as strict and the contents of `t` (bound to `ct`) as nonstrict.

The algorithm requires expected linear time because each application of `half` reduces the size of the input by a factor of two (in expectation). Thus, we perform a logarithmic number of calls to `halve` with inputs whose size decreases exponentially. How stable is the approach? As an example, Figure 8 shows an execution of `fold` that computes the sum of the integers in the lists [(a,0), (b,8), (c,5), (d,3), (e,2), (g,9), (h,6), (i,4), (j,1)] and [(a,0), (b,8), (c,5), (d,3), (e,2), **(f,7)**, (g,9), (h,6), (i,4), (j,1)]. The two lists differ in one position (the box indexed by f with payload 7). Comparing two executions, only the elements in the highlighted cells differ. It is not difficult to show that we have a constant number of such cells in each level and, based on this, prove that the approach is $O(\log n)$ stable in expectation. Thus, after inserting/deleting the box `f`, we can update the output in logarithmic time.

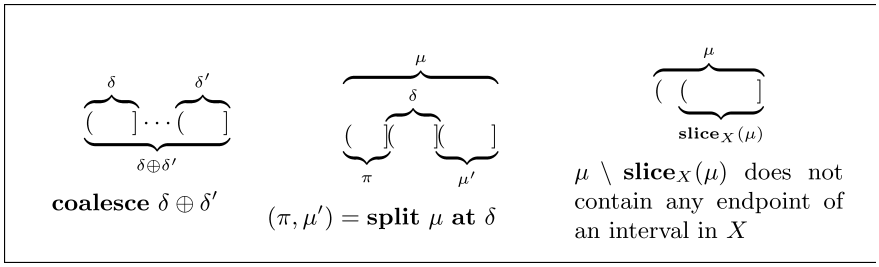


Fig. 9. An illustration of the operations on time stamps.

6. ALGORITHMS FOR SELF-ADJUSTING COMPUTATION

In this section, we describe efficient algorithms for implementing the self-adjusting computation model. These algorithms form the basis for our implementation described in Section 7. In our description, we assume the primitives as described in Section 3.2. We start with the crucial notion of time stamps and time intervals that we use to efficiently enforce various invariants.

6.1 Time Stamps and Intervals

We use time stamps and time intervals to efficiently support self-adjusting computation primitives.

We define a *time (stamp)* to be an element of a totally ordered, dense universe $(U, <)$ that can be viewed as a *(virtual) time line*. We denote a time stamp by t (and variants) and assume all time stamps are drawn from U.

We define an interval δ to be either empty, written $\emptyset$, or half-open, which is represented by two time stamps. We say that the interval δ is half-open if its *stop time*, written $t_e(\delta)$, is included in δ while its *start time*, written $t_s(\delta)$, is not. For $t_1, t_2 \in U$ and $t_1 \leq t_2$, the half-open interval $(t_1, t_2]$ is the set $\{t \in U : t_1 < t \leq t_2\}$. We use lower-case Greek letters μ, δ, π, and variants to denote intervals. It may be helpful to think of U as the real line and the intervals as the standard half-open real intervals.

We define the following relations between intervals.

—*Containment*. We say that an interval δ' is *contained* in another interval δ, written $\delta' \subseteq \delta$, if $t_s(\delta') \leq t_s(\delta) \leq t_e(\delta) \leq t_e(\delta')$.

—*Tail*. We say that a nonempty interval δ' is a *tail* of δ, denoted by $\delta' \sqsubseteq \delta$, if $t_s(\delta) \leq t_s(\delta') < t_e(\delta') = t_e(\delta)$. That is, δ' is a tail of δ if δ' is completely contained in δ and shares the end time with δ. If $\delta' \sqsubseteq \delta$ and $\delta' \neq \delta$, then δ' is a *proper tail* of δ, denoted $\delta' \sqsubset \delta$.

—*Slice*. For a given set X of nonempty intervals, we say that the interval μ' is an X*-slice* of μ if μ' is a proper tail of μ, and $\mu \setminus \mu'$ does not contain any start or stop time stamps of the intervals in X.

We define operations for coalescing and splitting intervals and for taking a slice of an interval. Figure 9 illustrates these operations.

—*Coalesce*. This operation, denoted by $\delta \oplus \delta'$ where δ and δ' are arbitrary intervals, returns the smallest interval that contains both intervals, that is, the interval $(\min\{t_s(\delta), t_s(\delta')\}, \max\{t_e(\delta), t_e(\delta')\}]$.

—*Split* μ *at* δ. If $\mu \neq \emptyset$, $\delta \subseteq \mu$, and $\delta \not\sqsubseteq \mu$, splitting μ and δ yields a pair of intervals (π, μ') such that π, δ, and μ' are mutually disjoint and $\pi \cup \delta \cup \mu' = \mu$. Moreover, we have $\mu' \sqsubseteq \mu$ and $(\delta \cup \mu') \sqsubseteq \mu$. If $\delta = \emptyset$, then $\pi = \emptyset$ and $\mu' = \mu$.

—$Slice_X(\mu)$ returns an X-slice of μ. The definition does not uniquely describe the result, but any X-slice will suffice for our purposes.

6.2 The Algorithms

We present precise algorithms that realize the high-level description previously presented in Section 3.3.

To realize the high-level description efficiently, we need to

(1) maintain the execution order of all reads in the DDG;
(2) represent the containment hierarchy such that all reads contained in a given read can be found efficiently;
(3) store DDGs in memo tables efficiently; and
(4) test whether a DDG is available for reuse (i.e., if the read sets of that DDG is disjoint from the current DDG).

To maintain the execution order of all reads and to represent the containment hierarchy, we use time stamps and intervals. The idea is to maintain, during an execution, a virtual execution time line and assign a *time interval* to each read, which is the time interval in which the read executes; the time interval of a read is represented with two time stamps, one for the start and one for the end. Using these intervals, we realize goals (1) and (2) as follows. We order reads by their start times. We can check for containment by performing interval containment tests—a read is contained within another if its time interval is contained within the other. Thus, the ordering and containment tests are simple checks on the time intervals.

For performance, we store in our memo tables only the DDGs that belong to the current computation. With this restriction, we can represent a sub-DDG of the current DDG with a time interval—the interval in which that DDG was created. This restriction also helps us avoid testing for disjointness. The idea is to restrict the `lift` operation to reuse DDGs only during change propagation—no reuse takes place in a from-scratch run. To support reuse during change propagation, we maintain an interval that we call the *current interval* and allow only DDGs that are contained within this interval to be reused. Before re-executing a read, change propagation sets the current interval to be the interval of that read. As reuse takes place, this interval is adjusted by deleting the part of the interval up to the end of the reused interval. This ensures that the current interval remains a single, connected interval, and the reused intervals are nonoverlapping and can be given a total order.

We represent a DDG by the triplet (V, R, D), where V is the set of modifiables, R is the set of reads, and $D \subseteq V \times R$ is the set of data dependences. We

associate each read r with its interval, denoted $\Delta(r)$, with the *source* modifiable, denoted $\mathrm{Src}(r)$, and the reader function, denoted $\mathrm{Rd}(r)$. We define Δ on a set of `reads` point-wise as $\Delta(R) = \{\Delta(r) \mid r \in R\}$. We say that a read r_1 is *contained* in another read r_2 if and only if the time interval of r_1 lies within the time interval of r_2.

We represent a memo table as a mapping from function names and strict arguments to a triple consisting of the interval of the call, the modifiable for the nonstrict variable, and the result of the call.

Our algorithms are "interval-passing"—we evaluate all expressions except for meta-operations in the context of an interval, called the *current interval*, denoted by μ (and its variants). Executing an expression e in the context of μ, written $e\ \mu$, returns an updated current interval $\mu' \sqsubseteq \mu$, a fresh interval δ, and a result a, that is,

$$(\delta, \mu', a) \leftarrow e\ \mu, \text{ where}$$
$$\mu' \sqsubseteq \mu \text{ and } \delta \cap \mu' = \emptyset \text{ and } \delta \cup \mu' \text{ is a } \Delta(R)\text{-slice of } \mu.$$

Intuitively, the reader may find it helpful to think of evaluating an expression as creating a fresh interval for its execution and consuming some of the current interval (by returning a tail).

The expressions that correspond to `mod`, `read`, `write`, `lift` primitives treat intervals specially. All other expressions coalesce intervals while propagating the current interval forward, for example, e_1 and e_2 are sequentially executed as follows

$$(\delta_1 \oplus \delta_2, \mu'', a) \leftarrow (e_1; e_2)\ \mu, \text{ where,}$$
$$(\delta_1, \mu', \_) \leftarrow e_1\ \mu \text{ and } (\delta_2, \mu'', a) = e_2\ \mu'.$$

Figure 10 shows the pseudo-code for the `mod`, `read`, `write`, `propagate`, and `lift` operations. Of the remaining meta primitives—`new`, `deref`, and `update`—the `new` primitive is identical to the core primitive `mod`. The `deref` primitive simply returns the value of a modifiable, and `update` destructively updates the value stored in a modifiable. These primitives are straightforward to support and will not be discussed in further detail.

The implementation maintains the following global structures: a `DDG` $= (V, R, D)$, a memo table Σ, and a priority queue Q of affected reads.

The `mod` operation extends the set of modifiables with a fresh modifiable m (that is not contained in the domain of V, i.e., $m \notin \mathrm{dom}(V)$) and returns m.

The `read` operation starts by creating a time interval. The first **slice** ensures that each `read` has its own unique start time. The second **slice** guarantees that the interval $\Delta(r)$ is nonempty. Since μ_3 is a tail of μ_1, the `read`'s interval $\Delta(r) = \delta = \mu_1 \setminus \mu_3$ will be half-open.

The `write` operation checks if the value being written is different from the value stored in the modifiable. If they differ, the readers of the modifiable are inserted into the priority queue Q. The `write` operation returns an empty interval, the current interval without modifications, and the modifiable that was written to.

Change propagation (`propagate`) updates the given interval δ by repeatedly extracting the earliest affected `read` in δ from the priority queue and

```
(* DDG is (V, R, D)
 * Q is the priority queue
 * Σ is the memo table
 *)

mod () μ = V ← V ∪ {m}, where m ∉ dom(V)
           return (∅, μ, m)

read (m,f) μ0 = μ1 ← slice_R(μ0)
                (δ0, μ2, a) ←f(Val(m))μ1
                μ3 ← slice_R(μ2)
                (δ, r) ← (μ1 \ μ3, [Rd ↦f, Src ↦m, Δ ↦ (μ1 \ μ3)])
                (R, D) ← (R ∪ {r}, D ∪ {(m, r)})
                return (δ, μ3, a)

write (m,n) μ = if Val(m) ≠n then
                    update(m,n)
                    Q ← Q ∪ {r | (m, r) ∈ D}
                return (∅, μ,m)


propagate δ = while r ← extractMin_δ(Q) do
                (_, μ, _) ← Rd(r)(Val(Src(r)))(Δ(r))
                R ← {r' ∈ R | Δ(r') ⊄ μ}
                (D, Q) ← (D|_R, Q|_R)

lift f (s,n) μ = case lookup_{Σ,μ}(f,s) of
                   Found(a, δ,m) ⇒ (π, μ') ← split μ at δ
                                   R ← {r ∈ R | Δ(r) ∩ π = ∅}
                                   (D, Q) ← (D|_R, Q|_R)
                                   write (m, n)
                                   propagate δ
                                   return (δ, μ', a)
                 | NotFound ⇒ m ← mod()
                              write (m,n)
                              (δ, μ', a) ← f(s,m)μ
                              Σ ← Σ[f(s, ·) ↦ (a, δ,m)]
                              return (δ, μ', a)
```

Fig. 10. An interval-passing semantics of the interface.

re-executing its reader. The re-execution takes place in the original interval of the read to ensure the following invariant: only the part of the virtual time line that belongs to the re-executed read is modified by the re-execution. When the re-execution is completed, the elements of R, D, and Q that do not belong to the new interval are trimmed by restricting R, D, and Q to the new interval μ. We write $X\lfloor_R$ to denote the restriction of X to elements that do not have `reads` outside R. Change propagation for an interval δ stops when the queue contains no read operations that are contained within δ.

The `lift` operation takes a function `f` along with a *strict argument* `s` and a *nonstrict argument* `n`. A memo lookup seeks for a call to `f` within the time interval μ whose strict argument is equal to `s`. When checking for the equality of

the strict arguments, the `lookup` operation uses shallow equality: two locations are considered equal if they have the same address (or identity). If the lookup succeeds, it returns a result a, an interval δ, and a modifiable m that contains the values of the nonstrict argument. The algorithm extracts the interval δ from μ using `split`, writes the new nonstrict argument into m, and change-propagates into δ, thereby adjusting the reused computation to all changes. The algorithm then expunges the elements that belong to the interval π and returns. This step ensures that all elements of the DDG that do not belong to the current computation are removed. If the memo lookup cannot find an appropriate entry, then the algorithm creates a new modifiable m, writes the nonstrict argument into m, and applies `f` to s and m. Finally, the algorithm stores a, m, and δ in the memo table Σ.

7. IMPLEMENTATION

We describe an implementation of the library interface presented in Section 4. The implementation is based on the algorithms described in Section 6 and relies on several efficient data structures (Section 7.1). The implementation enforces an important property we call *space-integrity*, which enables us to bound the memory usage (Section 7.2). Following that, we discuss some optimizations that help reduce constant factors involved in the implementation (Section 7.3). Finally, we speculate why applications of our library seem to constitute challenging benchmarks for current garbage collectors (Section 7.4).

7.1 Data Structures

Intervals. Since our algorithms operate on the time line extensively, maintaining it efficiently is critical for performance. We therefore use the (amortized) constant-time order maintenance data structure of Dietz and Sleator [1987] (our implementation, however, follows the simplified description of Bender et al. [2002]).

We maintain a global (virtual) time line that consists of a totally ordered set of time stamps. An interval is represented as a pair of time stamps, for example, the interval $(t_1, t_2]$ is represented with the the pair (t_1, t_2). Three operations are used to maintain the time line: the `insert` operation inserts a new time stamp immediately after a given time stamp on the time line; the `delete` operation removes a given time stamp from the time line; and the `compare` operation compares the ordering of time stamps on the time line. In addition to the time line, the implementation maintains two time stamps, called the *current time* and the *finger*. Together these define the *current interval*.

The `insert`, `delete`, and `compare` operations suffice to support all operations on intervals (Section 6.2). The `slice` operation is implemented by inserting a new time stamp t after the current time and setting the current time to t. The `split` operation is implemented by deleting all time stamps between the current time and the desired interval. Since the implementation of the `slice` operation advances the current time, an explicit use of the `coalesce` operation never arises.

Dynamic Dependence Graphs. The implementation globally maintains the current dynamic dependence graph (DDG). Each read consists of a closure and a pair of time stamps representing its interval. Each modifiable reference is implemented as a reference to a tuple consisting of a value and a read list. A *read list* is maintained as a doubly-linked list of reads.

Memo Tables. Memo tables are implemented as standard hash tables with chaining (see e.g., Knuth [1998]). Each entry represents (1) a function call, whose data is the result; (2) the time interval (two time stamps); and (3) the nonstrict arguments of the call. Memo lookups are performed by hashing the strict arguments of the call. A lookup succeeds if there is a memo entry within the current interval that has the same strict arguments. Insertion and deletion operations are supported as usual. The implementation ensures that the load factor of the hash tables does not exceed 0.5 by doubling the tables as necessary.

For fast equality checks and hashing, the implementation relies on boxing (a.k.a. tagging). Every strict argument to a lift function is required to be tagged with a unique identity (an integer). Since ML is type-safe, values of different types can use the same tag value. These tags are used both for computing the hashes and for resolving collisions.

7.2 Space Integrity

Consider running a program P with some input and then performing a sequence of change-and-propagate steps, where each step makes some changes and runs change propagation. Space integrity means that after performing all the changes, the total space usage is guaranteed to be the same as the space used by a from-scratch execution of P with the final input. The property implies that the space usage is independent of the past operations (i.e., history).

The implementation ensures space integrity by eagerly releasing all references to trace elements (modifiables, reads, memo entries, and time stamps) that do not belong to the current computation. This makes the trace elements available for garbage collection as soon as possible. Since modifiable references are only referenced by reads and memo entries; and since reads, memo entries and time stamps are not referenced by any other library data structures, releasing the reads, memo entries, and time stamps suffices to ensure space integrity.

To enforce eager releasing, the implementation ensures that all *live* reads and memo entries have an associated time stamp. Since all live reads have their own time intervals, their start time stamps can be associated with the read operations. Similarly, if a memoized computation has a nonempty interval, then we associate its last time stamp with the corresponding memo entry. If the memoized computation has an empty interval, then the library creates a time stamp and associates the entry with that time stamp.

The idea behind eager releasing is to maintain *back pointers* from each time stamp to the reads and memo entries that it is associated with. Thus, when the time stamp is deleted, the library can actively delete the corresponding reads and memo entries. More specifically, consider a time stamps t being deleted.

First, t is removed from the order maintenance data structure. Second, the back pointers of t are used to find the read or the memo entries associated with t. Third, depending on what was found in the second step, either the read is removed from the read list that contains it, or all associated memo entries are removed from the memo tables that they are contained in.

We have found that the space-integrity property is critical for the effectiveness of the library. Our earlier implementations suffered from space explosion because they lazily released reads and memo entries (by flagging deleted reads and memo entries and releasing flagged objects when next encountered).

To verify that our implementation ensures space integrity, we implemented a space-profiling version of our library. Using the library, we experimentally verified that the numbers of all live library-allocated objects (modifiables, reads, time stamps, memo entries, and memo tables) after various sequences of change-and-propagate steps are equal to the numbers obtained by a from-scratch execution.

7.3 Optimizations

Self-adjusting computation offers opportunities for new optimization techniques. In this section, we describe two such optimizations that proved to be particularly useful.

Single Reads. In many applications, most modifiables are read only once. For example, for all the applications considered in this article, the average number of reads per modifiable is less than 1.5 (cf., Section 9). To take advantage of this property, we implemented a version of the read lists data structure described in Section 7.1 that is specialized to contain no extra pointers when the list consists of only one read. When the list contains multiple reads, the data structures separate one of the reads as special and place all other reads in a doubly-linked list. The first element of the doubly-linked list points to the special list and the second element. This data structure can be thought of as a doubly-linked list that has two base cases: an empty and a single-read case.

This optimization makes the operations on read lists slightly more complicated than a standard doubly-linked lists, and also complicates the eager deletion of reads—when a read becomes the only read, then its release closure may need to be updated to account for this change. Since many modifiables are read only once, we found that this optimization can improve running times and reduce memory consumption. We also experimented with read lists optimized for both single and double reads, but observed no significant improvement over the single-read optimization.

Inlining `lift` *Operations.* This optimization eliminates modifiables and reads that are created by the lift operations due to nonstrict arguments when the nonstrict arguments are only passed to a tail call. The idea is to store the value of the nonstrict arguments directly in the memo table. When a memo match takes place, the recursive tail call is performed explicitly. This optimization can be viewed as inlining the lift operation; it saves a modifiable, a read, and two time stamps.

As an example, consider the call `f(s,n)` to a lift function `f` with a strict argument `s` and a nonstrict argument `n`. When called, the function first performs a memo lookup using `s`. If there is a memo match, then the match yields a result `r` and a modifiable `m` that stores the value of the nonstrict argument for the reused call. Before returning the result, the function writes `n` into `m` and performs change propagation. This change propagation adjusts the computation according to the value the nonstrict argument `n`. If n was just passed as argument to a tail call, then change propagation simply executes that call. With the optimization, the tail call is executed explicitly instead of relying on change-propagation.

To support this optimization, the library provides a version of the `mkLift` primitive that takes the function to be tail-called as a separate argument. To use the optimization, the programmer specifies the tail call explicitly. The opportunity for this optimization arises often. In our benchmarks, we have observed that the optimization can reduce the running time for both change propagation and from-scratch execution up to 40%, depending on the application.

7.4 Garbage Collection

Self-adjusting computation poses a number of challenging garbage-collection problems, particularly for tracing garbage collectors—perhaps the most commonly used collectors—which include copying, generational, and mark-and-sweep collectors. Tracing collectors identify the live and dead memory objects by performing a memory traversal to determine reachable objects starting from a set of roots, and reclaim the dead, unreachable objects. Since they traverse all of live memory, a tracing garbage collector remains efficient only when the size of the live data is small relative to the size of available heap. When the size of the live memory is large, a tracing garbage collector can slow down a program and even change its asymptotic complexity.

As a simple example, consider an execution where the heap space is available only for one more memory object, while the total amount of live space is linear in the input size. Now, every time we attempt to allocate an object, the run-time system will need to perform GC, which takes linear time. Thus, every memory allocation will effectively require linear time, and the asymptotic complexity of the program will increase by a linear factor. We can provide a more careful analysis by generalizing this example. It is known that with tracing garbage collectors, if f is the fraction of live memory, the cost for each reclaimed memory cell is $O(1/(1-f))$ (e.g., Jones [1996]), which grows very quickly as the fraction of live memory increases, approaching infinity in the limit (as f becomes 1). In particular, note that if $f = (n-1)/n$, where n is the input size, then the cost of GC for each reclaimed memory location is $O(n)$. Since self-adjusting programs record and maintain an execution trace, they tend to have large live data, potentially causing tracing collectors to slow down change propagation significantly.

Generational garbage collectors can somewhat alleviate this critical dependence on the size of the live data by dividing the heap into generations and garbage-collecting the younger generations first. They aim to reduce the

amount of tracing required by giving higher priority to those object that are more likely to become unreachable. For the approach to be effective, programs must conform to the so-called generational hypothesis, which postulates that the younger objects are more likely to become unreachable than the older ones. Unfortunately, self-adjusting programs do not conform to the generational hypothesis. In particular, a significant portion of the memory, that is, trace objects (modifiables, reads, memo table entries, and time stamps), are expected to have long lifespans, even if they are young. Furthermore, due to the side-effecting nature of `write` operations, it is common for old data to point to new data, making it more difficult to divide the memory into generations.

For these reasons, we expect tracing collectors to perform poorly with self-adjusting programs, except when the total live data is relatively small compared to the size of the available heap. Indeed, our experiments confirm these predictions (Section 9.9). A recent paper further discusses these challenges and offers a proposal for overcoming them [Hammer and Acar 2008].

8. BENCHMARKS

We evaluate the effectiveness of the approach using the following set of benchmarks.

—*Filter*. Takes a list ℓ and a boolean predicate p and returns the list of elements of ℓ that satisfy the predicate p.
—*Fold*. Takes a list ℓ and an associative binary operator and applies the operator to the elements of ℓ to produce a single value.
—*Map*. Takes a list ℓ and a function f and constructs a new list by applying f to each element of ℓ.
—*Quick-sort*. The quick-sort algorithm for list sorting.
—*Merge-sort*. The randomized merge-sort for list sorting.
—*Quick-hull*. The quick-hull algorithm for convex hulls [Barber et al. 1996].
—*Ultimate*. A randomized version of Chan's ultimate convex-hull algorithm [Chan 1996; Bhattacharya and Sen 1997].
—*Diameter*. Shamos's algorithm for finding the diameter of a set of points [Shamos 1978].

These benchmarks are chosen to span a number of computing paradigms, including simple iteration (`filter`, `map`, `split`); accumulator passing (`reverse`, `quick-sort`); incremental result construction (`graham-scan`); random sampling (`fold`); and divide-and-conquer (`merge-sort`, `quick-sort`, `ultimate`).

The `graham-scan` algorithm combines a convex-hull algorithm with a linear scan. The `diameter` algorithm combines a convex-hull algorithm with a linear scan. For some of the problems in this list, asymptotic bounds achieved by our apporach are shown to closely match the best bounds achievable by special-purpose algorithms developed in the algorithms community [Acar 2005].

Throughout our discussion of these benchmarks, we let n denote the input size.

List primitives (`filter`, `fold`, `map`, `reverse`, and `split`): Except for `fold`, all of these benchmarks perform a traversal of the list as they iteratively construct the output. As we have seen already (Section 5.3), a straightforward list-traversal implementation of `fold` is *unstable*. Instead, we implemented `fold` using a random-sampling technique. All list primitives, except for `fold`, have constant stability, and hence have constant update time for small changes (e.g., an insertion or deletion); `fold` is $O(\log n)$ stable and has $O(\log n)$ update time.

Sorting algorithms (`quick-sort` and `merge-sort`): Both the `quick-sort` and the randomized `merge-sort` algorithms are standard. The `quick-sort` algorithm uses the first element of the input as the pivot for partitioning the input list. The algorithm avoids concatenating the results by passing the sorted half in an accumulator. The randomized `merge-sort` algorithm divides its input into two sublists by randomly assigning each element in the input to a destination sublist. The sublists are then sorted recursively and merged as usual. We could also use the deterministic version of merge-sort, but it is not as stable as the randomized algorithm. Both algorithms require $O(n \log n)$ time for a complete run and are $O(\log n)$ stable (in expectation) under insertions and deletions [Acar 2005] (this is optimal). In our implementation, both algorithms use the `split` primitive to partition the input.

Computational-geometry algorithms (`quick-hull`, `ultimate`, and `diameter`): This group of algorithms consists of a number of convex-hull algorithms and an algorithm for computing the diameter of a point set (`diameter`). The convex hull of a set of points is the smallest polygon that encloses the point set. The *static* convex hull problem is to compute the convex hull of a static (unchanging) set of points. The *dynamic* convex hull problem is to maintain the convex hull of a set of points as the users are allowed to add and remove points. Both the static [Graham 1972; Kirkpatrick and Seidel 1986; Chan 1996; Barber et al. 1996; Wenger 1997] and the dynamic [Overmars and van Leeuwen 1981; Chan 1999; Brodal and Jacob 2002] settings have been studied extensively for over two decades.

Of the convex hull algorithms we consider, the randomized ultimate hull algorithm is the most sophisticated. The algorithm is optimal in the size of the output (not just the input). This divide-and-conquer algorithm performs a special elimination step before each recursive call. The elimination step is guaranteed to remove a constant fraction of points from the input of a recursive call; this is crucial to the optimality of the algorithm. Figure 11 illustrates how a randomized version of the algorithm works. Because of symmetry, we describe only how the upper half of the hull is constructed. Given the leftmost (L) and the rightmost (R) points, the algorithm picks a random pair of points (A, B) and finds the farthest point (M) from the line (A, B). The algorithm then pairs the points randomly and eliminates a constant fraction of the points in expectation by applying a constant time test to each point. The algorithm then computes the left and right halves of the problem defined by the extreme points (L, M) and (M, R), respectively.

The diameter of a set of points is the maximum distance between any pairs of points. It is straightforward from the definition that such a pair is on the

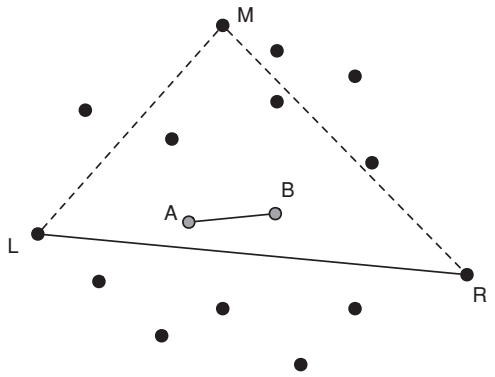


Fig. 11. Chan's ultimate hull algorithm for computing convex hulls.

convex hull of the point set. We use Shamos's algorithm [Shamos 1978] to compute the diameter. This algorithm first computes the convex hull and traverses it while computing the maximum distance between anti-podal points. In our implementation, `diameter` uses the quick-hull algorithm to compute the convex hull.

All our computational geometry benchmarks rely on the list primitives. For example, the `ultimate` and `quick-hull` benchmarks use `fold` to find the point furthest away from a given line.

9. EXPERIMENTS

We describe an experimental evaluation of the self-adjusting-computation framework by using the benchmarks described in Section 8. We report detailed results for three benchmarks and a summary of results for the remaining benchmarks. In these benchmarks, input changes involve inserting into and deleting from the input. We also investigate the effects of heap size on the performance of self-adjusting programs (Section 9.9).

9.1 The Test Suite

Our test suite consists of *static* (non-self-adjusting) and *self-adjusting* versions of our benchmarks (Section 8). The `filter` benchmark selects all odd elements from an integer list. The `map` benchmark takes an integer list as an argument and outputs a new list by adding a fixed constant to each element of the input list. We derive two benchmarks from `fold`: the `minimum` benchmark computes the minimum of a list of integers, and the `sum` benchmark computes the sum of floating-point numbers in a list. Each of our list benchmarks—`filter`, `map`, `minimum`, `sum`—performs 5 additional arithmetic operations each time an element is accessed. Our sorting benchmarks (`quick-sort` and `merge-sort`) sort strings. All convex-hull and the diameter benchmarks compute the convex hull and the diameter, respectively, of a set of points in two dimensions.

To give a sense of how much the standard and self-adjusting versions of our applications differ, Table I shows the number of lines and the number of tokens for both versions of our sorting and convex-hull benchmarks (as counted by the

Table I. Number of Lines and Tokens for Some Applications

Application	Static # Lines	Self-Adj. # Lines	Static # Tokens	Self-Adj. # Tokens
`merge-sort`	94	109	262	335
`quick-sort`	47	62	152	215
`quick-hull`	117	126	405	425
`ultimate`	207	208	630	697
`diameter`	177	199	558	660

`wc` utility).[4] The self-adjusting versions contain about 10% more lines and 20% more tokens than their static versions (on average). Much of this effectiveness is because we can compose self-adjusting functions, just as in conventional programming.

9.2 Inputs, Input Changes, and Measurements

In our evaluation, we use randomly generated datasets. To generate a list of n integers, we choose a random permutation of the integers from 1 to n. To generate a list of strings, we first generate a list of integers and then map the integers to their string representation (in decimal). For the computational geometry benchmarks, we generate an input of n points by picking floating-point numbers uniformly at random from the square $[0, 10n] \times [0, 10n]$.

When measuring the running time of our benchmarks, we exclude the time for input generation, and we measure the following quantities.

—*Time for from-scratch execution*. The time for executing the ordinary or the self-adjusting versions from scratch.

—*Average time for an insertion/deletion*. This is measured by applying a delete-propagate-insert-propagate step to each element. In each step, we delete an element, run change propagation, insert the element back, and run change propagation. The average is over all insertions and deletions performed (i.e., $2n$ operations, where n is the input size.)

—*Time for batch insertions/deletions*. This is measured by performing the following: for each element in the list, we delete the next k elements, run change propagation, reinsert these elements back, and run change propagation. We report the time taken to complete the experiment. Note that such an experiment involves a total of $2(n - k + 1)$ change-propagation operations.

—*Overhead*. This is the ratio of the time for the from-scratch execution of the self-adjusting version to the time for the from-scratch execution of the ordinary version with the same input.

—*Speedup*. This is the ratio of the time for the from-scratch run of the ordinary version to the average time for insertion/deletion.

—*Trace size*. We measure the trace size (the internal data structures maintained by self-adjusting computation to support efficient change propagation) by counting the numbers of (1) modifiables created, (2) reads performed, and (3) memo entries created in an initial execution.

[4]We do not report numbers for the list primitives because they simply call SML basis library for the corresponding function. A "token" is a string of characters delimited by white spaces.

We conducted our experiments on a 3.40Ghz Intel Xeon machine, with 32 GB of memory, running Fedora Core 7. We compiled our benchmarks with the MLton compiler version 20070806 using "`-runtime fixed-heap 30G gc-summary`" options unless otherwise stated. The "`fixed-heap 30G`" option directs the run-time system to use 30GB of memory on the system. More specifically, with this option, MLton allocates 30 GB of heap space divided into a to-space and a from-space (each 15 GB) and uses the copying collector until the live data exceeds 2 GB, at which point it switches to mark-and-compact collection, making all of the 30 GB available for allocation. Consequently, if the total allocations performed by an application is less than 30 GB, the GC time is likely to be negligible. The "`gc-summary`" option directs the run-time system to collect summary information about garbage collection (GC). In particular, the system reports the percentage of the total time spent garbage collecting.

In our experiments, we measure two types of timings that we call application time and total time. The *application time* is the user time spent for performing the experiment excluding GC time. The *total time* is the application time together with the GC time.

When measuring time, we carefully isolate the time from initialization by starting the timer after the initialization phase and stopping it after the completion of the experiment being measured. The initialization phase involves starting up the system and generating the input. In the case of the experiments for measuring the average time for an insertion/deletion, the initialization phase also requires the initial run. Since the initialization phase typically performs nontrivial computation, we force the run-time system to perform a garbage collection after the initialization and before starting the timer for the experiment. We do not force a garbage collection before terminating the experiments. Thus, if the space needed by the experiment is less than the heap size at the beginning of the experiment, then there will be no garbage collections and GC time will be zero.

The results that we report here differ somewhat from our preliminary results presented in the conference version of this article [Acar et al. 2006b]. The primary difference is that the conference version includes the time for the initialization phase in all timings (thus taking end-to-end timings) whereas here we exclude the time for the initialization phase. For reasons we describe next, we believe that the timings reported here provide a more accurate account of the actual performance than those shown in the conference version. When the actual computation is relatively inexpensive compared to the initialization, for example, in the list primitives, such as `filter` and `map`, including initialization penalizes the static version. In more complex benchmarks (such as sorting and computational geometry), initialization constitutes a small percentage of the total running time, causing no significant difference in the result.

9.3 Overview of Results

In the next three sections, we present detailed results for the applications `map`, `merge-sort`, and `ultimate`. These benchmarks are representative of the three classes of applications that we consider: list primitives, sorting, and

computational geometry. When discussing these benchmarks, we consider the application time, which excludes GC time. As we describe later, the GC time is negligible. We then present a summary of results for all our bechmarks (Section 9.7) at fixed input sizes that include timings both with and without GC.

One measure of the effectiveness of self-adjusting computation is the smallest input size at which a self-adjusting program becomes more efficient than its static, conventional counterpart. In Section 9.8, we measure this quantity, which we call the crossover size.

For all our benchmarks, we experimentally verify the space integrity property (Section 7.2) by checking that the size of the meta data stored by self-adjusting computation, measured by counting the numbers of active modifiables, reads, and memo entries, depends only on the current input and not on the history of changes (insertions/deletions) performed to reach that input. Our measurements show that the number of reads performed in our self-adjusting benchmarks is only slightly larger than the number of modifiables created—the number of reads per modifiable is slightly more than one on average. This finding motivates and justifies the single-read optimization (Section 7.3).

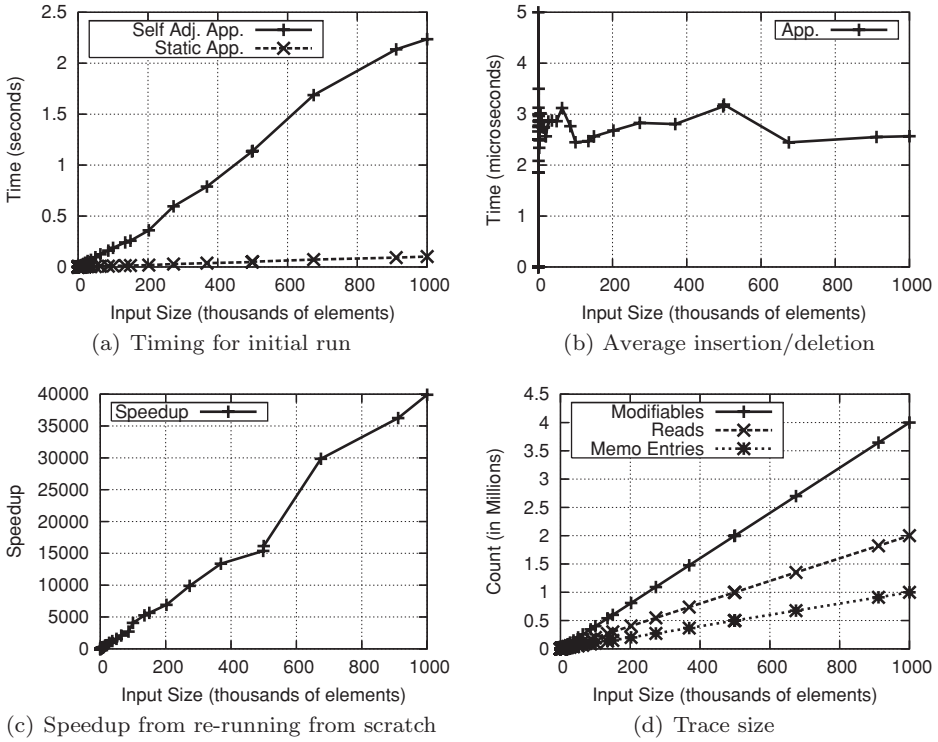
By running our experiments with a 30 GB of heap, we are able to minimize the GC cost by supplying significantly more memory than needed by our benchmarks. As we observe in the conference version of this article and elsewhere [Hammer and Acar 2008], GC can constitute a reasonably large fraction of a self-adjusting benchmark's running time. In Section 9.9, we discuss how the performance of our self-adjusting benchmarks is affected by the size of the heap.

9.4 Map

Figure 12 shows the results for `map`. The initial run graph (top left), which compares the application time for from-scratch runs of the static and self-adjusting versions, shows that the running time grows linearly in the input size. This indicates that the overhead of self-adjusting computation is constant, which in this case is about 25. The top right figure shows the average time for an insertion/deletion, which appears to be constant. The bottom left figure shows speedup, which grows linearly in the input size. As can be seen from this graph, when the input size is 1 million, change propagation is four orders of magnitude faster than recomputing from scratch. The bottom right figure shows the the number of modifiables, reads, and memo entries, which quantify the size of the trace. As the figure shows, all of these grow linearly with the input size.

9.5 Merge Sort

Figure 13 shows our experimental results for `merge-sort`. The initial-run graph (top left) indicates that self-adjusting version is a constant factor, 5 in this case, slower than the static version and that both are consistent with the $O(n \log n)$ asymptotic bound. The graph for insertions and deletions (top right) shows



(a) Timing for initial run

(b) Average insertion/deletion

(c) Speedup from re-running from scratch

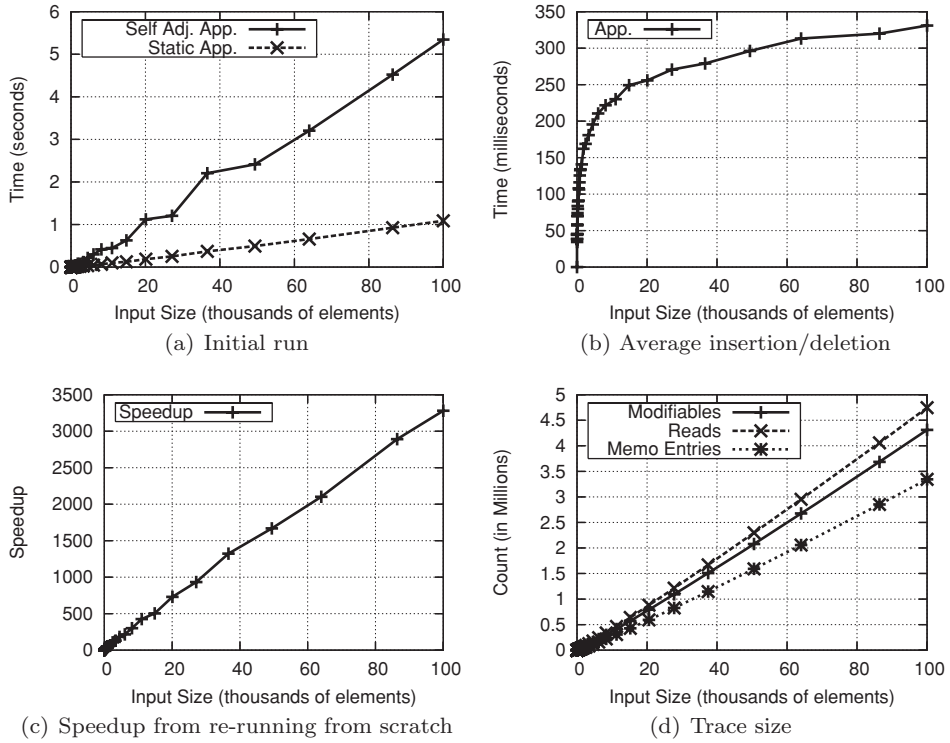
(d) Trace size

Fig. 12. Experimental results for `map`.

that the average change-propagation time grows logarithmically in the size of the input, which is consistent with the best possible (lower bounds). As can be seen from the speedup graph (bottom left), the speedup increases linearly with the input size, quickly reaching three orders of magnitude. The bottom right figure shows the number of modifiables, number of reads, and number of memo entries, which fit $O(n \log n)$ time-bound for from-scratch runs.

9.6 Ultimate

Figure 14 shows our experimental results for the ultimate convex-hull algorithm (`ultimate`). The initial run graph (top left) shows that the running time of self-adjusting version is constant factor, less than 2 in this case, slower than the static version. The graph for insertions and deletions (top right) shows that change-propagation time grows somewhat unevenly but still slowly, as with the input size. We believe that the sharp increases or decreases in the running time with differing input sizes is primarily because the ultimate algorithms is input/output-sensitive, that is, that its running time depends on the particular input/output and not just the size of the input. The speedup graph (bottom left) shows that change propagation can be more than three orders of magnitude faster than recomputing from scratch. The bottom right figure shows the the number of modifiables, number of reads, and number of memo entries to be slightly superlinear.



(a) Initial run (b) Average insertion/deletion

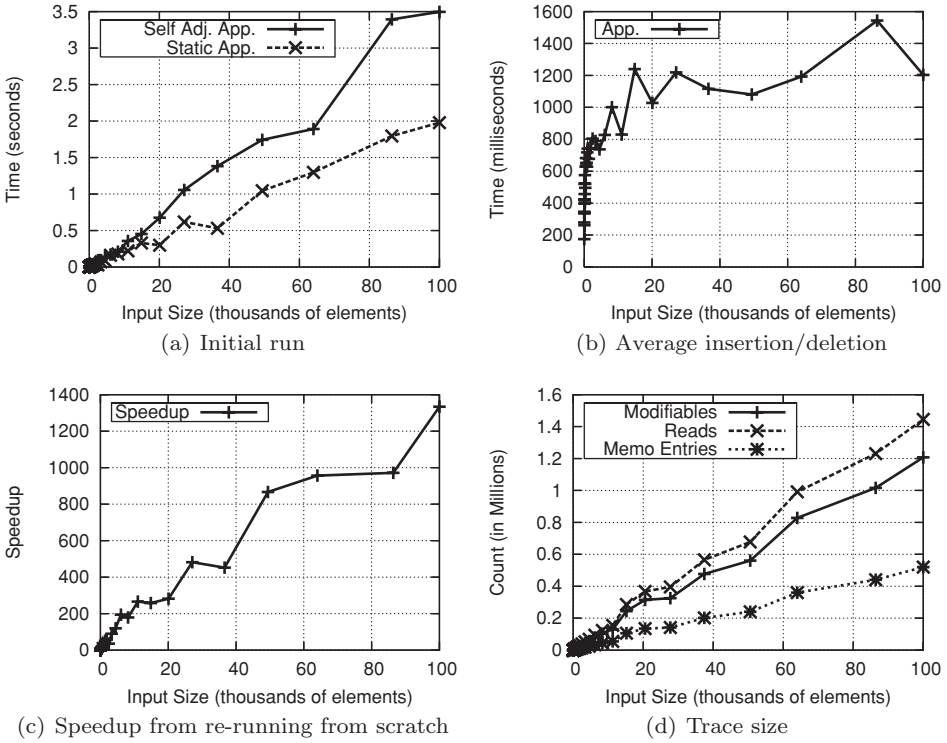
(c) Speedup from re-running from scratch (d) Trace size

Fig. 13. Experimental results for `merge-sort`.

9.7 Summary for All Benchmarks

Tables II and III show the measurements for all our benchmarks at fixed input sizes, excluding and including GC time, respectively. In both tables, the column titled "n" specifies the input size; the "Static Run" column shows the time for a from-scratch execution of the static version; the "Self-Adj. Run" column shows the time for a from-scratch execution of the self-adjusting version; and the "Self-Adj. Avg. Propagate" column shows the average time for change propagation after a modification (an insertion or a deletion). The "Overhead" and the "Speedup" columns show the corresponding quantities, as defined in Section 9.2. For example, we run the `filter` benchmark with input size $n = 10^6$ and obtain the following measurements: the ordinary version takes 0.12 and 1.89 seconds without and with the GC time, respectively; change propagation takes 3.8×10^{-6} seconds for both without and with the GC time; the overhead is a factor of 16.5 when the time for GC is excluded and 17.2 when the time for GC is included; and the speedup is a factor of 3.0×10^4 for both without and with GC time.

As can be seen from the tables, the differences between the measurements when excluding and including the GC time are negligible (less than 10%). This is because the heap size for these experiments, fixed at 30 GB, is significantly larger than the total live data needed (the live data peaks at 2.2 GB with



(a) Initial run (b) Average insertion/deletion

(c) Speedup from re-running from scratch (d) Trace size

Fig. 14. Experimental results for `ultimate`.

Table II. Summary of Benchmark Timings (without GC)

Application	Input size	Static Run (s)	Self-Adj. Run (s)	Self-Adj. Avg. Propagate (s)	Overhead	Speedup
filter	1×10^6	0.12	1.89	3.8×10^{-6}	16.5	3.0×10^4
map	1×10^6	0.10	2.24	2.6×10^{-6}	21.8	4.0×10^4
minimum	1×10^6	0.09	2.50	1.5×10^{-5}	26.4	6.1×10^3
sum	1×10^6	0.09	2.49	1.1×10^{-4}	29.0	769.72
merge-sort	1×10^5	1.09	5.34	3.3×10^{-4}	4.9	3.3×10^3
quick-sort	1×10^5	0.24	2.52	3.7×10^{-4}	10.4	654.06
quick-hull	1×10^5	1.64	3.19	2.7×10^{-4}	1.9	6.2×10^3
ultimate	1×10^5	1.98	3.50	1.2×10^{-3}	1.8	1.6×10^3
diameter	1×10^5	1.63	3.53	3.0×10^{-4}	2.2	5.5×10^3

`merge-sort`). The heap size is also reasonably large for the total allocated data, which peaks at 80 GB with `ultimate` (the maximum is reached when computing the time for an average insertion/deletion).

The tables show that overhead is moderately high for list benchmarks, `filter`, `map`, `minimum`, `sum`, but drop to reasonably small factors (less than 3) for more sophisticated applications such as the computational geometry applications, which perform more work for each call to a self-adjusting-computation primitive. We therefore expect the overheads to become smaller as the computational complexity of the application increases. As the tables show, change

Table III. Summary of Benchmark Timings (with GC)

Application	Input size	Static Run (s)	Self-Adj. Run (s)	Self-Adj. Avg. Propagate (s)	Overhead	Speedup
filter	1×10^6	0.12	1.98	3.8×10^{-6}	17.2	3.0×10^4
map	1×10^6	0.10	2.40	2.6×10^{-6}	23.4	4.0×10^4
minimum	1×10^6	0.09	2.52	2.3×10^{-5}	26.6	4.1×10^3
sum	1×10^6	0.09	2.51	1.5×10^{-4}	29.2	587.6
merge-sort	1×10^5	1.09	5.35	3.3×10^{-4}	4.9	3.3×10^3
quick-sort	1×10^5	0.24	2.53	4.8×10^{-4}	10.4	503.0
quick-hull	1×10^5	1.64	3.22	3.2×10^{-4}	2.0	5.1×10^3
ultimate	1×10^5	1.98	3.51	1.5×10^{-3}	1.8	1.3×10^3
diameter	1×10^5	1.63	3.54	3.5×10^{-4}	2.2	4.7×10^3

propagation leads to orders of magnitude speedup over recomputing from scratch. Except for two benchmarks, `quick-sort` and `sum`, the speedups exceed three orders of magnitude. For list benchmarks, which are run with inputs of 1,000,000 elements, the speedups can be as high as four orders of magnitude. With `quick-sort` and `sum`, the speedups are more than a factor of 500. The primary reason that `sum` does not obtain speedups that are as high is that it uses a different, more complicated algorithm than the static version. The static version simply traverses the input list and sums the elements, whereas the self-adjusting version performs random sampling (Section 5.4). As noted earlier, this random sampling is necessary for stability. The `quick-sort` benchmark does not yield speedups as high as `merge-sort` because `quick-sort` is less stable (e.g., inserting a new element at the head of the list requires linear work in `quick-sort`, whereas it only requires logarithmic work in `merge-sort`). The underlying reason for such high speedup numbers is the near-linear time asymptotic gap between recomputing from scratch and performing change propagation (Section 8)—as the input size increases, this asymptotic gap leads to large speedups.

9.8 Crossover Points

The results show that self-adjusting-computation benchmarks can incur moderate overheads compared to their static counterparts, but can respond to small changes to their data orders of magnitude faster than recomputing from scratch. Furthermore, the speedups obtained by change propagation tend to grow quickly (often linearly) with the input size. This motivates the question: when does change propagation become more effective than recomputing from scratch? More precisely, define an input size m to be the *crossover* if for all inputs size at least m, change propagation is faster than recomputing from scratch in the static version. Table IV shows the crossover sizes for our benchmarks (whether we include or exclude the GC time does not change the results). For all benchmarks except for `quick-sort` and `sum`, the crossover sizes are quite small, less than 100. For `quick-sort` and `sum`, they are somewhat higher, but still remain under 1000 elements. These results suggest that self-adjusting programs are faster than their static counterparts for all but some of the smallest inputs considered.

Table IV. Crossover Points

Application	Crossover
`filter`	39
`map`	70
`minimum`	100
`sum`	270
`merge-sort`	10
`quick-sort`	400
`quick-hull`	22
`ultimate`	17
`diameter`	22

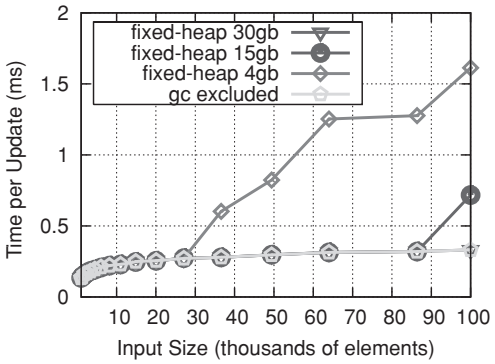


Fig. 15. Average update time for `merge-sort` as heap size is varied.

9.9 Garbage Collection vs. Available Memory

Since we allocated a relatively large heap for our experiments, our results suggest that the GC time is negligible. Unfortunately, as we discussed in Section 7.4 and observe elsewhere [Acar et al. 2006b; Hammer and Acar 2008], this is not the case in general: GC can cause the performance of self-adjusting program to degrade significantly when the heap size is not large compared to the size of the total live data. To verify this claim, we perform an experiment at different heap sizes. The experiment measures the average time for an insertion/deletion (we delete and insert each element and apply change propagation after each modification, as described in Section 9.2), the maximum size of the live memory, and total number of allocated bytes. For these experiments, we compile our benchmarks with the "`-runtime fixed-heap` *N*`G gc-summary`" options, where *N* denotes the size of the heap. Figures 15 and 16 show the average change-propagation time for an insertion/deletion with `merge-sort` and `ultimate` (respectively) with heap sizes of 4GB, 15GB, and 30GB, as well as the application time (computed with 30GB of fixed heap), which excludes the GC time.

With `merge-sort`, the experiment requires less than 2.2 GB of live memory and allocates about 18 GB of memory. Thus, when given 30 GB of heap space, the GC time is not significant. Indeed, as the figure shows, when the heap size is 30 GB, the time for change propagation overlaps with the application-time

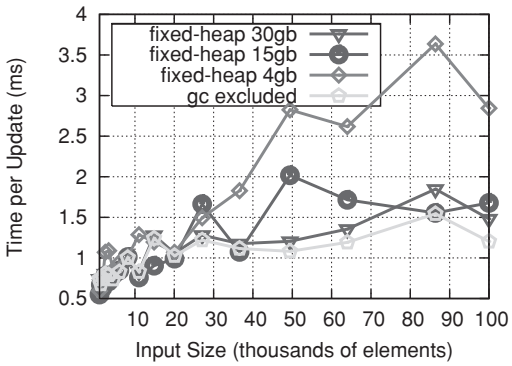


Fig. 16. Average update time for `ultimate` as heap size is varied.

line. With small heap sizes, however, GC time is more significant. With a 4 GB heap, the time for change propagation overlaps with the application-time line for smaller inputs, but starts increasing linearly when the input size exceeds about 20,000 (20K). The case for when the heap size is 15GB is similar: change-propagation time follows the application-time line up to about 90K and then starts diverging.

The case for `ultimate` is similar, except that reducing the heap size has a more pronounced impact. For these input sizes, the size of the live memory is less than 1 GB but `ultimate` allocates about 80 GB of memory throughout the experiments. With a 4 GB heap, the time for change propagation increases nearly linearly with the input size when it exceeds 20K. With larger heap sizes, garbage collection time largely follows the application-time line but is not negligible, since `ultimate` performs significantly more allocation and thus requires at least some garbage collection.

These measurements indicate the prediction that GC can change the asymptotic run-time behavior of change propagation when the heap size is small compared to the size of the maximum live data (Section 7.4). The measurements also indicate that `merge-sort`, which has twice as much live data as `ultimate`, slows down more significantly with a small heap (4 GB), even though `ultimate` allocates more than a factor of 4 more memory. This is consistent with the standard analysis of garbage collection (Section 7.4) that shows that it is the fraction of the live memory to that of the total memory that matters, not the amount of memory allocation.

10. RELATED WORK

The problem of adapting computations to small changes to their data has been studied extensively in several communities. In this section we review some of the previous work in the programming-languages community, broadly known as incremental computation, in the algorithms community, and other related work on self-adjusting computation.

10.1 Incremental Computation

The goal of the research on incremental computation in the programming-languages community is to devise general-purpose, language-centric techniques that enable programs to 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 among data in a computation, so that a change-propagation algorithm can update the computation when the input is changed. Demers et al. [1981] and Reps [1982] introduced the idea of *static dependence graphs* and presented a change-propagation algorithm for them. Hoover [1987] generalized the approach outside the domain of attribute grammars. Yellin and Strom [1991] used the dependence graph ideas within the INC language, and extended it by having incremental computations within each of its array primitives. The key limitation of static dependence graphs is that they are not general purpose: they do not permit the change-propagation algorithm to update the dependence structure. This limitation restricts the types of computations to which static-dependence graphs can be applied. For example, the INC language, which uses static dependence graphs for incremental updates, does not permit recursion.

The limitations of static dependence graphs motivated researchers to look into alternatives. Pugh and Teitelbaum [1989] applied memoization (also called function caching) to incremental computation. Memoization is a classic idea that dates back to the late 1950s [Bellman 1957; McCarthy 1963; Michie 1968]. It is general purpose, that is, applies to any purely functional program. Since the work of Pugh and Teitelbaum, others have investigated applications of various forms of memoization to incremental computation [Abadi et al. 1996; Liu et al. 1998; Heydon et al. 2000; Acar et al. 2003]. The idea behind memoization is to remember function calls and their results, and reuse them whenever possible. In the context of incremental computation, memoization can improve efficiency when re-executions of a program with similar inputs perform similar function calls. Although the reader may expect this to be intuitively true, it often is not. In fact, the effectiveness of memoization critically depends on the structure of the program and the kind of the input change being considered. For many computations, it is often possible to find an input change that prevents a large part of the computation from being reused (Section 2.2). Intuitively, the problem is that with memoization, all function calls that consume modified data and all their ancestors in the function call tree need to be re-executed (because these functions will notice that their arguments have been modified).

Other approaches to incremental computation are based on partial evaluation. Sundaresh and Hudak's approach [Sundaresh and Hudak 1991] 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 of the partition can be changed incrementally. The main limitation of this approach is that it allows input changes only within a predetermined partition. Field [1991], and Field and Teitelbaum [1990] present techniques for incremental computation in the context of lambda calculus. Their approach is

similar to Hudak and Sundaresh's, but present formal reduction systems that optimally use partially evaluated results.

For a more complete list of references on incremental computation, we refer the reader to the bibliography of Ramalingam and Reps [1993].

General-purpose techniques for incremental computation have also been investigated in the artificial-intelligence and logic-programming communities. In particular, the so-called "Truth Maintenance Systems" maintain relationships between proposition symbols as the values of these symbols and relationship between them change incrementally (e.g., [Stallman and Sussman 1977; Doyle 1987; Mcallester 1990]). These systems typically track dependences between proposition symbols and the relationships that they affect and use these dependences to perform efficient updates.

10.2 Dynamic Algorithms

The problem of devising efficient programs that can respond to incremental changes is typically approached from a diferent perspective in the algorithms community. In contrast to the programming-languages community, which focuses on general-purpose techniques, the algorithms community initiated the study of *dynamic algorithms* or *dynamic data structures* (e.g., [Sleator and Tarjan 1985, Chiang and Tamassia 1992; Eppstein et al. 1999]), where the main goal is to develop efficient solutions for individual problems. In a nutshell, dynamic data structures are a class of data structures capable of efficiently answering specific kinds of queries while allowing the user to modify the input (e.g., inserting/deleting elements). As an example, a dynamic data structure for computing the diameter (the distance between the farthest pair of points) allows the user to insert/delete points into/from a set of points, while on request, it is capable of efficiently reporting the diameter. Since dynamic algorithms are designed to carefully take advantage of the structural properties of the specific problems considered, they are often very efficient: we often see a linear-time (or more) gap between the update time of a dynamic algorithm and the run-time of its optimal static version.

Practical experience suggests that it is sometimes possible to devise simple algorithms that are not asymptotically efficient (in an easily quantifiable way), but nevertheless work reasonably well in practice. The reader may feel that dynamic or incremental problems may be of this nature. Unfortunately, this is not the case even for problems whose static versions are simple. Consider, as an example, the problem of computing the convex hull of a set of points as the point set changes (a convex hull is the smallest polygon enclosing the points). When a new point is inserted, we can compute the updated hull by traversing the hull and inserting the point as necessary. Performing this update efficiently requires designing and implementing the so-called point-location data structure that can locate the part of the hull that is visible by the point being inserted; this is a relatively difficult task. When a new point is deleted, we can compute the updated convex hull by finding the set of points that now become a member of the hull. Devising an efficient mechanism to perform this update is even more complicated than the insertion case. Indeed, this problem has been studied

extensively. Overmars and van Leeuwen's algorithm was the first to support insertions and deletions efficiently, in $O(\log^2 n)$ time. Reducing the update time to $O(\log n)$ took nearly two more decades [Brodal and Jacob 2002].

The convex-hull algorithm is not an anomaly. The dynamic version of static problems often appear to be significantly more difficult. For example, giving an algorithm for the minimum spanning tree (MST) problem is relatively straightforward in the static case. The dynamic MST problem, however, is much more difficult: providing an efficient solution to this problem is still an active topic after decades of research (see e.g., [Frederickson 1985; Eppstein et al. 1997; Henzinger and King 1997, 1999; Holm et al. 2001]). The MST problem is not an exception. Other examples include the problem of dynamic trees [Sleator and Tarjan 1983] (which is trivial in the static case), whose various flavors have been studied extensively [Sleator and Tarjan 1983, 1985; Cohen and Tamassia 1991; Radzik 1998; Henzinger and King 1999; Tarjan 1997; Alstrup et al. 1997; Frederickson 1997; Alstrup et al. 2003; Tarjan and Werneck 2005b, 20005a].

There have been proposals for more general algorithmic techniques that can be used to dynamize static algorithms. Bentley and Saxe's approach can be used to dynamize a certain class of divide-and-conquer algorithms, the so-called decomposable search problems [Saxe and Bentley 1979; Bentley and Saxe 1980]. Another approach based on the so-called influence and conflict graphs can be used to dynamize certain randomized algorithms (e.g., [Mulmuley 1994; Boissonnat and Yvinec 1998]). Although somewhat more general than the ad hoc approach, these approaches are still far from automatic: they require problem-specific algorithm design, for example, the data structures to be used.

Since, for efficiency, dynamic algorithms need to exploit the structure of the specific problems and the types of modifications they support, they are usually highly specialized (an algorithm may be efficient for some modifications to data but not others), naturally more complex than their static versions, and not composable. By composability, we refer to the ability to send the output of one function to another as input, that is, the composition of `f(·)` and `g(·)` is `f(g(·))`. Combined with the increased complexity, 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 efficiently performing motion simulations [Basch et al. 1999]. These data structures take advantage of the incremental nature of continuous motion (i.e., the computed properties seldom change combinatorially) by efficiently updating computed properties. Many kinetic data structures have been proposed and some have also been implemented (see e.g. [Agarwal et al. 2002; Guibas 2004] for surveys). These data structures share many characteristics of dynamic data structures. For example, there is no known efficient algorithm for performing motion simulation of convex hulls in three dimensions, even though the static version of the problem is very well understood (the kinetic 3D convex-hull problem has been open for one decade now [Guibas 1998]). In general, kinetic problems are as hard as dynamic problems, because when composed, kinetic problems often require handling

dynamic modifications to data such as insertions/deletions (e.g. [Alexandron et al. 2005]). They also pose additional implementation challenges [Agarwal et al. 2002; Guibas and Russel 2004; Russel et al. 2007; Russel 2007], due to the difficulties in motion modeling and handling of numerical errors.

10.3 Self-Adjusting Computation

The first work on self-adjusting computation [Acar et al. 2002], called adaptive functional programming (AFP), generalized dependence-graph approaches by introducing dynamic dependence graphs (DDGs) and providing a change-propagation algorithm for them. AFP can be applied to any purely functional program. This is made possible by a change-propagation algorithm that can update the dependence structure of the DDG by inserting and deleting dependences as necessary. Adaptive functional programs can be written by using type-safe linguistic facilities that guarantee safety and correctness of change propagation.

Although DDGs and AFP are general purpose, the effectiveness of change propagation is limited: certain modifications can require as much time as recomputing from scratch. This article proposed techniques for combining DDGs with memoization to dramatically improve the effectiveness of change propagation. In follow-up work, we present a semantics for the approach proposed here and prove correctness with mechanically checked proofs [Acar et al. 2007a].

Both in AFP and in the approaches proposed here, the key linguistic notion is that of a modifiable (reference), which holds the data that can change over time. Although modifiables are closely related to references, and the mutator program or the user can update their contents arbitrarily, all the aforementioned work requires that they be written no more than once within the self-adjusting program. Consequently, by using the techniques proposed here, we can make self-adjusting purely functional programs only. Recent work [Acar et al. 2008a] extended self-adjusting computation techniques proposed here to imperative programs that update modifiables (memory) destructively.

The aforementioned approaches to self-adjusting computation rely on specialized linguistic primitives that require a programming style closely related to monadic primitives. The reason for this is the desire to track dependences selectively, so that only dependences on data that can change over time are tracked. If selective dependence tracking is not desired, then it is possible to track all dependences without requiring programmer annotations [Acar 2005]. Earlier implementations of self-adjusting computation, which include the SML library [Acar et al. 2002] and its Haskell implementation by Carlsson [2002], and the proposed implementation here use this monadic interface, which can make it cumbersome to write self-adjusting programs by requiring substantial restructuring of existing code. More recent work proposed compilation techniques for self-adjusting computation that significantly reduce the burden of annotation overheads. The idea is to extend existing languages with several, simple-to-use, self-adjusting-computation primitives and generate self-adjusting code from code annotated with these primitives by using various static and dynamic analysis. Existing approaches include the SML-based Delta

ML language and its compiler [Ley-Wild et al. 2008, 2009], and the C-based, CEAL language and its compiler [Hammer et al. 2009; Hammer and Acar 2008]. An important advantage of the compilation-based approach is that it enables giving a precise cost semantics for analyzing the asymptotic complexity of self-adjusting programs [Ley-Wild et al. 2009].

Self-adjusting computation has been applied, in various incarnations, to a number of problems from a reasonably broad set of application domains, including dynamic algorithms, motion simulations, machine learning, incremental invariant checking [Shankar and Bodik 2007]. These applications confirm that the approach can be effective for a broad range of applications, often matching best-known asymptotic bounds, and performing efficiently in practice. For some problems, the approach even proved instrumental in solving problems that resist ad hoc approaches, for example, motion simulation of three-dimensional convex hulls [Acar et al. 2008b], and statistical inference on graphical models [Acar et al. 2007b]. For a discussion of these applications and a broader set of references, we refer the reader to a recent survey [Acar 2009].

Broadly speaking, self-adjusting computation aims to bridge the gap between programming-language-centric techniques, which tend to be general-purpose but suboptimal, and algorithmic approaches, which tend to be efficient but complex and problem-specific, by enabling the programmer to write self-adjusting programs much like ordinary programs, while facilitating such programs to efficiently respond to small modifications to their data.

11. CONCLUSION

This article describes and evaluates an approach, called self-adjusting computation, to the problem of enabling computations to respond to modifications to their data efficiently by using a combination dynamic dependence graphs and memoization. Due to interesting interactions between DDGs and memoization (e.g., memoization requires purely functional code, DDGs use side effects), combining them requires care. This article describes algorithms for combining them efficiently. We implemented the proposed approach as a Standard ML library, which offers an interface that enables the programmer to translate an ordinary program into a self-adjusting program by instrumenting the code. For efficiency, the library relies on several optimizations and satisfies a space-integrity property which ensures that the total space consumption can be bounded in terms of the most recent input size and does not depend on history. Using the library, we implemented a reasonably broad range of benchmarks and performed an extensive experimental evaluation. Our experiments demonstrate that the proposed approach often yields orders of magnitude speedups over recomputing from scratch.

REFERENCES

ABADI, M., LAMPSON, B. W., AND LÉVY, J.-J. 1996. Analysis and caching of dependencies. In *Proceedings of the International Conference on Functional Programming*. 83–91.

ACAR, U. A. 2005. Self-adjusting computation. Ph.D. dissertation, Department of Computer Science, Carnegie Mellon University.

ACAR, U. A. 2009. Self-adjusting computation (an overview). In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York.

ACAR, U. A., AHMED, A., AND BLUME, M. 2008a. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

ACAR, U. A., BLELLOCH, G. E., TANGWONGSAN, K., AND TÜRKOĞLU, D. 2008b. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*.

ACAR, U. A., BLUME, M., AND DONHAM, J. 2007a. A consistent semantics of self-adjusting computation. In *Proceedings of the 16th Annual European Symposium on Programming (ESOP)*.

ACAR, U. A., IHLER, A., METTU, R., AND SÜMER, O. 2007b. Adaptive Bayesian Inference. In *Neural Information Processing Systems (NIPS)*.

ACAR, U. A., BLELLOCH, G. E., BLUME, M., HARPER, R., AND TANGWONGSAN, K. 2006a. A library for self-adjusting computation. *Electron. Notes Theor. Comput. Sci. 148*, 2.

ACAR, U. A., BLELLOCH, G. E., BLUME, M., AND TANGWONGSAN, K. 2006b. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York.

ACAR, U. A., BLELLOCH, G. E., AND HARPER, R. 2006c. Adaptive functional programming. *ACM Trans. Program. Lang. Syst. 28*, 6, 990–1034.

ACAR, U. A., BLELLOCH, G. E., HARPER, R., VITTES, J. L., AND WOO, M. 2004. Dynamizing static algorithms with applications to dynamic trees and history independence. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York.

ACAR, U. A., BLELLOCH, G. E., AND HARPER, R. 2003. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

ACAR, U. A., BLELLOCH, G. E., AND HARPER, R. 2002. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*. 247–259.

AGARWAL, P. K., GUIBAS, L. J., EDELSBRUNNER, H., ERICKSON, J., ISARD, M., HAR-PELED, S., HERSHBERGER, J., JENSEN, C., KAVRAKI, L., KOEHL, P., LIN, M., MANOCHA, D., METAXAS, D., MIRTICH, B., MOUNT, D., MUTHUKRISHNAN, S., PAI, D., SACKS, E., SNOEYINK, J., SURI, S., AND WOLEFSON, O. 2002. Algorithmic issues in modeling motion. *ACM Comput. Surv. 34*, 4, 550–572.

ALEXANDRON, G., KAPLAN, H., AND SHARIR, M. 2005. Kinetic and dynamic data structures for convex hulls and upper envelopes. In *Proceedings of the 9th Workshop on Algorithms and Data Structures (WADS)*. Lecture Notes in Computer Science, vol. 3608, Springer, Berlin, 269—281.

ALSTRUP, S., HOLM, J., DE LICHTENBERG, K., AND THORUP, M. 1997. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming*, 270–280.

ALSTRUP, S., HOLM, J., DE LICHTENBERG, K., AND THORUP, M. 2003. Maintaining information in fully-dynamic trees with top trees. The Computing Research Repository (CoRR)[cs.DS/0310065].

BARBER, C. B., DOBKIN, D. P., AND HUHDANPAA, H. 1996. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw. 22*, 4, 469–483.

BASCH, J., GUIBAS, L. J., AND HERSHBERGER, J. 1999. Data structures for mobile data. *J. Algorithms 31*, 1, 1–28.

BELLMAN, R. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ.

BENDER, M. A., COLE, R., DEMAINE, E. D., FARACH-COLTON, M., AND ZITO, J. 2002. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th European Symposium on Algorithms (ESA 2002)*. Lecture Notes in Computer Science, vol. 2461, 219–223.

BENTLEY, J. L. AND SAXE, J. B. 1980. Decomposable searching problems I: Static-to-dynamic transformation. J. *Algorithms 1*, 4, 301–358.

BHATTACHARYA, B. K. AND SEN, S. 1997. On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm. *J. Algorithms 25*, 1, 177–193.

BOISSONNAT, J.-D. AND YVINEC, M. 1998. *Algorithmic Geometry*. Cambridge Press.

BRODAL, G. S. AND JACOB, R. 2002. Dynamic planar convex hull. In *Proceedings of the 43 rd Annual IEEE Symposium on Foundations of Computer Science*. 617–626.

CARLSSON, M. 2002. Monads for incremental computing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, 26–35.

CHAN, T. M. 1996. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Computat. Geometry 16*, 361–368.

CHAN, T. M. 1999. Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 92–99.

CHIANG, Y.-J. AND TAMASSIA, R. 1992. Dynamic algorithms in computational geometry. *Proc. IEEE 80*, 9, 1412–1434.

COHEN, R. F. AND TAMASSIA, R. 1991. Dynamic expression trees and their applications. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 52–61.

DEMERS, A., REPS, T., AND TEITELBAUM, T. 1981. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Language*s. ACM, New York, 105–116.

DIETZ, P. F. AND SLEATOR, D. D. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*. ACM, New York, 365–372.

DOYLE, J. 1987. A truth maintenance system. In *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann, Palo Alto, CA, 259–279.

EPPSTEIN, D., GALIL, Z., AND ITALIANO, G. F. 1999. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook*, CRC Press, Ch. 8.

EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND NISSENZWEIG, A. 1997. Sparsification—A technique for speeding up dynamic graph algorithms. *J. ACM 44*, 5, 669–696.

FIELD, J. 1991. Incremental reduction in the lambda calculus and related reduction systems. Ph.D. dissertation, Department of Computer Science, Cornell University.

FIELD, J. AND TEITELBAUM, T. 1990. Incremental reduction in the lambda calculus. In *Proceedings of the ACM Conference on LISP and Functional Programming*. ACM, New York, 307–322.

FREDERICKSON, G. N. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput. 14*, 781–798.

FREDERICKSON, G. N. 1997. A data structure for dynamically maintaining rooted trees. *J. Algorithms 24*, 1, 37–65.

GRAHAM, R. L. 1972. An efficient algorithm for determining the convex hull of a finete planar set. *Inform. Process. Lett. 1*, 132–133.

GUIBAS, L. 2004. Modeling motion. In *Handbook of Discrete and Computational Geometry*, 2nd ed., J. Goodman and J. O'Rourke, Eds. Chapman and Hall/CRC, 1117–1134.

GUIBAS, L. AND RUSSEL, D. 2004. An empirical comparison of techniques for updating Delaunay triangulations. In *Proceedings of the 20th Annual Symposium on Computational Geometry (SCG'04)*. ACM Press, New York, 170–179.

GUIBAS, L. J. 1998. Kinetic data structures: A state of the art report. In *Proceedings of the Third Workshop on the Algorithmic Foundations of Robotics (WAFR'98)*. 191–209.

HAMMER, M. A. AND ACAR, U. A. 2008. Memory management for self-adjusting computation. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*. 51–60.

HAMMER, M. A., ACAR, U. A., AND CHEN, Y. 2009. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York.

HENZINGER, M. R. AND KING, V. 1997. Maintaining minimum spanning trees in dynamic graphs. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*. Springer, Berlin, 594–604.

HENZINGER, M. R. AND KING, V. 1999. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM 46*, 4, 502–516.

HEYDON, A., LEVIN, R., AND YU, Y. 2000. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 311–320.

HOLM, J., DE LICHTENBERG, K., AND THORUP, M. 2001. Polylogarithmic deterministic fully- dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM 48*, 4, 723–760.

HOOVER, R. 1987. Incremental graph evaluation. Ph.D. dissertation, Department of Computer Science, Cornell University.

JONES, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York. (With a chapter on distributed garbage collection by R. Lins.)

KIRKPATRICK, D. G. AND SEIDEL, R. 1986. The ultimate planar convex hull algorithm. *SIAM J. Comput. 15*, 1, 287–299.

KNUTH, D. E. 1998. *The Art of Computer Programming: Sorting and Searching*, vol. 3, 2nd ed. Addison-Wesley, Ch. 6, 481–489.

LEY-WILD, R., ACAR, U. A., AND FLUET, M. 2009. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

LEY-WILD, R., FLUET, M., AND ACAR, U. A. 2008. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming*.

LIU, Y. A., STOLLER, S., AND TEITELBAUM, T. 1998. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst. 20*, 3, 546–585.

MCALLESTER, D. 1990. Truth maintenance. In *Proceedings of the 8th National Conference on Artificial Intelligence*. 1109–1116.

MCCARTHY, J. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds. North-Holland, Amsterdam, 33–70.

MICHIE, D. 1968. "Memo" functions and machine learning. *Nature 218*, 19–22.

MULMULEY, K. 1994. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ.

OVERMARS, M. H. AND VAN LEEUWEN, J. 1981. Maintenance of configurations in the plane. *J. Comput. Syst. Sci. 23*, 166–204.

PUGH, W. AND TEITELBAUM, T. 1989. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 315–328.

RADZIK, T. 1998. Implementation of dynamic trees with in-subtree operations. *ACM J. Exper. Algor. 3*, 9.

RAMALINGAM, G. AND REPS, T. 1993. A categorized bibliography on incremental computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 502–510.

REPS, T. 1982. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, 169–176.

RUSSEL, D. 2007. Kinetic data structures in practice. Ph.D. dissertation, Department of Computer Science, Stanford University.

RUSSEL, D., KARAVELAS, M. I., AND GUIBAS, L. J. 2007. A package for exact kinetic data structures and sweepline algorithms. *Comput. Geom. Theory Appl. 38*, 1-2, 111–127.

SAXE, J. B. AND BENTLEY, J. L. 1979. Transforming static data structures to dynamic structures (abridged version). In *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*. 148–168.

SHAMOS, M. I. 1978. Computational geometry. Ph.D. dissertation, Department of Computer Science, Yale University.

SHANKAR, A. AND BODIK, R. 2007. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York.

SLEATOR, D. D. AND TARJAN, R. E. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci. 26*, 3, 362–391.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *J. ACM 32*, 3, 652–686.

STALLMAN, R. M. AND SUSSMAN, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Art. Intell. 9*, 2, 135–196.

SUNDARESH, R. S. AND HUDAK, P. 1991. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 1–13.

TARJAN, R. AND WERNECK, R. 2005a. Dynamic trees in practice. In *Proceeding of the 6th Workshop on Experimental Algorithms (WEA'07)*. 80-93.

TARJAN, R. AND WERNECK, R. 2005b. Self-adjusting top trees. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*.

TARJAN, R. E. 1997. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Mathemat. Program. 78*, 167–177.

WEGMAN, M. N. AND CARTER, L. 1979. New classes and applications of hash functions. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*. 175–182.

WENGER, R. 1997. Randomized quickhull. *Algorithmica 17*, 3, 322–329.

YELLIN, D. M. AND STROM, R. E. 1991. INC: A language for incremental computations. *ACM Trans. Program. Lang. Syst. 13*, 2, 211–236.

Received November 2007; revised March 2009; accepted April 2009