Traceable Data Types for Self-Adjusting Computation

Umut A. Acar [*]
Max-Planck Institute
for Software Systems
umut@mpi-sws.org

Guy Blelloch Ruy Ley-Wild [†]
Kanat Tangwongsan
Carnegie Mellon University
{blelloch,rleywild,ktangwon}@cs.cmu.edu

Duru Türkoğlu [‡]
University of Chicago
duru@cs.uchicago.edu

Abstract

Self-adjusting computation provides an evaluation model where computations can respond automatically to modifications to their data by using a mechanism for propagating modifications through the computation. Current approaches to self-adjusting computation guarantee correctness by recording dependencies in a trace at the granularity of individual memory operations. Tracing at the granularity of memory operations, however, has some limitations: it can be asymptotically inefficient (e.g., compared to optimal solutions) because it cannot take advantage of problem-specific structure, it requires keeping a large computation trace (often proportional to the runtime of the program on the current input), and it introduces moderately large constant factors in practice.

In this paper, we extend dependence-tracing to work at the granularity of the query and update operations of arbitrary (abstract) data types, instead of just reads and writes on memory cells. This can significantly reduce the number of dependencies that need to be kept in the trace and followed during an update. We define an interface for supporting a traceable version of a data type, which reports the earliest query that depends on (is changed by) revising operations back in time, and implement several such structures, including priority queues, queues, dictionaries, and counters. We develop a semantics for tracing, extend an existing self-adjusting language, ΔML, and its implementation to support traceable data types, and present an experimental evaluation by considering a number of benchmarks. Our experiments show dramatic improvements on space and time, sometimes by as much as two orders of magnitude.

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

General Terms Languages

Keywords self-adjusting computation, traceable data types

[*] Acar is partially supported by gifts from Intel, Microsoft Research, and Jane Street Capital.

[†] Ley-Wild is partially supported by a Bell Labs Graduate Fellowship.

[‡] Türkoğlu is partially supported by gifts from Intel and Jane Street Capital.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019-3/10/06. . . $10.00

1. Introduction

Many applications must process data that changes, sometimes continuously, over time possibly with small changes at each step. For example, a traffic controller needs to update a traffic map after an accident blocks a road segment, a robot may need to update its motion plan after encountering a new obstacle, a theorem prover may need to update its conclusions after discovering a new fact, or a blood-flow simulator must compute properties of molecules that move continuously over time. In these and similar applications, small or continuous changes to data often cause only small updates to the output, making it possible to respond to dynamically-changing data more efficiently than re-computing the output from scratch, often asymptotically. To exploit this potential, one can develop so called "dynamic" of "kinetic" algorithms that are optimized to deal with particular forms of changing input. Indeed, there has been significant progress on such algorithms (e.g., [7, 11, 14]). Designing and implementing these algorithms turns out to be quite difficult even for problems that are simple in the absence of data changes, such as many of the simple graph and computational geometry algorithms. All too often, when such algorithms exist, they are quite complex and difficult to implement.

Alternatively, the programming languages community has developed techniques that automate or mostly automate the process of translating an implementation of an algorithm for fixed input into a version for changing input (e.g., [12, 15, 23]) by storing certain trace information during the computation. A recent approach based on a combination of dynamic dependence graphs [2] and memoization has been used to develop asymptotically efficient versions of a reasonably broad set of problems [5, 25]. This approach, called self-adjusting computation (SAC), generates a computation dependence graph while running the program on the initial data, and stores information at each node of the graph representing the code that needs to be rerun if its input changes. A change-propagation algorithm propagates any input changes through this graph, updating the parts of the graph and the output that depend on them. Memoization allows the algorithm to reuse portions of the graph during propagation. The time taken by change-propagation depends on how stable the computation trace is with respect to changes in input [20]. Recent work shows that self-adjusting computation and its variants can be supported by extending existing languages, such as C [16], Java [25], and Standard ML [19].

To achieve automatic and correct updates under data modifications, existing self-adjusting computation techniques trace dependencies at the (memory) cell level by recording memory operations. Although very flexible, this fine-grained approach to tracing has some performance problems, both in terms of time and space. First, it creates a considerable time overhead, slowing down essentially every memory operation. Second, it requires significant memory space for storing fine-grained dependence information. Third, when implementing data types using the approach, updates can cause many changes internal to the data type implementation, even when

the changes that propagate to the interface are small—i.e., the computation can be stable with respect to operations of the data type, but not stable with respect to individual cell accesses—ultimately causing sub-optimal updates. An example of this third problem is in maintaining a priority queue, where inserting a single element can require linear time for change-propagation at the cell level even if it only creates a single change (an additional insert operation) at the interface level. Many algorithms that use a priority queue will suffer from this problem.

In this paper, we extend the tracing of dependencies to support the query and update operations of arbitrary (abstract) data types, instead of just the reads and writes of a cell. For many applications, this asymptotically reduces the number of dependencies that are traced, reducing memory and time overhead, and for some it can speed up change propagation dramatically by making them more stable. This extension involves developing *traceable* versions of any data type that needs its dependencies to be traced directly, and adapting the change-propagation algorithm to handle the more general dependence tracing. The change propagation algorithm itself remains insensitive to the specifics of the data structures, which we achieve by providing a unified interface for all traceable data types. From the perspective of a user who is implementing self-adjusting programs, the changes to the code are minimal: all this requires is to substitute a different library or implementation for the data type. In addition to improving performance, the approach can also greatly simplify the analysis of stability since the user need only consider the operations on the data type instead of all the memory accesses inside of it. In Section 2, we explain the problem of tracing dependencies at the memory cell level in more detail, give an overview of our approach, and present algorithms that use traceable data types.

We define a relatively simple interface that one must implement to support a *traceable data type* (TDT) (Section 3). It is based on maintaining an operation trace for each instance and allowing any operation of the standard data type to be invoked or revoked anywhere in the trace. Such revisions must return a pointer to the earliest following query in the trace made inconsistent by the revision, if any. Implementations of TDTs do not need to be aware of the change-propagation algorithm beyond the interface. We have implemented traceable versions of several data types, including queues, priority queues, dictionaries, and accumulators (Section 3.1). We note that we do not expect that new data types would be implemented very often.

To support traceable data types uniformly, we modify change propagation in some relatively small but subtle ways. First, instead of storing closures with just the read operations on each cell, we store a closure for each query on TDTs. In fact, we treat cells as a TDT instance with read and write operations, and refer to them as *modifiable references*. Second, as with standard change propagation, during execution we keep a time-ordered priority queue of inconsistent queries (reads), but instead of tracking all inconsistencies for all TDT instances, we only keep the earliest inconsistency for each instance. This is critical for efficiently handling certain data types (Section 3). We present a formal self-adjusting core calculus that is extensible by arbitrary TDTs (Section 4). We present a static and dynamic semantics for the core calculus, including change propagation for traceable data types. A key component of the calculus is the open-endedness of the tracing and change-propagation semantics to support arbitrary traceable data types, without knowledge of how they are implemented.

We assess the effectiveness of the approach by extending the ΔML language [19, 20] to support TDTs, and implementing a number of benchmarks, including heap sort, Dijkstra's shortest path algorithm, breadth-first search on graphs, Huffman coding, and interval stabbing (Section 6). An interesting property of TDTs is that they enable operating efficiently on certain continuously varying values such as "time". Taking advantage of this property, we implement a library for algorithmic motion simulation that enables performing motion simulation with self-adjusting programs by appropriately changing the "time" (Section 7). Specifically, we consider an algorithm for computing convex hulls in 3D making it possible to safely implement motion simulation without requiring unsafe manipulation of the internals of the run-time system to ensure efficiency as in previous work [4].

Using these benchmarks, we perform an experimental evaluation of the proposed approach (Section 6). The experiments show substantial time and space improvements. Even on moderate input sizes, the improvements range between a factor of 3 and 20 reduction in from-scratch running time, between a factor of 4 and 50 reduction in space, and between a factor of 4 and $5,000$ reduction in update time compared to the version using only modifiable references (memory cells).

2. Overview

We motivate traceable data types, overview their structure, and consider some examples in the ΔML language.

2.1 Motivation

Consider a priority queue whose signature is shown in Figure 1. The priority queue provides a `new` operation that takes a comparison function on keys returns an empty priority queue, an `insert` function for inserting a key and a value into a priority queue, and a `delMin` function for removing the element with the minimum priority. Consider a program that uses this priority queue data structure. Using existing self-adjusting computation techniques (e.g., [16, 20]), we can write a self-adjusting version of priority queues and the program.

A self-adjusting program responds interactively to modifications to its input data. To achieve this, as the program executes, the run-time system constructs a trace of the execution. A *change-propagation* algorithm uses this trace to update the output when the input is modified. In existing self-adjusting computation techniques, the trace will be constructed by recording operations on so called *modifiable references* (*modifiables* for short) that hold *changeable* data, i.e., data that can change over time. For example, to make a heap data structure (a priority queue) self-adjusting, the child pointers of each heap node can be replaced by modifiables. Languages such as ΔML [19, 20] make this transformation relatively straightforward.

By placing data in modifiables, it is possible for a program to respond to input changes efficiently. Although relatively straightforward, this approach suffers from several limitations. First, the execution is traced at a fine granularity constructing a relatively large trace, typically asymptotically as large as the running time of the program. Second, by tracing every operation on a modifiable, the program is slowed down significantly by introducing reasonably large overheads. Third, our response times could suffer because change-propagation can spend significant time maintaining the fine-grain dependence information recorded in the trace.

As a concrete example, consider a self-adjusting version of a heap data structure where each child pointer is placed in a modifiable. The trace would record every access to the child pointer, contributing significantly to the size of the program trace. Since the work performed after each child access is essentially a comparison between priorities (keys) and thus relatively small, tracing would slow down the computation significantly. Interestingly, dependence-tracing at the level of modifiables can also result in sub-optimal update performance with change-propagation. To see this we will need to look further into the structure of the trace. As an example, we consider a worst-case scenario.

Consider a self-adjusting program P that takes as input an integer list, creates an empty queue and inserts the first element of

```
signature PRIORITY_QUEUE = sig
  type ('k,'v) t
  val new: ('k * 'k -> order) -> ('k,'v) t
  val insert: ('k,'v) t * 'k * 'v -> unit

  val delMin: ('k,'v) t -> ('k * 'v) option

end
```

Figure 1. The signature for priority queues.

```
signature PRIORITY_QUEUE_TRACEABLE = sig
  type ('k,'v) t
  val new: ('k * 'k -> order) -> ('k,'v) t
  val invoke_insert: ts * (('k,'v) t * 'k * 'v) -> ts option * unit
  val revoke_insert: ts -> ts option
  val invoke_delMin: ts * ('k,'v) t -> ts option * ('k * 'v) option
  val revoke_delMin: ts -> ts option
end
```

Figure 2. The signature for traceable priority queues.

$$[\texttt{new}\,() \Rightarrow Q]\ [a \to Q]\ [1 \to Q]\ \langle 1, a\rangle\ [Q \to 1]\ [2 \to Q]\ \langle 2, a\rangle\ [Q \to 2]\ [3 \to Q]\ \langle 3, a\rangle\ [Q \to 3]\ \cdots [n \to Q]\ \langle n, a\rangle\ [Q \to n]$$
$$\checkmark \quad \times \quad \checkmark \quad \times \quad \checkmark \quad \checkmark \quad \times \quad \checkmark \quad \checkmark \quad \times \quad \checkmark \quad \cdots \quad \checkmark \quad \times \quad \checkmark$$
$$[\texttt{new}\,() \Rightarrow Q]\ [b \to Q]\ [1 \to Q]\ \langle 1, b\rangle\ [Q \to 1]\ [2 \to Q]\ \langle 2, b\rangle\ [Q \to 2]\ [3 \to Q]\ \langle 3, b\rangle\ [Q \to 3]\ \cdots [n \to Q]\ \langle n, b\rangle\ [Q \to n]$$

$$[\texttt{new}\,() \Rightarrow Q]\ [a \to Q]\ [1 \to Q]\ [Q \to 1]\ [2 \to Q]\ [Q \to 2]\ [3 \to Q]\ [Q \to 3]\ \cdots [n \to Q]\ [Q \to n]$$
$$\checkmark \quad \times \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \cdots \quad \checkmark \quad \checkmark$$
$$[\texttt{new}\,() \Rightarrow Q]\ [b \to Q]\ [1 \to Q]\ [Q \to 1]\ [2 \to Q]\ [Q \to 2]\ [3 \to Q]\ [Q \to 3]\ \cdots [n \to Q]\ [Q \to n]$$

Figure 3. Two pairs of traces of a hypothetical program P at the level of queue operations and comparisons (top) and at the level of abstract queue operations (bottom). Each pair corresponds to a run of P with inputs $[a, 1, 2, \ldots, n]$ and $[b, 1, 2, \ldots, n]$.

the list into the priority queue. Starting with the second element, the program then inserts each element into the priority queue using the element both as a priority and as a value and removes the minimum element by performing a `delMin`. Assume that the priority queue is implemented via conventional self-adjusting computation techniques requiring the tracing of every comparison. Figure 3 (top) shows the traces (represented abstractly) for an execution of P with the input $[a, 1, 2, \ldots, n]$ and $[b, 1, 2, \ldots, n]$, where $a, b > n$ and $a \neq b$. We write $[i \to Q]$ for an instance of the operation $\texttt{insert}(Q, i, i)$, $[Q \to i]$ for an instance of the operation $\texttt{delMin}(Q)$ that returns i as the minimum priority, and $\langle i, j\rangle$ for a comparison of the keys i and j. Now comparing the two traces, note that every comparison in the first trace has the form $\langle i, a\rangle$ and every comparison in the second trace has the form $\langle i, b\rangle$ $(1 \leq i \leq n)$ and no two comparisons match—the difference between the two traces is $\Theta(n)$. In the figure, we use ✓ and ✗ to indicate the operations of the trace that match and that do not match (respectively). Consequently, starting with the first input running the program P, changing the input by replacing a by b, and performing change-propagation would require at least linear time to update the output (a more precise account of the relationship between the trace distances and change-propagation can be found elsewhere [20]).

This argument extends to any priority queue data structure, because every time a new key i is inserted, the priority queue contains only the element with the largest key, either a or b and thus a comparison with i must be performed to determine the minimum priority required by the next operation. It is thus not possible to use change-propagation based on conventional self-adjusting computation to update the output in less than linear time. Fortunately, there is great potential for improvement. To see this suppose that we record just the priority queue operations in the trace and not the comparisons. As shown in Figure 3 (bottom), the traces of the two runs of P are very similar; they differ in only one operation.

The example shows that if we can record dependencies at the level of priority queue operations instead of the internal comparisons performed by the priority queue operations, then the trace is smaller and there are fewer differences between the computations, and thus change-propagation can be performed more efficiently. This is the main idea behind traceable data types. As we discuss in Section 6 both the improvements in the size of the trace and the update time can be asymptotic. Challenges to realizing traceable data types include the question of whether it is possible to design and implement them efficiently, whether they can be made to work with change-propagation so that programs can still respond automatically to modifications to their data, and whether they can be supported naturally without requiring a cumbersome interface.

2.2 Traceable Data Types

A traceable data type (TDT) permits tracing dependencies at the level of its operations rather than memory cells. For an abstract data type, a traceable version of the data type provides an analogous operation to initialize the data type and two versions, called *invoke* and *revoke*, for each of the remaining operations. These operations essentially allow revisions to the sequence of operations performed on a data structure by inserting new operations (via invoke) and deleting existing operations (via revoke). To enable efficient revisions, TDTs maintain an internal *operation trace* of the operations performed labelled with their timestamp (of type `ts`).

If the abstract data type has an operation `op:` α `->` β, the traceable version has the operations

```
invoke_op: ts * α -> ts option * β
revoke_op: ts -> ts option
```

These operations revise the operation trace by inserting a new `op` operation at a given timestamp (`invoke_op`) and by removing an `op` operation at a given timestamp from the operation trace (`revoke_op`). Both the invoke and revoke operations return an optional timestamp corresponding to the next operation, if any, that has been invalidated by the revision. As an example, Figure 2 shows the signature for the traceable priority queue.

To enable change-propagation, invoke and revoke operations identify the first operation of the trace made inconsistent by the revision by returning the timestamp for that operation. We call an operation *inconsistent* if its return value changes after the revision. Suppose for example that we perform the operations

`insert(pq,3,3)`, `insert(pq,2,2)`, `insert(pq,1,1)`, `delMin(pq)`, `delMin(pq)`, `delMin(pq)`.

The `delMin` operations will return the values $1, 2, 3$ in sorted order. If we now `revoke_insert (pq,1)`, the first `delMin` operation will be the earliest affected operation and thus its timestamp be returned by this revision. Note that in fact all other `delMin` operations are inconsistent. In Section 3 we define traceable data types more precisely, consider several examples, and describe how they can be implemented efficiently. In Section 4 we present an extensible semantics for integrating TDTs into a self-adjusting language including change-propagation.

The proposed interface with invoke and revoke operations are significantly more cumbersome to use than the standard data types. Fortunately, these operations need not be used by the programmer at all. In fact, it is possible to present a "user-level" interface for traceable data types that is essentially the same as the standard version. We describe how to achieve this in the context of the ΔML language.

2.3 ΔML with Traceable Data Types

The ΔML language extends Standard ML (SML) with support for self-adjusting computation. The principal extensions to SML are *modifiable references* which are ML-style references with support for dependence-tracing, *adaptive functions* that help identify opportunities for computation reuse, and a *change-propagation* mechanism for updating computations and outputs. In ΔML, after a self-adjusting program executes, the contents of the input modifiables may be modified and the output can be updated by calling change-propagation. To support efficient compilation and updates, ΔML offers two kinds of function spaces: conventional functions of type α->β and adaptive functions of type α-$>$>β. Application of adaptive function f to argument a is written f a. Adaptive functions, defined by keywords **afun** and **mfun**, can be either non-memoized or memoized (respectively), and can call conventional function as well as adaptive functions. Conventional functions are not permitted to call adaptive functions.[1]

Consider a program that uses some (standard) data type, e.g., a priority queue, and suppose that we have a traceable version of that data type. To enable the user to operate on the traceable data types in the same way as the standard data types, we provide a *user-level* interface to the data type that essentially matches the standard interface with the exception of requiring each operation to be an adaptive function. More specifically, each operation of type α->β becomes α-$>$>β. For example, the user-level interface for the traceable priority queue would be:

```
signature PRIORITY_QUEUE_TDT_USER = sig
  type ('k,'v) t
  val new: ('k * 'k -> order) -$> ('k,'v) t
  val insert: ('k,'v) t * 'k * 'v -$> unit
  val delMin: ('k,'v) t -$> ('k * 'v) option
end
```

Given a program using the user-level interface to traceable data types, our (extended) ΔML compiler can translate the program to use the corresponding invoke and revoke operations and integrate them with change-propagation. To this end, the compiler generates the necessary code for tracing the invoke and revoke operations and for finding and re-executing them when necessary during change-propagation. For example, we can compile some program that uses the above interface to traceable priority queues shown in Figure 2.

[1] Each self-adjusting program has a single entry point which itself is an adaptive function.

```
structure PQ : PRIORITY_QUEUE_TDT_USER
afun heapsort (compare, l) =
let
  val heap = PQ.new $ compare
  afun insert x = PQ.insert $ (heap, x, ())
  mfun loop m =
    case m of
      NONE => NIL
    | SOME (k, ()) =>
        let val t = loop $ (PQ.deleteMin $ heap)
        in CONS(k, put $ t) end
in
  (List.app insert $ l;
   put $ (loop $ (PQ.deleteMin $ heap)))
end
```

Figure 4. Code for heap sort in ΔML.

2.4 Example: Heap Sort

As an example of how a traceable data type can be used in a user program, we consider a ΔML implementation of heap sort as shown in Figure 4. The algorithm first allocates an empty priority queue and inserts all the keys in its input to the priority queue (with unit payload). It then constructs the sorted output by repeatedly removing the minimum element until the queue is empty and returning them in a list. This algorithm has several notable features: First, it has an optimal $O(n \log n)$ running time. Second, but most importantly, it is highly stable under small modifications to its input when the trace is at the granularity of priority queue operations. Thus, with traceable priority queues, we can obtain an efficient self-adjusting sorter (our experiments in Section 6 confirms that the algorithm performs well in practice).

Finally, the self-adjusting version in ΔML only differs from the standard SML implementation in the underlined code fragments (also highlighted in red) and the use of the user-level traceable priority queue. The only major differences are the use of modifiable lists where the tail of each cell is placed in a modifiable, and that the priority queue functions have the adaptive function type. We define `loop` as a memoized function as it performs non-constant work.

This example provides evidence that programming with TDTs requires little modifications to existing code. Note also that instead of using a traceable priority queue, we could also use a self-adjusting version of a priority queue that has the same interface (e.g., a heap or a treap implemented using modifiables). As discussed in Section 2.1 and further in Section 6, such modifiable-based implementations, however, perform significantly worse.

2.5 Example: Dijkstra's Algorithm

As another example, we consider Dijkstra's algorithm for computing single-source shortest-paths, whose ΔML code is shown in Figure 5. The code strongly resembles the SML implementation: dropping the underlined text yields the SML code. We omit some details to focus attention on the aspects relevant to our interest here. Dijkstra's algorithm takes a graph and a root node and finds the shortest-path distance from the root to every node in the graph. We represent the input graph as a dictionary of nodes (`t_graph`), mapping each node to the list of its neighbors along with the edge weights. Similarly, we represent the output as a dictionary of nodes (`dict_sp`), mapping each node to its distance to the root. The key idea in the algorithm is to maintain a set of explored vertices and their distances to the root and expand this set iteratively. For this purpose, we maintain a priority queue (`pq_v`) of visited vertices and their current distances. The algorithm starts by inserting the root into the priority queue with distance 0. It then repeatedly visits the vertices in the order of their current distance by calling the function `loop`. Given

```
structure Dict : DICTIONARY = struct ... end
structure PQ : PRIORITY_QUEUE_TDT_USER
structure List : LIST = struct ... end
type t_node = ...
type t_dist = ...
type t_graph = (t_node, (t_node * t_dist) List.t) Dict.t
afun dijkstra (root: t_node, graph: t_graph) =
let
  val dict_sp: (t_node, t_dist) Dict.t = Dict.new $ ()
  val pq_v: (t_dist * t_node) PQ.t = PQ.new $ ()
  afun visit (u, d:t_dist) =
  let afun ins (v,w) = PQ.insert $ (pq_v, (d + w, v))
  in case Dict.lookup $ (graph, u) of
       NONE => ()
     | SOME ns => List.app ins $ ns
  end

  mfun loop (u:t_node, d:t_dist) =
  (if (Dict.lookup $ (dict_sp, u)) = NONE then
    (Dict.insert $ (dict_sp, u, d); visit (u,d))
   else ();
   case PQ.deleteMin $ pq_v of
     NONE => dict_sp
   | SOME (d, v) => loop $ (v,d))
in loop $ (root, 0) end
```

Figure 5. Code for Dijkstra's algorithm in ΔML.

a vertex u and its current distance `d`, the function `loop` checks if u is already visited. If so then it continues by removing the next vertex from the priority queue. If not then the exact distance for u is found; it inserts u into the output (`dict_sp`) and visits it. To visit a vertex, it traverses each outgoing edge (u, v) by inserting v into the priority queue with an updated distance. As with the heap sort algorithm, we are able to obtain a self-adjusting version of the algorithm without modifying its structure, and we can use both a traceable priority queue or a self-adjusting priority queue implemented by using modifiables directly (Section 6 compares these implementations).

3. Traceable Data Types

We define an (conventional) abstract data type D as a quadruple (τ **tdt**,S,`mk`, {`op_1`, ..., `op_n`}) consisting of

- a type constructor τ **tdt**,
- a state constructor S,
- a creation (make) operation `mk`, and
- a set of operations `op_`i ($1 \leq i \leq n$).

A *specification* of a data type is a set of state-transformation relations, one for each constructor or operation. The state-transformation relation for the constructor maps a given value v to an initial state S_0, written $v \xrightarrow{\texttt{mk}} S_0$. The state-transformation relations for the other operations map a state and a given value to another state and another value, e.g., $S; v \xrightarrow{\texttt{op\_}i} S'; v'$ defines how `op_`i operation with argument v transforms the state from S to S' and yields result v'. Any data type can, however complex, be specified in this way by coming up with an appropriate representation for the state and by specifying the state-transformation function.

To define the traceable version of a data type D=(τ **tdt**,S,`mk`, {`op_1`, ..., `op_n`}), we let T denote a totally ordered set of timestamps. We define a *operation-trace* H for a data type as an initial state S_0 and a sequence $[(t_1, o_1, v_1), (t_2, o_2, v_2), \ldots, (t_n, o_n, v_n)]$, where the $t_i \in T, t_i < t_{i+1}$, and each o_i is an operation of the form `op_`$k\ v_i'$ that takes some v_i' as an argument to return v_i. Let $v_D(H, t)$ be the value returned by performing the sequence of operations $(o_1, o_2, \ldots)$ in H up to time t, inclusive. We say that an element $(t_i, o_i, v_i) \in H$ of the operation-trace is *inconsistent* if $v_D(H, t_i) \neq v_i$. We say that an operation-trace is inconsistent if any element is inconsistent and consistent otherwise.

operation: type	state-transformation
put : $\tau \rightarrow \tau$ **modref**	$v \xrightarrow{put}$ *modref* v
get : **unit** $\rightarrow \tau$	*modref* v; () $\xrightarrow{get}$ *modref* v; v
set : $\tau \rightarrow$ **unit**	*modref* v; $v' \xrightarrow{set}$ *modref* v'; ()
mod : τ **cmp** $\times \tau \rightarrow \tau$ **mod**	$(v_c, v) \xrightarrow{mod}$ *mod* (v_c, v)
mget : (τ, τ') **dis** $\rightarrow \tau'$	*mod* (v_c, v); $v_d \xrightarrow{mget}$ *mod* (v_c, v); v'
pq : **unit** $\rightarrow (\tau_k, \tau_v)$ **pq**	() $\xrightarrow{pq}$ *pq* $\langle\rangle$
ins : $\tau_k \times \tau_v \rightarrow$ **unit**	*pq PQ*; $(v_k, v_v) \xrightarrow{ins}$ *pq PQ* + (v_k, v_v); ()
min : **unit** $\rightarrow \tau_k \times \tau_v$	*pq PQ*; () $\xrightarrow{min}$ *pq PQ* − (v_k, v_v); (v_k, v_v)

Table 1. Formal specification of modifiables, modular modifiables, and priority queues.

For a data type D, the *traceable version* D_r abstractly maintains an operation-trace H for each instance and provides the following operations:

- `mk`(v) : Returns a new operation-trace H with initial state S_0 (where $v \xrightarrow{\texttt{mk}} S_0$) and empty operation sequence.
- `invoke`(H, o, t) : Computes $v = v_D(H, t)$, updates the operation-trace H by inserting (t, o, v), and returns v and the time of the earliest inconsistent operation.
- `revoke`(H, t) : Removes the element with time t from H (if any) and returns the time of the earliest inconsistent operation.

We refer to invoke and revoke (meta-)operations as *revisions* and require them to be applied as part of a *revision sequence*—a sequence of revisions on an initially consistent operation-trace such that (1) the times of the revisions are increasing, and (2) for each revision at time t, all operation at times before t are consistent.[2]

It may seem odd that revisions only return the earliest inconsistent operation as opposed to all of them. In fact, this suffices because revision sequences require that the earliest inconsistency is fixed (revoked and possibly reinvoked) before proceeding to the next one. Fixing the first inconsistency will then return the next inconsistent operation, if any. This ability to return inconsistent operations lazily is critical for efficiency because otherwise we would have to maintain a potentially large sequence of inconsistent operations as some become consistent or others become inconsistent, and we would not be able to take advantage of subsequent revisions fixing inconsistencies. For example imagine invoking an additional `insert` operation on a priority queue inserting an element with higher priority than all the others. This will cause all the rest of operations to become inconsistent. Invoking another `deleteMin` operation subsequently, however, would make all operations consistent by removing the newly inserted element.

As we formalize in Section 4, a traceable data type can be used to support the underlying data type in self-adjusting computation. This allows a modular way to use new data types without having to know anything about the change-propagation algorithm itself.

3.1 Examples

We describe the interface of several TDTs. Sample formal specifications of abstract data type are given in Table 1.

Modifiables. A modifiable provides the functionality an ML-style reference with type constructor τ **modref** and state constructor *modref* v where v is a value of type τ. This is what we have

[2] Multiple revision sequences can be applied to an operation-trace sequentially, each returning the operation-trace to a consistent state before the next starts.

informally referred to as memory cells. Modifiable commands include the creation operation $\mathbf{put}$ and manipulation operations $\mathbf{get}$ for dereference and $\mathbf{set}$ for update. Table 1 shows the signature types and state-transformations. Intuitively, creating a modifiable with contents v and then dereferencing the modifiable multiple times yields an operation-trace with initial state *modref* v and operation sequence $[(t_1, \mathbf{get}\,(), v), \ldots, (t_n, \mathbf{get}\,(), v)]$. If we change the initial value to v' then the initial state becomes *modref* v' and the timestamp t_1 identifies the earliest inconsistent operation. Change propagation can successively reinvoke each revision to obtain the consistent sequence $[(t_1, \mathbf{get}\,(), v'), \ldots, (t_n, \mathbf{get}\,(), v')]$.

Modular Modifiables. In some applications the domain of data may be continuous even when the computation produces a discrete result, e.g., a program computing the convex hull of a set of moving points represented combinatorially. In such a case, using modifiables makes change-propagation sensitive to any change forcing recomputation even if the result is the same. In many of these cases, we partition the continuous domain into some discrete number of sets and consider values equal if they fall into the same set. For example, we may care only about the sign of a real number. Our motion simulation benchmarks make critical use of modular modifiables for storing the time variable.

A *modular modifiable* allows discretizing a totally-ordered continuous set to avoid recomputation when modifications don't affect the discrete outcome. The type of modular modifiables is $\tau\,\mathbf{mod}$ and the state constructor is *acc* (v_c, v), where v_c is a comparison function of type $\tau\,\mathbf{cmp}(= \tau \times \tau \rightarrow \mathbf{order})$ (where $\mathbf{order}$ is the SML order datatype) and v is the value of the modifiable. Modular modifiables are created by the $\mathbf{mod}$ operation and manipulated by the modular dereferencing operation $\mathbf{mget}$: A modular dereference takes a *discretization* argument v_d of type $(\tau, \tau')\,\mathbf{dis}$ which is a (finite) partition of the continuous type τ together with an assignment of values from the discrete type τ' to each equivalence class. Formally, the discretization is represented by a list $[c_1, \ldots, c_n]$ that partitions τ into intervals and the assignment is a list $[d_0, \ldots, d_n]$ of τ' elements. The result of such a dereference is $v' = d_i$ where the current value of the modular modifiable is $c_i \leq v < c_{i+1}$. Due to the structure of the partition, the outcome of a modular dereference only changes when the value of the modular modifiable changes equivalence classes.

Priority Queues. A *priority queue* with τ_k priorities and τ_v values has type $(\tau_k, \tau_v)\,\mathbf{pq}$. The state constructor is *pq PQ* where *PQ* is a sequence of pairs $\langle (v_{ki}, v_{vi}) \rangle$ where entry v_{vi} has priority v_{ki}. Priority queue commands include the creation operation $\mathbf{pq}$ and manipulation operation $\mathbf{ins}$ for inserting an element v_v with priority v_k and $\mathbf{min}$ for deleting the element with lowest priority: where $PQ + (v_k, v_v)$ adds the element v_v with priority v_k, and $PQ - (v_k, v_v)$ removes the element v_v with highest priority v_k.

Accumulator Modifiables. An *accumulator modifiable* provides efficient change-propagation for adding elements from a commutative group and querying the total. The query must come after all updates. Adding to a (regular) modifiable-based accumulator involves fetching the current value of the accumulator and storing the updated sum, which makes the operation sensitive to the current partial sum and thus change-propagation may take linear time in the number of additions. An accumulator modifiable provides a primitive addition operation that is not sensitive to the intermediate sums and can change-propagate in constant time by using the group's inverse operation to update the result of querying a total.

Queues and Dictionaries. In addition to the above examples, traceable versions of many other data structures can be specified by giving their state-transformations functions. We have also formulated and implemented traceable first-in-first-out queues and unordered dictionaries.

3.2 Implementing Traceable Data Types

We briefly describe how to implement the traceable version of the data types described in the previous subsection. In the context of this paper, these descriptions indicate how to implement functions for signatures such as one in Figure 2. A more complete description of the data types and how to implement others can be found elsewhere [3]. The basic idea behind the implementations is to keep an augmented version of the operation trace. In particular, most of our structures maintain a data structure for the trace that is ordered by timestamps and supports $\mathsf{insert}(T, v, t)$ (insert v at time t), $\mathsf{delete}(T, t)$ (delete the element at time t), $\mathsf{findPrev}(T, t)$ (returns the greatest element in T that is less that t) and $\mathsf{findNext}(T, t)$. For a trace with n entries, all these can be implemented in $O(\log n)$ time using balanced trees. Some of our traceable data types also maintain balanced trees ordered by keys (e.g., the priority queue, and modular modifiables).

We first consider the traceable implementation of a modifiable (a read/write cell). Our implementation maintains a time-ordered sequence of operations. Each operation is tagged with the value it has read or written. To invoke a $\mathbf{get}$ (read) or $\mathbf{put}$ (write) at time t, we insert the operation into the trace data structure at t. If the operation is a $\mathbf{get}$, then we also use $\mathsf{findPrev}(T, t)$ to access the value returned by the read—the previous element in the trace might either be a $\mathbf{get}$ or $\mathbf{put}$, but both types of operation are stored with values. Note that a revision sequence requires that all operations before time t are consistent; therefore, the value of this previous element contains the correct value for time t. To revoke a $\mathbf{get}$ or $\mathbf{put}$ at time t, we simply delete the operation from the trace. For all revisions (invokes or revokes) we can use $\mathsf{findNext}(T, t)$ to return the earliest inconsistent operation, if any. In particular, if the next operation is a $\mathbf{get}$ and has a different value, then it is inconsistent and is returned, otherwise nothing is returned. All operations on a trace with n elements take $O(\log n)$ time.

The implementation of dictionaries is based on modifiables as described in the previous paragraph. Basically, we create a standard hash table, where each entry in the table is a modifiable with its own trace. The first time an operation is invoked on a particular key k, we create a new modifiable for that key with its own trace—we refer to this as m_k. Any insert of a key-value pair (k, v) into the dictionary at time t will correspond to a $\mathbf{put}$ of value v into m_k at t. Any delete of a key k from the dictionary at time t will correspond to a $\mathbf{put}$ of value $\emptyset$ into m_k at t, where $\emptyset$ is a special value indicating that the dictionary has no entry at that key. Any search of a key k at time t corresponds to a $\mathbf{get}$ m_k at t. Finally, if a revoke of an operation on key k removes the last operation from the trace of m_k, then we can delete m_k from the dictionary (this avoids a memory leak).

The implementation of priority queues is beyond the scope of this paper, but we note that it can be done with two balanced trees one ordered by time for the trace and the other by key. In addition, during an update sequence, the implementation maintains two additional balanced trees, one for insertions invoked during the current update sequence and the other for insertions revoked during the sequence. All operations take $O(\log n)$ time. A modular modifiable is implemented by keeping all the boundary elements c_i for all $\mathbf{mget}$ operations on a modular modifiable m sorted by their ordering. We call this S_m. Invoking or revoking a $\mathbf{mget}$ operation on m corresponds to inserting or deleting the partitioning elements from S_m. Changing the initial value will identify all partitions that are crossed by the change of value and return the earliest as inconsistent. An accumulator modifiable is implemented simply by "adding" to the sum using the commutative operator on an invoke and subtracting from the sum on a revoke. For any value other than the identity, this will return the next read as the earliest inconsistent operation.

Finally, we use an order-maintenance data structure [13] to implement time stamps. Simpler alternatives such as using integers, fixed-precision floating-point numbers do not work because they do not allow insertions of new timestamps between two adjacent integers. Arbitrary precision real numbers would work but are not efficient.

4. The Tgt Language

The ΔML language (Section 2) is compiled into the Tgt language by the ΔML compiler. In this section we present the Tgt language to show how TDTs can be integrated orthogonally into a language with intrinsic support for self-adjusting computation.

The Tgt language provides both *evaluation* to reduce expressions to values and *change propagation* to adapt computations to input changes, and is open-ended to extension by any TDT. The semantics of the Tgt language uses *traces* to capture the structure of the computation, which are used by change-propagation to identify the need for recomputation and the opportunity for computation reuse. The former approach to self-adjusting computation used trace actions that correspond to individual memory operations. To support TDTs, the new approach to self-adjusting computation uses trace actions that correspond to high-level TDT operations. The invoke and revoke operations of TDTs are used by the semantics of the Tgt language to identify which parts of a computation, i.e., which actions of the trace, are affected by changes.

The Tgt language is a simply-typed λ-calculus with natural numbers and recursive functions[3], extended with a *memoization* primitive and any number of traceable data types (TDTs). The syntax of Tgt is given by the following grammar, which defines types τ, expressions e, values v, and adaptive commands κ, using identifier metavariables f and x.

$$\begin{array}{lcl} \tau & ::= & \mathbf{res} \mid \mathbf{nat} \mid \tau_x \to \tau \mid \tau\,\mathbf{tdt} \\ e & ::= & v \mid \mathbf{caseN}\, v_n\, e_z\, (x.e_s) \mid e_f\, v_x \\ v & ::= & x \mid \mathbf{zero} \mid \mathbf{succ}\, v \mid \mathbf{fun} f.x.e \mid \ell \mid \kappa \\ \kappa & ::= & \mathbf{halt}\, v \mid \mathbf{memo}\, e \mid \mathbf{mk}\, v_{mk}\, v_k \mid \mathbf{op}\, v_l\, v_{arg}\, v_k \end{array}$$

Tgt enforces a continuation-passing style (CPS) discipline to help identify opportunities for reuse and computations for re-execution.[4] The type $\mathbf{res}$ is an opaque answer type for continuations, while $\mathbf{halt}$ is a continuation that injects a final value into the $\mathbf{res}$ type. The CPS discipline allows pure computations (e.g., natural numbers and recursive functions) to be introduced by values and eliminated by expressions, with the $\mathbf{caseN}$ scrutinee and function application argument restricted to be values. The $\mathbf{caseN}$ primitive case-analyzes a natural number v_n and branches to e_z or e_s according to whether it is zero or a successor number. The $\mathbf{mk}$ and $\mathbf{op}$ primitives correspond to schematic TDT operations with an explicit continuation v_k. The $\mathbf{mk}$ primitive creates a TDT initialized by the seed value $v_m k$, while the $\mathbf{op}$ primitive takes the a reference v_l to a TDT and argument value v_{arg}.

Since adaptivity identifies the need for recomputation, Tgt programs use an indirection through the store to manipulate TDTs and isolate the differences between computations. We take a *store* σ to be a finite map from *locations* ℓ to TDT *state constructors* S; the notation $\sigma[\ell \mapsto S]$ denotes the store σ updated with ℓ mapped to S. *Contexts* Γ and Σ, and TDT *signatures* Δ are maps from variables, locations, and TDT commands to types, respectively.

The Tgt language is open-ended to extension by any number of TDTs. As described in Section 3, a TDT is classified by a type $\tau\,\mathbf{tdt}$ and has a state constructor S. Furthermore, each TDT extends the language with a creation command $\mathbf{mk}\ v_{mk}\ v_k$ and any number of manipulation (i.e., queries and updates) commands $\mathbf{op}\ v_l\ v_{arg}\ v_k$; TDT commands are formulated in CPS with an explicit continuation v_k identifying the computation that follows the command and manipulation commands take a location argument v_l.

The typing judgement $\Sigma; \Gamma \vdash e : \tau$ (rules elided) ascribes the type τ to the expression e in the contexts Γ and Σ. TDT commands have type $\mathbf{res}$ if their arguments match the types prescribed by the TDT signature. A creation command $\mathbf{mk}$ must have an argument v_{mk} of type τ_{mk} and a continuation v_k expecting a $\tau\,\mathbf{tdt}$. A manipulation command $\mathbf{op}$ must have a location argument v_l of type $\tau\,\mathbf{tdt}$, an argument v_{arg} of type τ_{arg}, and its continuation v_k should expect a τ_{res}.

Figure 6 gives the evaluation semantics of Tgt. The large-step evaluation relation $\dot{T}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'$ (resp. $\dot{T}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'$) reduces the expression e (resp. the adaptive command κ) under the store σ to the value v' and the updated store σ'. For the present time, we suggest that the reader ignore the $\dot{T}$ and T' components; we discuss them in detail in Section 4.1. The auxiliary evaluation relation $e \Downarrow v'$ reduces an expression e to a value v'; such evaluation is pure and independent of the store.

A $\mathbf{mk}\ v_{mk}\ v_k$ creation command ($\mathbf{mk}$) generates a TDT state S' with seed v_{mk} according to the state-transformation semantics, extends the store σ with a fresh location ℓ bound to S', and delivers ℓ to the continuation v_k. An $\mathbf{op}\ \ell\ v_{arg}\ v_k$ manipulation command ($\mathbf{op}$) fetches the TDT state S from the store σ at ℓ, performs the corresponding state-transformation, updates the store with ℓ bound to the new state S', and delivers the result v_{res} to the continuation v_k. For the present time $v_{mk}; \dot{T} \overset{\mathbf{mk}\,\ell}{\Rightarrow} S'; \dot{T}'$ and $S; v_{arg}; \dot{T} \overset{\mathbf{op}\,\ell}{\Rightarrow} S'; v_{res}; \dot{T}'$ (discussed in detail in Section 4.1) should be read as the state-transformation judgements $v_{mk} \overset{mk}{\to} S$ and $S; v_{arg} \overset{op}{\to} S'; v_{res}$. A memoized expression $\mathbf{memo}\ e$ simply evaluates the expression when evaluated from scratch ($\mathbf{memo/miss}$), but enables the reuse of computations *across runs* during change-propagation (Section 4.1). The $\mathbf{halt}\ v$ command yields a computation's final result value.

4.1 Change Propagation

In order to update a program's output in response to changes in its input, a *change-propagation* mechanism is employed to re-execute the portions of the computation affected by the changes and to reuse the unaffected portions. The evaluation relation records information necessary for change-propagation in a *trace* T, a sequence of TDT state and memo *actions* terminated by a halt action:

$$\begin{array}{lcl} A_s & ::= & \mathtt{mk}_{v_k}^{v_{mk}\uparrow\ell} \mid \mathtt{op}_{v_k\,\square}^{\ell, v_{arg}\downarrow v_{res}} \\ \square & ::= & \checkmark \mid \boldsymbol{\times} \\ A & ::= & A_s \mid \mathtt{memo}^{e} \\ T & ::= & \mathtt{halt}^{v} \mid A{\cdot}T \\ \dot{T} & ::= & \circ \mid T \end{array}$$

The evaluation relation $\dot{T}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'$ (resp. $\dot{T}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'$) may now be interpreted as reducing the expression e (resp. the command κ) under the store σ and the (optional) reuse trace $\dot{T}$, yielding the value v', the updated store σ', and the computation trace T' for the current run.

The evaluation of each command extends the computation trace with the corresponding trace action labeled by the relevant arguments and results. A halt action carries the final result value and a memo action carries the memoized expression. A creation action records the seed value, the location allocated, and the continuation. In order for the semantics to identify the possibility of computation reuse, each TDT manipulation action records the location accessed, the argument and result values, and the continuation; the action is additionally labeled by a *checkmark* $\square$ to indicate its replayability during change-propagation. Furthermore, the dynamic semantics maintains *consistency* of the reuse trace, i.e., the prefix trace

[3] The Tgt language may easily be extended with products, sums, recursive types, *etc.*; we have omitted such constructs as they provide no additional insight, but are supported by the implementation.

[4] Previous work shows how to compile a direct-style language into this continuation-passing style [19].

$$\frac{}{v \Downarrow v} \qquad \frac{e_z \Downarrow v}{\mathbf{caseN}\,\mathbf{zero}\,e_z\,(x.e_s) \Downarrow v} \qquad \frac{[v_n/x]e_s \Downarrow v}{\mathbf{caseN}\,(\mathbf{succ}\,v_n)\,e_z\,(x.e_s) \Downarrow v} \qquad \frac{e_f \Downarrow \mathbf{fun}\,f.x.e \qquad [v_x/x][\mathbf{fun}\,f.x.e/f]e \Downarrow v}{e_f\,v_x \Downarrow v}$$

$$\frac{e \Downarrow \kappa \qquad \dot{T};\sigma;\kappa \Downarrow_{\mathrm{K}} T';\sigma';v'}{\dot{T};\sigma;e \Downarrow_{\mathrm{E}} T';\sigma';v'} \qquad \frac{\ell \notin \mathrm{dom}\,\sigma \qquad v_{mk};\dot{T} \overset{\mathbf{mk}}{\Rightarrow}^{\ell} S';\dot{T}' \qquad \sigma_l = \sigma[\ell \mapsto S'] \qquad \dot{T}';\sigma_l;v_k\,\ell \Downarrow_{\mathrm{E}} T';\sigma';v'}{\dot{T};\sigma;\mathbf{mk}\,v_{mk}\,v_k \Downarrow_{\mathrm{K}} \mathtt{mk}_{v_k}^{v_{mk}\uparrow\ell}\cdot T';\sigma';v'}\ \mathbf{mk}$$

$$\frac{\sigma(\ell) = S \qquad S;v_{arg};\dot{T} \overset{\mathbf{op}}{\Rightarrow}^{\ell} S';v_{res};\dot{T}' \qquad \sigma_l = \sigma[\ell \mapsto S'] \qquad \dot{T}';\sigma_l;v_k\,v_{res} \Downarrow_{\mathrm{E}} T';\sigma';v'}{\dot{T};\sigma;\mathbf{op}\,\ell\,v_{arg}\,v_k \Downarrow_{\mathrm{K}} \mathtt{op}_{v_k\checkmark}^{\ell,v_{arg}\downarrow v_{res}}\cdot T';\sigma';v'}\ \mathbf{op}$$

$$\frac{\dot{T};\sigma;e \Downarrow_{\mathrm{E}} T';\sigma';v'}{\dot{T};\sigma;\mathbf{memo}\,e \Downarrow_{\mathrm{K}} \mathtt{memo}^{e}\cdot T';\sigma';v'}\ \mathbf{memo/miss} \qquad \frac{\sigma;T;e \overset{\mathrm{m}}{\rightsquigarrow} T_e \qquad T_e;\sigma \curvearrowright T';\sigma';v'}{T;\sigma;\mathbf{memo}\,e \Downarrow_{\mathrm{K}} \mathtt{memo}^{e}\cdot T';\sigma';v'}\ \mathbf{memo/hit} \qquad \frac{}{\dot{T};\sigma;\mathbf{halt}\,v \Downarrow_{\mathrm{K}} \mathtt{halt}^{v};\sigma;v}$$

Figure 6. Reduction $e \Downarrow v$ (top) and evaluation $\dot{T};\sigma;e \Downarrow_{\mathrm{E}} T';\sigma';v'$ and $\dot{T};\sigma;\kappa \Downarrow_{\mathrm{K}} T';\sigma';v'$ (bottom).

of actions with a valid checkmark ✓ are replayable by change-propagation and the earliest (if any) manipulation action with an invalid checkmark ✗ must be re-executed by change-propagation. The *trace reparation* and *operation invocation* judgements (Figure 7) use the state-transformation rules to maintain trace consistency.

The trace reparation judgement $S;\dot{T} \overset{\mathrm{rep}\,\ell}{\Rightarrow} \dot{T}'$ takes a TDT state S at location ℓ and an optional reuse trace $\dot{T}$ (with possible inconsistencies in actions that manipulate ℓ) to produce the consistent optional trace $\dot{T}'$. Intuitively, trace reparation identifies the earliest inconsistent action that manipulates ℓ and marks it with an invalid checkmark. A halt action isn't subject to any repair. Any action that does not manipulate ℓ is preserved and the tail of the trace is recursively repaired (**rep/indep**). For any action that manipulates ℓ, the state-transformation is simulated on the TDT state S. If the state-transformation produces the same answer, the action receives a valid checkmark ✓ and the tail of the trace is recursively repaired with the simulated new TDT state S' (**rep/✓**). Otherwise the action receives an invalid checkmark ✗ and the resulting trace is consistent (**rep/✗**).

The invocation judgements $v_{mk};\dot{T} \overset{\mathbf{mk}\,\ell}{\Rightarrow} S';\dot{T}'$ and $S;v_{arg};\dot{T} \overset{\mathbf{op}\,\ell}{\Rightarrow} S';v_{res};\dot{T}'$ use the corresponding state-transformation judgements for creating and manipulating TDT state. Furthermore, since invoking the operation may affect the consistency of actions in the reuse trace $\dot{T}$ (if any) that manipulate location ℓ, the trace reparation judgement is used to maintain the consistency of the reuse trace (**mk/invoke** and **op/invoke**). Hence, the **mk** and **op** evaluation rules use the invocation judgements to perform the state-transformation and preserve trace consistency; moreover the manipulation action is labeled by a valid checkmark because it is consistent with the rest of the execution trace.

The **memo/miss** rule evaluates a memoization expression $\mathbf{memo}\,e$ and yields a trace $\mathtt{memo}^{e}\cdot T'$, where T' is the trace of the evaluation of e. A present reuse trace T is itself a computation trace from a previous evaluation and is supplied to change-propagation to guide the update. In particular, evaluation may reuse computations memoized in the previous evaluation: the **memo/hit** evaluation rule uses the *memoization* judgement $\sigma;T;e \overset{\mathrm{m}}{\rightsquigarrow} T_e$ (Figure 8) to find a reuse trace T_e that corresponds to a previous run of e (under a (possibly) different store) and switches to change-propagating T_e under the current store. Note that while the expression e may have free locations, the memoization judgement is independent of the store. Hence, the rule switches to change-propagating T_e under the current store to correct any invalid actions in the reuse trace T_e.

The memoization judgement $\sigma;T;e \overset{\mathrm{m}}{\rightsquigarrow} T_e$ searches the reuse trace T for a suffix trace T_e that follows a memoization action $\mathtt{memo}^{e}$; since some actions may be discarded from the reuse trace T, the remaining tail of the trace needs to be made consistent relative to the current store σ. A matching memo action (**hit**) returns the tail of the trace for change-propagation. Memo and TDT state actions can be discarded by proceeding to match the tail of the trace. Discarding a memo does not affect the consistency of the trace because it does not touch the store. Discarding a creation action of location ℓ or a manipulation action on a location ℓ that is not in the store does not affect the consistency of the trace because the location ceases to be in the store; if the location is later re-allocated during evaluation (**mk**), then the reuse trace will be made consistent by the invocation judgement. A manipulation action $\mathtt{op}_{v_k\square}^{\ell,v_{arg}\downarrow v_{res}}$ on a location ℓ that is in the store (**op/rev**) must be explicitly *revoked* because it will no longer be performed, thus the tail of the trace must be repaired relative to the current state $S = \sigma(\ell)$.

Turning to the change-propagation relation (Figure 8), recall that we interpret $T;\sigma \curvearrowright T';\sigma';v'$ as replaying the computation trace T under the store σ, yielding the value v', the updated store σ', and the updated computation trace T'. Replaying a halt action yields the (unchanged) computation result. Replaying a memoization action recursively change-propagates the tail of the trace. Whenever change-propagation is recursively applied, the updated computation trace is extended with an appropriate action. A creation operation $\mathtt{mk}_{v_k}^{v_{mk}\uparrow\ell}$ is consistent with the current store if $\ell \notin \mathrm{dom}\,\sigma$ and can thus be replayed (**mk/reuse**) by regenerating the TDT state S' with seed v_{mk}, extending the store with ℓ bound to S', and recursively change-propagating the tail of the trace. A manipulation operation $\mathtt{op}_{v_k\square}^{\ell,v_{arg}\downarrow v_{res}}$ is consistent with the current store if it has a valid checkmark and thus can be replayed (**op/reuse**) by reexecuting the state-transformation to yield, by invariant, the same result v_{res}, updating the store with ℓ bound to the new state S', and recursively change-propagating the tail of the trace.

Change-propagation falls back to execution either nondeterministically or because the head action is inconsistent with the current store and thus not replayable. A creation operation is inconsistent if the location is already in the store and a manipulation operation is inconsistent if it has an invalid checkmark. Since actions capture their continuation, a trace T can be *reified* back into an command $\lceil T \rceil$ that represents the rest of the computation:

$$\lceil \mathtt{halt}^{v} \rceil = \mathbf{halt}\,v \qquad \lceil \mathtt{memo}^{e}\cdot T \rceil = \mathbf{memo}\,e$$
$$\lceil \mathtt{mk}_{v_k}^{v_{mk}\uparrow\ell}\cdot T \rceil = \mathbf{mk}\,v_{mk}\,v_k \qquad \lceil \mathtt{op}_{v_k\square}^{\ell,v_{arg}\downarrow v_{res}}\cdot T \rceil = \mathbf{op}\,\ell\,v_{arg}\,v_k$$

Thus, change-propagation can reify and re-evaluate an inconsistent trace T (**change**), while keeping the trace T for possible reuse later. Note that the reified **mk** (resp. **op**) command forgets the (stale) location (resp. result value).

We can now sketch the use of change-propagation by a host program that (re-)evaluates a self-adjusting computation. Suppose we have a Tgt program e such that $\Sigma;\cdot \vdash e : \mathbf{res}$ and an initial store σ_0 such that $\vdash \sigma_0 : \Sigma \uplus \Sigma_0$. Thus, we may (initially) evaluate e under

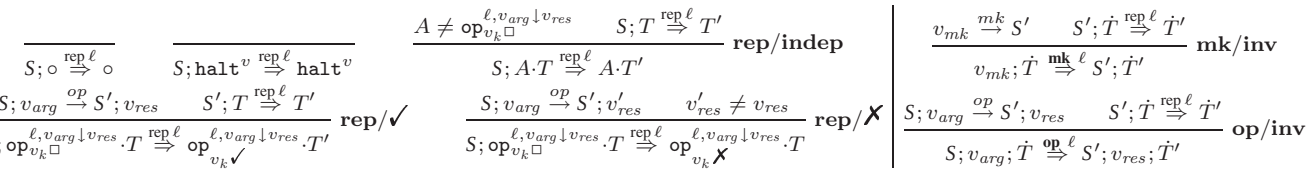

Figure 7. Trace reparation $S; \dot{T} \overset{\text{rep}\,\ell}{\Rightarrow} \dot{T}'$ (left) and invocation $v_{mk}; \dot{T} \overset{\mathbf{mk}\,\ell}{\Rightarrow} S'; \dot{T}'$ and $S; v_{arg}; \dot{T} \overset{\mathbf{op}\,\ell}{\Rightarrow} S'; v_{res}; \dot{T}'$ (right).

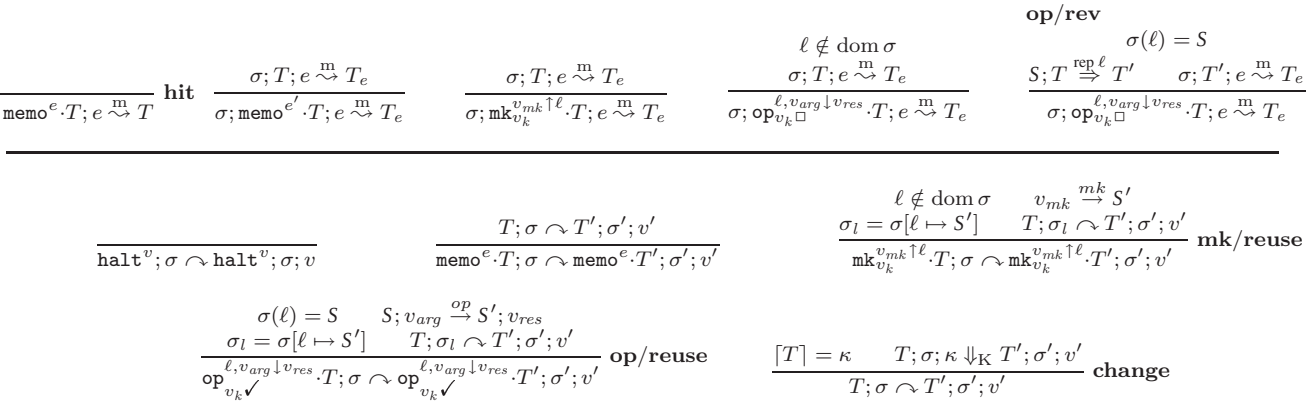

Figure 8. Memoization $\sigma; T; e \overset{\text{m}}{\rightsquigarrow} T'$ (top) and change-propagation $T; \sigma \curvearrowright T'; \sigma'; v'$ (bottom).

the store σ_0 and an empty reuse trace, yielding the (initial) result v_0' and a computation trace T_0': $\circ; \sigma_0; e \Downarrow_{\text{E}} T_0'; \sigma_0'; v_0'$. Now, suppose we have a modified store σ_1 such that $\vdash \sigma_1 : \Sigma \uplus \Sigma_1$. We are interested in the result v_1' yielded by (re-)evaluating e under σ_1. To obtain v_1', we may change-propagate the trace T_0' under the store σ_1: $T_0'; \sigma_1 \curvearrowright T_1'; \sigma_1'; v_1'$. The correctness of change-propagation asserts that the v_1', σ_1', and T_1' obtained via the change-propagation relation could also have been obtained via the evaluation relation: $\circ; \sigma_1; e \Downarrow_{\text{E}} T_1'; \sigma_1'; v_1'$. Hence, change-propagation suffices to determine the output of a program on changed inputs.

5. Extensions to ΔML

We extended the ΔML language to support TDTs and implemented a number of traceable data structures (as specified in Section 3). The ΔML language is implemented as an extension to the MLton compiler and a library for self-adjusting computation implemented in Standard ML. Our extensions to ΔML consist of some small modifications to the change-propagation implementation and a mechanism for integrating TDTs with change propagation. Like earlier implementations of change propagation, we use a totally ordered set of timestamps to represent trace elements, which now include the TDT operations.

Each TDT is implemented as a Standard ML module (see Section 3.2) but integrated with the library for self-adjusting computation through the use of boilerplate code. For each invoke operation, we create a timestamp, essentially making the operation an element of the trace. When the change-propagation algorithm deletes a timestamp, we revoke the operation that is associated with that timestamp. During change-propagation, trace elements that need re-evaluation are stored in a queue prioritized by their timestamps, including the inconsistent operations of all TDTs. Since the set of inconsistent operations dynamically changes over time as a result of invokes and revokes, we adjust the priority queue dynamically to maintain the correct set of inconsistent operations.

Benchmark	Data Types Used
hsort-int	priority queue
dot-product	accumulator
intersection	dictionary
huffman	priority queue
stabbing	priority queue, counter
graham-scan	priority queue
dijkstra	priority queue, dictionary
bfs	queue, dictionary
Motion Simulation	modular modifiable

Table 2. Summary of data types used in our benchmarks. Every self-adjusting program also use the modifiable data type.

6. Experiments

We empirically investigate the performance of the proposed approach. We use a set of diverse benchmarks to compare the space usage and time performance of programs using TDTs to that of programs using standard, modifiable-based implementations. The results show that traceable data structures significantly help improve speed and reduce memory consumption. To understand the source of this performance improvement, we study how tracing at the granularity of data-structuring operations affects the trace size and stability. Our findings suggest that tracing operations on data structures helps reduce the trace size and improve stability by asymptotic factors. In Section 7, we demonstrate the utility of the proposed approach to motion simulation.

6.1 Benchmarks

We developed a set of benchmark to study the performance characteristics of the proposed approach. Each benchmark is specified by a static algorithm's description. Based on this description, we implemented three versions: (1) a static program ("static"), (2) a self-adjusting program that does not utilize TDTs ("modref-based"), and (3) a self-adjusting program that makes use of TDTs whenever appropriate ("traceable"). In developing the test suite, we first implemented the static program and transformed it into a

self-adjusting program using approaches taken in previous work. The traceable version is identical to the modref-based version, except the traceable version makes calls to traceable data structures whereas the modref-based version makes calls to modref-based implementations of data structures. We summarize in Table 6.1 the data types used in each benchmark.

— **Heap sort** (hsort-int): sort an integer list using the standard heap sort algorithm.

— **Dot product** (dot-product): compute the dot product of two real-number vectors represented as a list of ordered pairs, by first computing the product for each component and using an accumulator to compute the sum.

— **List intersection** (intersection): compute the intersection of lists ℓ_1 and ℓ_2, by inserting the elements of ℓ_1 into a dictionary and selecting the elements of ℓ_2 that are present in the dictionary.

— **Huffman code** (huffman): construct a Huffman tree for a list of keys and frequencies using the standard Huffman algorithm.

— **Interval stabbing** (stabbing): take as input a list of intervals $I = \{[a_i, b_i)\}_{i=1}^{n}$ and a list of queries $Q = \{q_j\}_{j=1}^{m}$, and report for each query q_j how many intervals this query "stabs" (i.e., the size of the set $\{(a_i, b_i) \in I : a_i \leq q_j < b_i\}$). We present a plane-sweep algorithm: First, insert into a priority queue the endpoints of all the intervals and the query values, known as events, and set initialize a counter c to 0. Then, to answer queries, consider the events in an increasing order of their values, incrementing the counter on a left endpoint, decrementing it on a right endpoint, and outputting the counter value on a query.

— **Graham Scan** (graham-scan): compute the convex hull of a set of points in 2D using the Graham's scan algorithm (more in Section 6.8).

— **Dijkstra** (dijkstra): compute the shortest-path distances in a weighted graph from a specified source node using Dijkstra's algorithm and output a dictionary mapping each node to its distance to the source.

— **Breadth-First Search** (bfs): perform a breadth-first search, which computes the shortest paths in an unweighted graph from a specified source node and outputs a dictionary mapping each node to its distance to the source.

6.2 Modref-based Data Structures

We implemented modref-based data structures for every data type used in the benchmarks. These implementations may not be the best one can obtain using modifiables alone, but they are reasonable baselines because we believe they are representative of what a programmer with significant background in self-adjusting computation would come up with after some optimization. The accumulator data structure is implemented by maintaining a modifiable list and running a self-adjusting `fold` operation to obtain the solution. Both the dictionary and priority queue data structures are implemented using the Treap data structure. For priority queues, we found that Treap is more stable than common alternatives (e.g., leftist heap, binary heap). The queue data structure is obtained by essentially transforming a standard purely functional implementation of a queue, one which maintains two lists; however, we are especially careful about when the front list is reserved to enhance stability.

6.3 Input Generation

We use randomly generated data sets for all the experiments. Let n be the target input size. For the sorting benchmarks, we generate a random permutation of $\{1, 2, \ldots, n\}$. For dot-product, we generate random vectors by picking floating-point numbers uniformly at random from $[0.0, 10.0]$ (with 5 significant digits). For intersection, We generate a pair of lists of lengths n and m by picking integers uniformly at random from the set $\{0, \ldots, t\}$, where $t = \frac{1}{4}\min\{n, m\}$; this choice of t ensures that the two lists have a common element with high probability. For huffman, the alphabets are simply the numbers 1 to n, and the frequencies are random integers drawn from the range $[1, 10n]$. For stabbing, the endpoints and query values are random numbers in the range $[0, n/10]$ chosen uniformly at random. For convex hulls, we generate inputs by drawing points uniformly from the circumference of a unit-radius circle. This arrangement is known to be a challenging pattern for many convex-hull algorithms. For our graph benchmarks, we generate random, connected graphs with approximately $\sqrt{n}$-separators, mimicking the fact that many real-world graphs have small separators (e.g., $n^{1-\varepsilon}$).

6.4 Metrics and Measurements

The metrics for this study are **(1)** the time to run a program from scratch, denoted by T_i **(2)** the average update time after a modification, denoted by T_u, and **(3)** the space consumption, denoted by S. To measure the second metric, for example, in list-based experiments, we apply a delete-propagate-insert-propagate step to each element (i.e., in each step, delete an element, run change-propagation, insert the element back, and run change-propagation) and divide the end-to-end time by $2n$, where n is the list's length. This quantity represents the expected running time of change-propagation if a random update to the input is performed. We can use this measurement in graph experiments, where here the delete-propagate-insert-propagate is applied to each edge in turn. All measurements were taken on a standard Linux machine[5].

We measure the space consumption by noting the maximum amount of live data as reported by ΔML's garbage collector. This is an approximation of the actual space usage because garbage collection may miss the high-water mark.

When measuring time, we carefully break down the execution time into application time and garbage collection (GC) time. In these experiments, we have found that GC is at most 20% of the execution time. For this reason, we only report the application time to isolate the GC effects and highlight the asymptotic performance.

6.5 Modref-based Programs vs. Traceable Programs

The first set of experiments studies how TDTs provide the performance benefits over traditional, modref-based implementations. Recall that T_i is the time to run a program from scratch and T_u is the average time that change propagation takes to perform an update. Table 3 shows the performance of our benchmark programs, comparing the traceable versions to their modref-based counterparts. Note that for graham-scan, the modref-based program uses merge sort whereas the traceable program uses heap sort; the modref-based version of heap sort is too slow except for extremely small inputs. We explore this in more detail in Sections 6.8 and 6.9.

We find that compared to the modref-based programs, the traceable versions are 3–20 times faster to run from scratch and 4–5000 times faster to perform an update. Moreover, traceable versions consume 4–50 times less space than the modref-based ones. We remark that these experiments involve relatively small input sizes because with larger inputs our experiments with some modref-based applications require too much time to complete.

[5] **Technical Setup:** Our experiments were conducted on a 2.0Ghz Intel Xeon E5405 with 32 GB of memory running Ubuntu 8.04 (kernel 2.6.24-19). Programs were compiled using the ΔML compiler [19], a modified version of the MLton compiler version 20070826, with the option "`-runtime ram-slop 0.9 gc-summary`" These options direct the runtime system to make available 90% of the physical memory to the benchmark and report statistics about garbage collection (GC).

Experiment	Size	Traceable			Modref-based			Modref-based ÷ Traceable		
	N	T_i (ms)	T_u (μs)	S (MB)	T_i (ms)	T_u (μs)	S (MB)	T_i	T_u	S
hsort-int	10^3	7.50	35.00	0.61	85.00	27695.00	14.04	11.33	791.28	23.02
dot-product	10^5	280.00	6.75	52.88	872.50	121.55	223.80	3.11	18.00	4.23
intersection	10^5	1372.50	82.00	382.78	11207.50	1948.45	1509.17	8.16	23.53	3.94
huffman	10^4	157.50	492.00	22.13	2575.00	2530000.00	707.61	16.34	5142.28	31.98
stabbing	10^3	17.50	115.00	1.92	240.00	98195.00	23.56	13.71	853.87	12.27
graham-scan	10^4	375.00	265.50	24.90	1542.50	1105.50	277.24	4.11	4.16	11.13
bfs	10^3	37.50	845.56	2.74	717.50	23784.07	139.39	19.13	28.12	50.82
dijkstra	10^3	42.50	1160.03	2.74	725.00	34528.30	72.41	17.05	29.76	26.42

Table 3. Traceable vs. modref-based implementations: T_i (in ms) is the from-scratch execution time, T_u (in μs) is the average time per update, and S (in MB) is the maximum space usage as measured at garbage collection.

Experiment	Size	Traceable			Static	Overhead	Speedup
	N	T_i (ms)	T_u (μs)	S (GB)	T_i (ms)	(SAC T_i)/(static T_i)	((static T_i)/SAC T_u
hsort-int	10^6	14390.00	59.02	1.75	2599.75	5.5	4.4×10^4
dot-product	10^6	2787.50	7.45	0.44	100.25	27.80	1.3×10^4
intersection	10^6	12820.00	74.91	2.19	1091.50	11.74	1.5×10^4
huffman	10^6	22975.00	1021.04	1.08	6447.25	3.56	6.3×10^3
stabbing	10^6	38832.50	202.11	1.70	10609.75	3.60	5.2×10^4
graham-scan	10^5	4307.50	297.30	0.70	547.75	7.86	1.8×10^3
bfs	10^4	445.00	1310.59	0.12	47.50	9.36	36.2
dijkstra	10^4	490.00	1783.68	0.12	52.50	9.33	29.4

Table 4. Traceable SAC versus static: T_i (in ms) is the from-scratch execution time, and T_u (in μs) is the average time per update. Update times are reported in **microseconds** (μs).

6.6 Traceable Programs vs. Static Programs

Our second set of experiments, shown in Table 4, draws a comparison between traceable programs and static programs, quantifying the effectiveness of the approach in more absolute terms. First, consider the overhead column, calculated as the ratio of the from-scratch run of the traceable implementation to that of the static implementation. This quantity represents the overhead of the proposed approach (e.g., due to dependence tracing, runtime system). We find the overhead to relatively small: the traceable versions are about a factor of 10 slower than their static counterparts, except for dot-product, which is about 30 time slower. We believe this is because the benchmark dot-product is relatively lightweight computationally.

Second, consider the speedup column, calculated as the ratio of the static from-scratch run time to the update time. Results show that the traceable versions can perform updates many orders of magnitude faster. One exception is our graph algorithms, which are output-sensitive and may need to update the results at many nodes even after a small modification, e.g., deleting a single edge can change the shortest distance of many nodes. We discuss this in greater depth next.

6.7 Graph Algorithms

Graph algorithms can be challenging with previous approaches to SAC. We discuss how TDTs can help overcome some of these challenges. While previous SAC approaches worked well on problems with structured data (e.g., lists and trees), computations involving unstructured data (e.g., graphs) often require use dynamic data structures whose traditional self-adjusting versions can require the tracing and updating of large amount of dependencies. Traceable data types address this problem by reducing the amount of required tracing and exploiting problem-specific structures, thereby dramatically decreasing the update time.

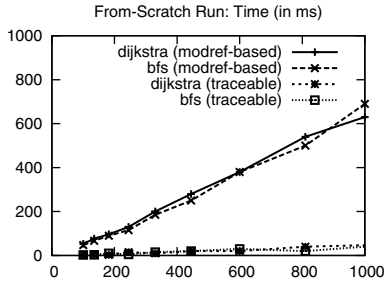

Figure 9. From-scratch runs with our graph benchmarks: timing (vertical axis in **ms**) as input size (horizonal axis) varies.

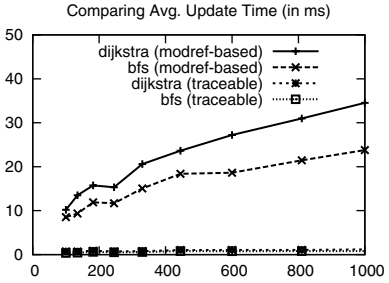

Figure 10. Updates with our graph benchmarks: timing (vertical axis in **ms**) as input size (horizonal axis) varies.

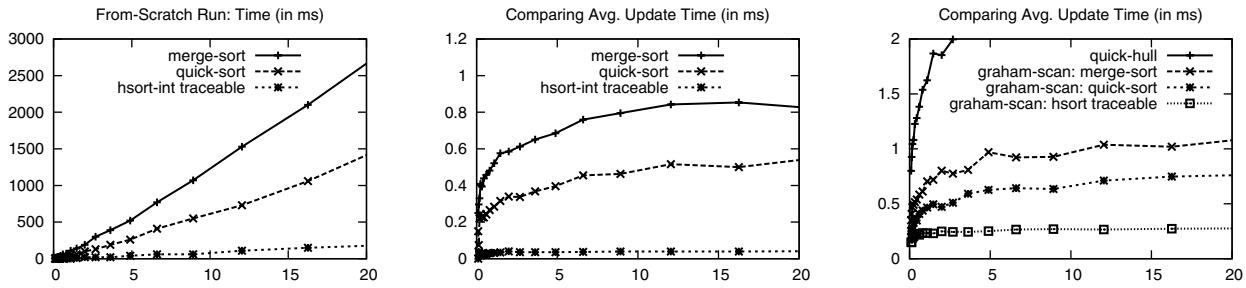

Figure 11. Detailed measurements for the sorting and graham-scan experiments: timing (vertical axis in **ms**) as input size (horizontal axis in **thousands** of elements) is varied.

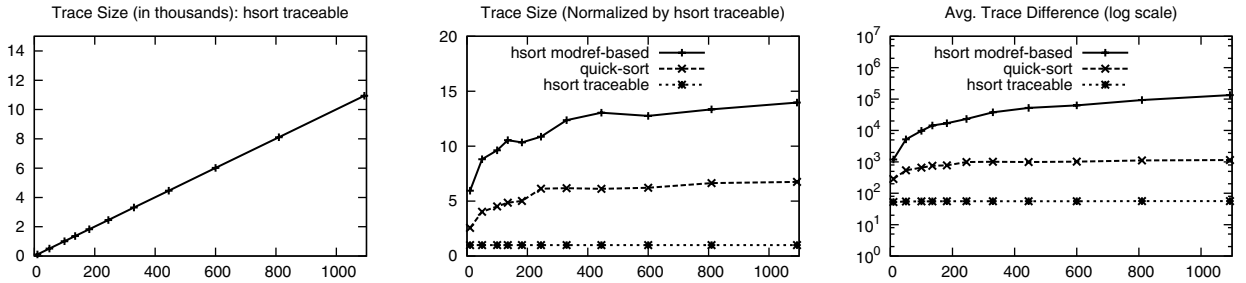

Figure 12. Trace size (in **thousands** of trace elements) and average trace difference (in trace elements on a log scale) of sorting benchmarks as input size is varied: trace size of traceable heap sort (left), trace size of quick sort and modref-based heap sort as normalized by the trace size of traceable heap sort (center), and average trace difference (right).

We consider two algorithms: the Dijkstra's single-source shortest path algorithm (dijkstra) and the classic breath-first-search algorithm (bfs). Our implementations follow the standard textbook descriptions (Figure 5 shows the pseudo-code for dijkstra). Both algorithms use a dictionary to represent a graph.

Figures 9 and 10 contrast the performance of traceable versions of shortest-path algorithms with that of the traditional, modref-based versions. Figure 9 shows from-scratch execution times of dijkstra and bfs. Both perform similarly, and their traceable versions are significantly faster than their traditional versions, by more than an order of magnitude at peak. Figure 9 shows the average update times for an edge deletion/insertion. Again, both benchmarks perform similarly and the traceable versions are significantly faster than the traditional, by approximately an order of magnitude at $N = 1,000$.

We note that both dijkstra and bfs are highly output sensitive algorithms. Since inserting/deleting an edge can change the shortest-path distances on a large number of nodes, these benchmarks are highly output sensitive. Specifically, if the shortest-path distances change on t nodes, both benchmarks will need to update all t nodes, requiring at least $\Omega(t)$ time.

6.8 Sorting and Convex Hulls

Another noteworthy feature of the TDT framework is modularity, specifically the fact that we can often enjoy substantial performance improvements by simply replacing the modref-based implementations of data structures with the compatible traceable versions. As an example, consider the problem of computing the convex hull of 2D data points. Given a set of 2D points, Graham's scan algorithm first orders the points by the x coordinates and computes the convex hull by scanning the sorted points. Here we compare the traceable version of our heap sort (hsort) and graham-scan benchmarks with other modref-based algorithms considered in previous work. The fastest version turns out to be identical to the old graham-scan code, except the sort routine is now a traceable heap sort.

As shown in Figure 11 (left and center plots), traceable heap sort outperforms the quick-sort and merge-sort algorithms by nearly an order of magnitude for both from-scratch runs and updates. Since graham-scan uses sorting as a substep, it shows the same performance trends (rightmost plot). Compared to the previous modref-based implementation of the quick-hull algorithm [8], graham-scan is extremely fast.

6.9 Trace Size and Stability

Our empirical measurements thus far illustrate the performance benefits of TDTs, both in running time and space consumption. Here we investigate the question of whether these improvements are related to potential constant factor improvements in the runtime systems or to the benefits of TDTs as we expect them to be. Our measurements suggest the latter and indicate asymptotic improvements in performance. To this end we consider two abstract measures: trace size and trace stability. Trace size measures the size of the memory consumed. Trace stability measures how much the trace changes as a result of an input modification—this ultimately determines how fast the program can respond to modifications. In our experiments, we measure the trace size by the number of trace elements, and the trace stability by counting the average number of trace elements created and deleted during change propagation after a single insertion/deletion. These measures are independent of the specifics of the hardware as well as the specifics of the data structures used for change propagation—they only depend on the abstract representation of the trace. They are, however, specific to the particular self-adjusting program and the class of input changes considered. As an example, we consider here sorting with integers, specifically hsort-int, with traceable and modref-based priority queues, and a self-adjusting implementation of quicksort.

Figure 12 (leftmost) shows the trace size for traceable heap sort as the input size increases. Regression analysis shows a perfect fit with $10n + 12$ (n is the input size), providing strong evidence that the trace size of traceable heap sort is $O(n)$. This is consistent with

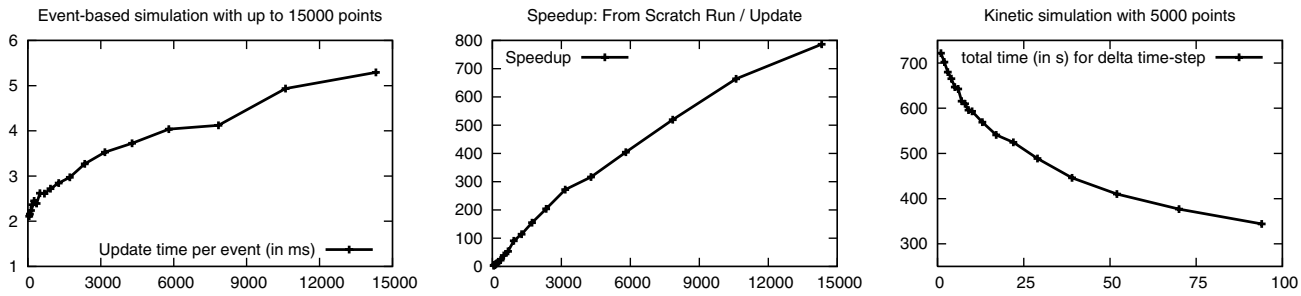

Figure 13. Left: Time per kinetic event. **Center:** Speedup for an update. **Right:** Total simulation time (seconds) with time-slicing.

a simple analytical reasoning on traces: since we record dependencies at the level of priority-queue operations and since heap-sort performs linear number of such operations, the trace has linear size.

Figure 12 (center) shows the trace size of hsort-int using both traceable and modref-based priority queues normalized to the trace size of the traceable heap sort. The figure suggests that the traces of traceable heap sort are by a factor of $\Theta(\log n)$ smaller than those of the modref-based. This explains why traceable heap sort has a significantly smaller memory footprint than the modref-based counterpart.

Figure 12 (right) shows our measurements of average trace difference on a vertical log scale for a single insertion/deletion. Trace difference is constant for traceable heap sort, because a single insertion/deletion requires inserting/deleting a single priority queue operation from the trace. The modref-based implementation heap sort appears to have super-logarithmically larger trace difference. The reason for this is the internal comparisons traced by the modref-based priority queues. This finding explains the difference in the runtime performance between the two implementations of heap sort.

Figure 12 also compares the traceable heap sort to our self-adjusting quicksort implementation, which, until now, has been the most efficient self-adjusting sorter. Traceable heap sort appears faster by at least a moderate constant factor.

7. Motion Simulation

With traceable data types, programs can natively and safely handle continuous-domain input, so that the output may be updated efficiently when the input changes. We consider motion simulation as an interesting application of this capability. Consider a program P that computes a geometric property (e.g., convex hull) of a given set of static objects (e.g., points). In motion simulation, we want to compute the output of P as the objects move continuously, i.e., when each coordinate is a function of "time". Motion simulation can be performed by time slicing and recomputing the output at fixed intervals. This approach, however, is inaccurate because it only approximates the times at which the output changes, and inefficient because it cannot take advantage of the similarity of output at consecutive intervals. With no advance knowledge of how points move, time slicing can be the only option. In some cases, however, we can represent the coordinates of moving objects with polynomials of time and compute exactly the times at which the output can change by finding the roots of certain polynomials (e.g., [7]). We call this an *event-based simulation*.

We implement a library for motion simulation by using both event-based and time-slicing simulation techniques while taking advantage of self-adjusting computation to update the output. The library allows the "time" to be set arbitrarily and uses change-propagation to update the output. We use modular modifiables (Section 3) to represent outcomes of geometric tests. The library consist of $3,200$ lines of ΔML code supplying primitives for polynomials, geometric operations, and performing motion simulation. As benchmarks, we implement self-adjusting versions of several 2D convex hulls algorithms and one 3D convex hull algorithm called incremental-hull algorithm. We also implement a visualizer that helps us observe motion simulations in real time by simultaneously running the visualizer and the self-adjusting program performing the simulation. Some example movies can be found on the web site `http://sites.google.com/site/sacmotion/`.

Figure 13 shows some experimental results with 3D hulls using event-based approach. For both figures the horizontal axis is the input size consisting of points (up to 15000). The plot on the left shows the average time to update the output by change propagation after changing the time in an event-based simulation (average taken over n updates for each input size n). The update time appears to grow slowly (poly-logarithmically) with the input size. The plot on the right shows that updates are nearly three orders of magnitude faster for larger inputs; speedups are computed by comparing to static from-scratch execution.

Imagine performing motion simulation by recomputing the convex hull periodically every δ milliseconds, i.e., by time slicing. If δ is reasonably small, we expect the output computed at consecutive intervals to be similar. Self-adjusting computation allows us to take advantage of this similarity. Figure 13 (right) shows the total simulation time for varying interval sizes with 5000 moving points. The horizontal axis represents δ in milliseconds and the vertical axis represents the total simulation time. As the interval size increases the total simulation time decreases quite dramatically especially initially, because as the interval size increases, more events can be processed simultaneously and fewer events occur in total.

Finally, although we do not discuss here in detail, the ability to handle continuous-domain inputs makes a range of modifications possible. For example, our approach allows the time to be set to any value, even in the past and update the output efficiently.

8. Related Work

The problem of having computation respond to slowly changing data has been studied extensively. Early work in the programming languages community, broadly called *incremental computation*, focused on developing techniques for translating static/conventional programs into incremental programs that can respond automatically to input modifications. Recent advances on self-adjusting computation have generalized these approaches and dramatically improved their effectiveness. The algorithms community devised dynamic and kinetic data structures to address these same problems. This section is a brief survey of related work in these two areas; detailed information can be found elsewhere [7, 11, 14, 24].

Incremental Computation. The most effective incremental computation techniques are based on dependence graphs, memoization, and partial evaluation. Dependence graph techniques record the de-

pendencies between data in a computation and rely on a change-propagation algorithm to update the computation when the input is modified (e.g., [12, 18]). These techniques have been shown to be effective in some applications, e.g., syntax-directed computations. They are not general-purpose because they do not allow the change-propagation algorithm to update the dependence structure. For example, the INC language [27], which uses static dependence graphs, does not permit recursion. As an alternative to dependence graphs, memoization (also called function caching) has been investigated (e.g., [1, 17, 23]). This classic idea dating back to the late 1950's [9, 21, 22] applies to any purely functional program and therefore is more broadly applicable than static dependence graphs. In incremental computation, memoization can improve efficiency when executions of a program with similar inputs perform similar function calls. This turns out to be relatively rare: it is often the case that small input modifications can prevent reuse via memoization as the arguments to many functions are modified. Partial incremental computation with partial evaluation [15, 26] requires the user to fix the partition of the input that the program will be specialized on and can then process modification faster by partially evaluating the program with respect to the fixed part of the input. The main limitation of this approach is that it allows input modifications only within a predetermined partition.

Self-Adjusting Computation. Self-adjusting computation combines dynamic dependence graphs [2] and a form of computation memoization [5] to achieve efficient updates. Variants of self-adjusting computation have been implemented in several host languages such as C [16], Java [25], Haskell [10], and SML [19]. The approach has been shown to be effective for a reasonably broad range of problems (e.g., [4, 5]. Recently, techniques inspired by self-adjusting computation have resulted in an efficient algorithm for dynamic maintenance of well-spaced point sets, settling an open problem [6].

9. Conclusion

We present an approach to tracing dependencies in computations at the level of (abstract) data types operations. Since the number of accesses to an abstract data type can be asymptotically less than the number of accesses to memory, our approach can asymptotically reduce the number of dependencies to be traced. For example in heapsort there are only $O(n)$ accesses to the heap (priority queue) instead of $O(n \log n)$ total operations, and indeed our experiments show an order of magnitude improvement. In the context of self-adjusting computation, these techniques translate to dramatic improvements in space and time. Furthermore in some cases the trace with respect to the data type operations can be stable even if at the memory cell level it is not. This can greatly improve the performance of change propagation, as seen in the Huffman code benchmark.

References

[1] M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and Caching of Dependencies. In *Proceedings of the International Conference on Functional Programming*, pages 83–91, 1996.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems*, 28(6):990–1034, 2006.

[3] U. A. Acar, G. E. Blelloch, and K. Tangwongsan. Non-oblivious retroactive data structures. Technical report, Carnegie Mellon University, 2007.

[4] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, September 2008.

[5] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(1): 3:1–3:53, 2009.

[6] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *SCG '10: Proceedings of the 26th Annual Symposium on Computational Geometry*, 2010.

[7] P. K. Agarwal, L. J. Guibas, H. Edelsbrunner, J. Erickson, M. Isard, S. Har-Peled, J. Hershberger, C. Jensen, L. Kavraki, P. Koehl, M. Lin, D. Manocha, D. Metaxas, B. Mirtich, D. Mount, S. Muthukrishnan, D. Pai, E. Sacks, J. Snoeyink, S. Suri, and O. Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002. ISSN 0360-0300.

[8] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[9] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[10] M. Carlsson. Monads for Incremental Computing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming*, pages 26–35. ACM Press, 2002.

[11] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[12] A. Demers, T. Reps, and T. Teitelbaum. Incremental Evaluation of Attribute Grammars with Application to Syntax-directed Editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

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

[14] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

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

[16] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[17] A. Heydon, R. Levin, and Y. Yu. Caching Function Calls Using Precise Dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 311–320, 2000.

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

[19] R. Ley-Wild, M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming*, 2008.

[20] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

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

[22] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218: 19–22, 1968.

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

[24] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 502–510, 1993.

[25] A. Shankar and R. Bodik. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming language Design and Implementation*, 2007.

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

[27] D. M. Yellin and R. E. Strom. INC: A Language for Incremental Computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, Apr. 1991.