Adaptive Memoization [*]

Umut A. Acar　　Guy E. Blelloch　　Robert Harper

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{umut,blelloch,rwh}@cs.cmu.edu

Abstract

We combine adaptivity and memoization to obtain an incremental computation technique that dramatically improves performance over adaptivity and memoization alone. The key contribution is *adaptive memoization*, which enables result re-use by matching any subset of the function arguments to a previous function call and updating the result to satisfy the unmatched arguments via adaptivity.

We study the technique in the context of a purely functional language, called IFL, and as an ML library. The library provides an efficient implementation of our techniques with constant overhead. As examples, we consider Quicksort and Insertion Sort. We show that Quicksort handles insertions or deletions at random positions in the input list in $O(\log n)$ expected time. For insertion sort, we show that insertions and deletions anywhere in the list take $O(n)$ time.

1 Introduction

Memoization [15, 16, 11, 4] and adaptivity [2, 5] are techniques for making any program incremental. Although each technique works well for certain classes of applications under certain input changes, neither works well in general. This paper combines adaptivity and memoization. The result is a general technique that significantly improves performance over adaptivity and memoization alone. For many applications, however, an orthogonal combination of memoization and adaptivity does not yield good performance. We therefore introduce *adaptive memoization* that enables memo lookups based on matching any subset of function arguments to a previous function call and updating the re-used result to satisfy the unmatched arguments via adaptivity.

Memoization [7, 13, 12] is based on the idea of caching the results of each function call indexed by the arguments to that call. If a function is called with the same arguments a second time, the result from the cache is re-used and the call is skipped. Pugh [15], and Pugh and Teitelbaum [16] were the first to apply memoization or function caching to incremental computation. They developed techniques for implementing memoization efficiently and studied incremental algorithms using memoization. They showed that certain divide-and-conquer algorithms using so-called stable decompositions can be made incremental efficiently by using memoization. Liu, Stoller, and Teitelbaum [11] presented systematic techniques for developing incremental programs using function caching. Their techniques automatically determine what result need to be cached and use transformations to make a standard program incremental. In recent work [4] we presented selective memoization techniques to provide control over the performance of memoization based on facilities for determining precise input-output dependences, defining equality, and controlling space usage.

Adaptivity [2] is based on the idea of representing computations with dependence graphs. Dependence graph techniques for incremental computation were first introduced by Demers, Reps, and Teitelbaum [8, 18] and have been successfully applied to many applications [17]. Dependence graphs represent data dependences in a computation in such a way that when an input is changed, all data that depends on that input can be updated by propagating changes through the graph.

The key difference between adaptivity and the previously proposed dependence-graph techniques is that in adaptivity the dependence graphs are dynamic as opposed to static. With static dependence graphs, change propagation only updates the values of the vertices of the dependence graph leaving the dependence structure unchanged. As Pugh points out [15] this limits the kinds of applications that can be made incremental using static dependence graphs. In contrast, dynamic dependence graphs enable the change propagation algorithm to update the dependence structure by removing obsolete dependences and inserting newly created dependences based on execution. Dynamic dependence graphs can be used to make any purely functional program incremental, although the effectiveness will depend on the application.

Adaptivity and memoization complement each other in the way they support result re-use. While adaptivity pinpoints parts of a computation that are affected by some input change, memoization identifies those parts of the computation that remain the same. As a result, memoization handles well *shallow* input changes which affect function calls at the top of the function-call tree, whereas adaptivity handles well *deep* changes which affect leaves of the function call tree [4]. When given an input change that affects some call in the middle, they can both perform poorly.

This paper shows that a combination of adaptivity and memoization yields powerful techniques for incremental computing by drawing on the complimentary strengths of memoization and adaptivity. We present two techniques to this end: (1) an orthogonal combination that combines adaptivity and memoization by preserving their semantics, and (2) a more sophisticated combination based on the notion of *adaptive memoization.*

Adaptive memoization allows an imprecise lookup of a

[*]This work was supported in part by the National Science Foundation under the grant CCR-9706572 and also through the Aladdin Center (www.aladdin.cs.cmu.edu) under grants CCR-0085982 and CCR-0122581.

memoized function in the function cache by matching just some of the arguments that the result depends on. Instead of returning the result, which is in general incorrect, the lookup returns an "adaptive computation" in the form of a dynamic dependence graph. The arguments of this computation are then adjusted to match the arguments of the current call, and the changes are propagated to update the re-used result.

Adaptive memoization provides for flexible result re-use by permitting the re-use of the result of a previous function call in place of a call of that function with somewhat different arguments. This flexibility enables us to obtain asymptotically efficient incremental algorithms from static algorithms. As examples, we consider Quicksort and Insertion Sort on a list (Section 5). We show that Quicksort handles an insertion or deletion at a random position in the input in expected $\Theta(\log n)$ time. For insertion sort, we show that an insertion or deletion anywhere in the input takes expected $\Theta(n)$ time. These results rely heavily on adaptive memoization; with the orthogonal combination the bounds are $\Theta(\log^2 n)$ for Quicksort and $\Theta(n^2)$ for Insertion Sort. With memoization or adaptivity alone, the bounds are $\Theta(n \log n)$ for Quicksort, and $\Theta(n^2)$ for insertion sort.

Challenges to combining adaptivity and memoization and supporting adaptive memoization stem from complexities of the interaction between adaptivity and memoization. One issue is the maintenance of the topological ordering of a dynamic dependence graph while allowing parts of the graph to be re-used. We show that the topological ordering can be maintained with constant overhead by restricting the memo lookups to the part of the dependence graph being discarded by change propagation. Another issue is supporting adaptive memoization efficiently. Adaptive memoization relies on encapsulating selected sub-computations as stand alone adaptive computations. This requires techniques to isolate and update the inputs of sub-computations efficiently. We describe a copy-on-read technique for supporting adaptive memoization with constant overhead.

A key property of our approach is that it accepts a simple and asymptotically efficient implementation. The implementation extends our previous implementations for adaptivity [2] and selective memoization [4]. The overhead of the implementation—slowdown caused by our techniques with respect to a non-incremental semantics—is constant.

2 Overview

We present an overview of previous work on memoization and adaptivity and describe how they can be combined. Section 4 formalizes the techniques presented here. Examples for motivating the need for combining adaptivity and memoization are given in Section 2.3.

2.1 Adaptivity and Dynamic Dependence Graphs

Adaptivity [2] is based on the notion of a *modifiable reference* or a *modifiable* for short. Modifiable references hold values that can change as a result of the user's revisions to the input. What distinguishes a modifiable reference from an ordinary reference is that the system keeps track of the readers of the modifiable and when the value is changed, all values that depend on that modifiable can be updated by a change propagation algorithm.

Language support for adaptivity requires constructs for creating, reading, and writing modifiables. Each read of a modifiable specifies a *reader* function that computes a value based on the value of the modifiable read, called the *source.* Since values that are computed by reading modifiables can change due to an input change, a reader must write its result to a modifiable. In this paper, we require that each reader writes to exactly one modifiable called the *target.*

As an adaptive program executes, it builds a *dynamic dependence graph* or *DDG* that represents the data and control dependences in the execution. Creating a modifiable adds a vertex for that modifiable to the dependence graph. Reading a modifiable inserts an edge from the source to the target of the read and tags the edge with the reader function. Writing a modifiable tags the vertex for that modifiable with the value written. To represent the control dependences, a *containment hierarchy* of reads is maintained. A read r is *contained* in some other read r' if r is created during the execution of r'. The containment hierarchy represent the nesting of the reads of a computation. In the implementation, the containment hierarchy is represented using time stamps instead of containment edges (see Section 3).

When the input to an adaptive computation is changed, the output and the dependence graph can be updated by propagating changes through the dependence graph. Change propagation maintains a set of *affected* readers, readers whose sources have been changed, and re-executes them in sequential-execution order. Re-executing a reader re-establishes the relationship between its source and target by updating the value of the target, which can make affected the readers of the target. Re-executing a reader removes the dependences and the modifiables that was created by that reader in the previous execution, and inserts the dependences and modifiables created by re-execution. Note that due to conditionals, the dependences and size of the graph can change radically after an input change.

Adaptivity yields efficient incremental or dynamic algorithms for certain classes of algorithms and input changes. For example, in our original paper, we showed that Quicksort on a list updates its output in expected $O(\log n)$ time when its input is changed by inserting or deleting one key at the end. In recent work, we developed analytical techniques based on trace-stability for measuring the efficiency of algorithms made incremental using adaptivity [5]. As an example, we showed that the tree contraction algorithm of Miller and Reif yields a data structure for the dynamic-trees problem of Sleator and Tarjan [19]. Our experimental evaluation of the dynamic-trees data structure obtained by adaptivity shows that it is efficient in practice [6].

2.2 Memoization

Memoization caches results of all or selected function calls so that when a call is performed for a second time, the cached result is re-used instead of executing the call. Although memoization can improve performance dramatically, obtaining good performance in general requires control over certain aspects of memoization. These aspects include the type of equality tests that determine cache hits and the identification of precise dependences between input and output.

In his thesis [15], Pugh developed techniques for implementing memoization efficiently and presented techniques for constant-time equality checks and space-management [14]. Based on static program analysis and transformations, Liu and Teitelbaum [11] developed techniques for determining what results to cache and how to use them. Since in general the result of a function call may not depend on all its arguments, it is important to cache result based on precise input-output dependences. Abadi, Lampson, and Levy [1], and Heydon, Levin, and Yu [10] investigated techniques for this purpose based on labeled lambda

```
fun map l =


 case l of
   nil => nil
 | cons(h,t) =>
   cons(h+5,map t)
```

```
fun amap l =
 tar = new modifiable
 read l with reader (fun vl =
   case vl of
     nil => write(tar,nil)
   | cons(h,t) =>
     write(tar,cons(h+5,amap t)))
 return tar
```

Figure 1: The code for standard and adaptive map.

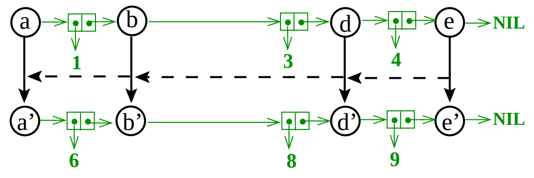

Figure 2: DDG of **amap** on input [1,3,4].

calculus. In recent work, we presented selective memoization techniques that provide programmer control over the issues of precise dependences, equality tests, and some control over space management [4]. Selective memoization enables performance of memoized applications to be analyzed using conventional techniques. As an example, we showed that a memoized version of Quicksort handles an insertion or deletion anywhere in the list in expected $O(n)$ time.

In the context of incremental computation, memoization yields efficient incremental algorithms for certain classes of algorithms and input changes. Pugh [15], and Pugh and Teitelbaum [16] show that divide-and-conquer algorithms that are based on the so-called stable decompositions can be made incremental efficiently using memoization.

2.3 Examples: Map and Insertion Sort.

We apply adaptivity and memoization to "map" and insertion sort and motivate the need for combining adaptivity and memoization. The insertion sort example motivates adaptive memoization by showing that an orthogonal combination of memoization and adaptivity does not suffice for efficient incremental computing in general.

Example I: map. Figure 1 shows the code for a simple map function (left) that maps a list to another list by adding five to each element. The pseudo-code for the adaptive version `amap` is shown on the right. Figure 2 shows an example dynamic dependence graph for `amap` with input list $[1, 3, 4]$. The input to `amap` is a modifiable list, a list where all tails are inside modifiables, and so is the output. In Figure 2, the vertices are modifiables, straight edges are reads, and dashed edges are the containment edges. Values of modifiables are shown in green (or gray). The value of each modifiable is either a cons cell or nil. The readers of the edges are all the same function as shown in the code in Figure 1. Containment edges originate at a reader and end at the reader for the caller—containment edges essentially represent the function call tree of the computation. An edge is contained in all the edges to its left.

Figure 3 shows an example of how the input to `amap` can be changed and the output can be updated. The value two is inserted to the input by creating a new modifiable c and changing the modifiable b. Change propagation involves re-executing the only read of b, which recursively calls `amap` on the new modifiable c. The recursive call recomputes the result for the tail of the input list starting at c. This creates modifiables c', f and g and the edges (c, c'), (d, f) and (e, g).

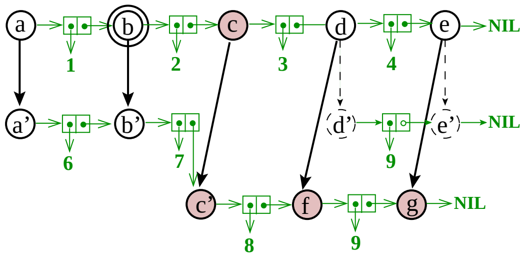

Figure 3: Input change and change propagation.

Since the edges (d, d') and (e, e') are contained in the re-executed edge (b, b') they are removed from the dependence graph along with vertices d' and e'. The removed elements are shown with thinner, dashed lines.

As the example demonstrates, when a new key is inserted to the input, the adaptive map function `amap` will re-compute the result for the tail of new cons cell. Thus an insertion at the end of the input list will be handled in constant time—such an insertion is a deep change, because it affects a leaf of the call tree. In general `amap` will take linear time to update its output.

As an alternative consider a memoized version of `map` where each call to `map` is cached in a memo table. Since inserting a new key into the input re-creates the prefix of the input list up to new key, a memo match will not occur until after the tail of the new key. Thus, an insertion at the head of the list will be handled in constant time–such an insertion is a shallow change because it affects the root of the call tree. In general general memoized `map` will take linear time to update its output.

Since adaptivity handles deep changes well and memoization handles shallow changes well, we can expect that their combination would work well for all changes. Indeed, consider caching the result of calls for `amap` based on the input argument. When the input is changed by an insertion, the reader of the changed modifiable will be re-executed and the second recursive call that the reader performs will find its result in the memo. In our example, inserting three will re-execute the edge (b, b') and the result will be found in the memo when `amap` is called with the modifiable d. Thus a combination of memoization and adaptivity will yield a constant time incremental map for insertions or deletions anywhere in the input list.

Example II: Insertion Sort. As an example where the orthogonal combination of adaptivity and memoization does not yield good performance we consider insertion sort. We show that insertion sort requires worst-case $\Theta(n^2)$ time for an insertion in the middle of the list when using the orthogonal combination. When using adaptive memoization, we show that this reduces to $O(n)$.

Figure 4 shows the code for insertion sort that builds the result by inserting keys to a sorted accumulator list (`a`). Suppose we would like to make insertion sort incremental under a single insertion into the input list (`l`). Consider using adaptivity. As with the map example, when a new key is inserted into the input, the adaptive version will completely re-sort the tail of the list starting at the new key. Thus, although an insertion at the very end of the input will take linear time, an insertion at the head or the middle of the list will take $\Theta(n^2)$ time in the worst case. As an alternative consider the memoized version of insertion sort. Inserting a key at the head or middle of the list will change

```
fun insert (p,l) =
  case l of
    nil => cons (p,nil)
  | cons(h,t) =>
      if (p < h) then cons(p,l)
      else cons(h,insert(p,t))

fun iSort (l,a) =
  case l of
    nil => a
  | cons(h,t) => iSort (t,insert(h,a))
```

Figure 4: Standard insertion sort.

the accumulator for all the following recursive calls, because they will now contain the new key. Thus no results will be found in the memo after that point. Therefore with both memoization and adaptivity insertion at the head or middle will require $\Theta(n^2)$ time.[1] Combining them will not help.

As a concrete example, Figure 5 shows the the accumulators built by the standard insertion sort algorithm (Figure 4) with input $l = [6, 5, 4, 8, 7, 0]$ (left) and $l' = [6, 5, 4, \mathbf{9}, 8, 7, 0]$ (right)—l' is obtained from l by inserting the key 9. Each column corresponds to an insertion to the accumulator; the time advances from left to right. Since each call to `insert` re-creates the accumulator list up to the position where the key is placed and re-uses the tail, some tails are shared—curved arrows show such sharing. Each computation is divided into two boxes, A, B, and A', B', corresponding to the parts before and after the call to `iSort` where the newly inserted key 9 is inserted to the accumulator. In particular, box B corresponds to the call `iSort(`$[8, 7, 0], [4, 5, 6]$`)`, and box B' corresponds to the call `iSort(`$[\mathbf{9}, 8, 7, 0], [4, 5, 6]$`)`.

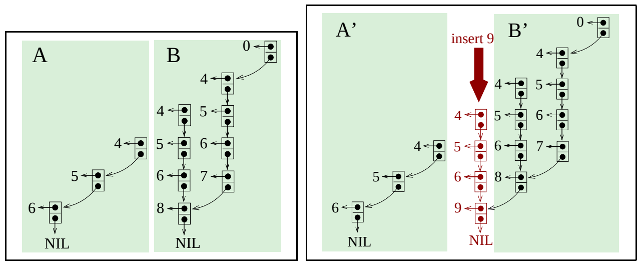

Figure 5: The accumulators for insertion sort with inputs $[6, 5, 4, 8, 7, 0]$ and $[6, 5, 4, \mathbf{9}, 8, 7, 0]$

The goal is to create the computation pictured on the right from the computation on the left. When using the adaptivity and memoization in combination, the result in box A will be re-used because of adaptivity, *i.e.*, A'=A. But box B will not be re-used when constructing box B', because the insertion of 9 into the accumulator will create a entirely new accumulator and no call to `iSort` will find its results in the memo. The issue is that memoization permits result re-use only when the arguments to the function match exactly.

As motivation for an $O(n)$-time solution note that the accumulators in boxes B and B' are very similar—the only difference is the key 9. Thus, if we encapsulate the computation pictured in box B as a stand-alone adaptive computation with accumulator $[4, 5, 6]$ and input list $[8, 7, 0]$ we can create the computation in box B' by re-using B, changing its accumulator to $[4, 5, 6, \mathbf{9}]$ and propagating this change.

[1]The bound is the same for the variant of insertion sort that inserts on the way up the recursion of `isort` instead of on the way down.

2.4 Adaptive Memoization.

Continuing on the insertion sort example, suppose that the results for the function `iSort` (Figure 4) are memoized based on just the input list and not on the accumulator. With this memoization policy, the result will be found in the memo when the call `iSort(`$[8, 7, 0], [4, 5, 6, \mathbf{9}]$`)` is performed. The returned result, $[0, 4, 5, 6, 7, 8]$, however, will be the result from before the input change, *i.e.*, that of the call `iSort(`$[8, 7, 0], [4, 5, 6]$`)` and will be incorrect. The key idea is that with adaptivity this is not a problem because the accumulator can be changed to $[4, 5, 6, \mathbf{9}]$ and the result can be updated with change propagation (see Section 5).

Adaptive memoization enables the result of a function call to be re-used by matching any subset of the arguments. When a result is re-used, the non-matched arguments will be changed and the result will be updated by change propagation. For this to work, the non-matched arguments must be stored inside modifiables.

Insertion sort demonstrates a general problem: the orthogonal adaptivity and memoization combination will generally be ineffective for algorithms that operate on some core data structure threaded through the computation. In such algorithms, an incremental input change can make some deep but small change to the core data structure forcing re-execution of a large number operations. In insertion sort, the core data structure is the accumulator list. Adaptive memoization will therefore be essential for making many interesting algorithms incremental.

3 Implementation

Building on our implementations of adaptivity [2] and selective memoization [4], we implemented the combination of adaptivity and memoization. We present an overview of this implementation and apply it to the map and insertion sort examples in Section 2. The dynamic semantics given in Section 4 presents a more precise definition of an implementation. Section 5 presents a more detailed treatment of insertion sort and Quicksort.

We study the orthogonal combination and adaptive memoization separately. To achieve constant-time overhead, our implementation relies on the representation of containment hierarchy of dynamic dependence graphs based on time-stamps, which we review first.

Dynamic dependence graphs and time stamps. To represent the containment hierarchy and a topological ordering of the dependence edges, the implementation uses time-stamps respecting the sequential execution order. Each read is assigned the time-interval of its execution and containment between reads is checked in constant time: a read r is contained in some other read r' if the time interval of r is contained in that of r'.

Since change propagation modifies the dependence structure of dynamic dependence graphs, the order of time-stamps must be maintained dynamically. The implementation therefore maintains the time stamps using the constant time Dietz-Sleator order-maintenance data structure that supports, creation, deletion, and comparison of time stamps in constant time [9].

Figure 6 shows the dynamic dependence graph for adaptive map, `amap`, using time-stamps instead of the explicit control edges as in Figure 2 (Figure 1 shows the code for `amap`). Each read (downward arrows between circular nodes) is contained in all the reads to its left. Time-intervals of the reads are shown as pairs. For example, the time-interval of

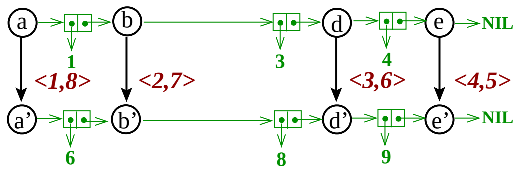

Figure 6: DDGs of `amap` with inputs $[1, 3, 4]$.

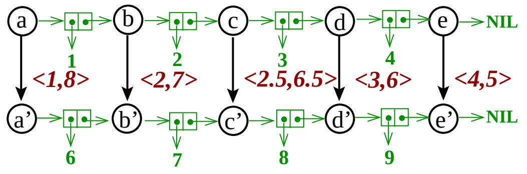

Figure 7: DDGs of `amap` with input $[1, \mathbf{2}, 3, 4]$.

the read (b, b') is $\langle 2, 7\rangle$ and that of (d, d') is $\langle 3, 6\rangle$ and indeed the interval $\langle 3, 6\rangle$ is contained in the interval $\langle 2, 7\rangle$.

3.1 The Orthogonal Combination

To combine adaptivity and memoization, we extend conventional memoization to support re-use of dependence graphs by remembering the dependence graph of a function call in addition to the result. We refer to this combination as the orthogonal combination because it does not change the semantics of memoization or adaptivity.

For correctness, the implementation must ensure that (1) no dependence graph (or result) is used more than once, and (2) the containment hierarchy is updated properly when a dependence graph is re-used. The first restriction is necessary because adaptivity requires that any two calls of a function have disjoint dynamic dependence graphs.

The implementation satisfies these two properties efficiently by (1) only allowing re-use of result that would otherwise be deleted by change propagation and (2) requiring that re-used dependence graphs do not conflict with the containment hierarchy of the current dependence graph. More concretely, change propagation maintains a *re-use interval* (r_s, r_e) that is initialized to the time-interval of the read currently being re-executed. A dependence graph with time interval (d_s, d_e) can only be re-used if its interval falls within the re-use interval, *i.e.*, $r_s < d_s$ and $d_e < r_e$ (a dependence graph has the given time interval if all of its reads are in that time interval). When the dependence graph is re-used, the re-use interval is moved past the dependence graph by setting $r_s := d_e$ and deleting all reads whose time-intervals fall within (r_s, d_s). When re-execution of a read completes, the remaining reads within the re-use interval are deleted.

The implementation remembers the dependence graph of a memoized result by storing the time-interval of the dependence graph for that result in the memo table. For example, the memo table of the computation in Figure 6 maps input `a` to the result $[6, 8, 9]$ consisting of the modifiables a', b', d', e', and the time interval $\langle 1, 8\rangle$. The time interval identifies the sub-graph of the current dynamic dependence graph that corresponds to the memoized call. Since a result can only be re-used if it is a subgraph of the dependence graph and if it falls within the current re-use interval, remembering the time-interval suffices.

To implement the orthogonal combination efficiently, we combine the implementations adaptivity [2] and selective memoization [4] and extend memo tables for storing time-intervals. Since the memo tables store time-intervals along

```
mfun m_insert (!k, (!h,?t)) =
  d = new modifiable
  read t with reader (fn vt =>
    case vt of
      NIL => write (d, CONS (k,emptyModlist))
    | CONS(hh,tt) =>
       if (k < hh) then write (d,CONS(k,t))
       else write (d, CONS(hh,m_insert (?k,(!hh,?tt)))
```

Figure 8: Pseudo code for `insert`.

with results, a result can be cached multiple times with different intervals. In this paper, we only consider applications that computes and caches any result no more a constant times. With this restriction, the overhead of the orthogonal combination is expected constant.

As an example of how the orthogonal combination works, execute the call `amap`$([1, 3, 4])$ and change the input by inserting the new key 2 by changing the modifiable b (the change is shown in Figure 3). Performing change propagation with this change on the dependence graph of Figure 6 will build the dependence graph in Figure 7 in expected constant time. Change propagation algorithm will re-execute the read (b, b') of Figure 6 after initializing the re-use interval to $\langle 2, 7\rangle$. Re-execution of this read will recursively call `amap` on c. The call will create the modifiable `c'`, read `c`, and call `amap` on d. The read of `c` will be time-stamped with $\langle 2.5, 6.5\rangle$ to fit between the intervals $\langle 2, 7\rangle$ and $\langle 3, 6\rangle$. Since the call `amap(d)` falls inside the re-use interval, it will be re-used. Since there are no more changes, change propagation will terminate updating the result in expected constant time. Note the only modifiable created during re-execution is `c'`—in contrast conventional change propagation re-creates the whole tail of the result as shown in Figure 3.

3.2 Adaptive Memoization

Adaptive memoization changes the semantics of memoization by allowing previous results to be re-used based on matching any subset of the arguments. For correctness, arguments that are not matched must be modifiables. For example, calls of the function $f(a, b)$ can be memoized and re-used when the values of a match regardless of b, as long as b is a modifiable.

To support adaptive memoization, the implementation encapsulates dependence graphs as stand-alone adaptive computations by making a local copy of each unmatched argument. The local copies of the unmatched arguments are designated as input to the memoized computation. When a result is re-used the unmatched arguments are connected to the corresponding local copies, the values of the local copies are changed to those of the unmatched arguments and change propagation is performed to update the re-used result. To implement adaptive memoization, we extend the implementation for the orthogonal combination by having memo tables remember the local copies for the dependence graphs.

As an example, Figure 8 shows the pseudo-code for the adaptively memoized version of `insert` of the insertion sort example (Figure 4). Function `m_insert` inserts a given key `k` to the list `t`. The argument `h` is the last inspected key and used for memoization only. The banged parameters, `k,h`, are matched (used for memo lookups), and the argument with the questions mark, `t`, is not matched (not used for memo look ups). Thus the memo table for `m_insert` maps `k` and `h` to a result and an adaptive computation consisting of a time interval and a local copy of `t`.

Figure 9 shows the dependence graph for the call

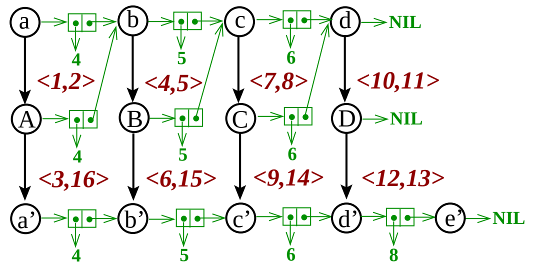

Figure 9: DDGs of `m_insert` $(8, [4, 5, 6], 0)$.

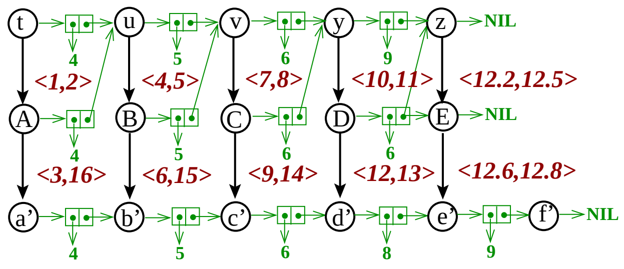

Figure 10: DDGs of `m_insert` $(8, (0, [4, 5, 6, \mathbf{9}]))$.

`minsert`$(8, (0, [4, 5, 6]))$ that inserts 8 to $[4, 5, 6]$. The input consists of the modifiables `a,b,c,d`. Since memoization does not match on the input list, the modifiables `a,b,c,d` are copied before being read; the modifiables `A,B,C,D` are the local copies. The reads between `a,b,c,d` and `A,B,C,D` copy values of the input modifiables to their local copies. Since `m_insert` reads only the local modifiables its dependence graph can be re-used by linking the unmatched inputs of the call to the local copies.

Adaptive memoization allows results re-use based on the content or the structure of the input rather than its identity. As an example of how this is useful consider performing the call `m_insert` $(8, (0, [4, 5, 6, \mathbf{9}]))$ after the call `m_insert` $(8, (0, [4, 5, 6]))$. Although the inputs $[4, 5, 6]$ and $[4, 5, 6, \mathbf{9}]$ are structurally similar, they may consist of different modifiables deeming conventional memoization ineffective—indeed this is the problem with insertion sort described in Section 2. Adaptive memoization can exploit the similarity between the two inputs by re-using a large part of the dependence graph for the call `m_insert` $(8, (0, [4, 5, 6]))$ as shown in Figure 10.

To see how adaptive memoization works, suppose the changed input consists of the new modifiables t, u, v, y, z as shown in Figure 10. Since the result for the call `m_insert` $(8, (0, [4, 5, 6, \mathbf{9}]))$ is memoized based only on $(8, 0)$ it will be found in the memo. The dependence graph with interval $\langle 3, 16\rangle$ will be re-used and `t` will be copied to `A`, creating a dependence from `t` to `A`. Copying will change the value of `A` (its tail now points to `u` instead of `b`) and the read (A, a') will be re-executed. The result will again be found in the memo and `u` will be copied to `B` and so on until the call with `z`, whose result will not be found in the memo because the key 9 has never been seen before. Thus a local copy for z will be created. The update will take linear time and will synchronize the old and the new computation so that the results are identical except for the newly inserted key. In the context of the insertion sort, this will suffice for synchronizing the computations before and after an insertion and updating the result in expected $O(n)$ time.

4 An Incremental Functional Language

We present a purely functional language, called IFL, that combines adaptivity and memoization. The language extends a product of the AFL language for adaptivity [2] and the MFL language for memoization [4] with support for adaptive memoization.

Our implementation of the IFL language closely follows the dynamic semantics of IFL. The main difference is that instead of using traces, like the dynamic semantics does, the implementation uses dynamic dependence graphs and memo tables. This is purely for efficiency reasons.

Selective memoization [4] enables the programmer to express the precise input-output dependences of a memoized function. To support adaptive memoization, we extend selective memoization with constructs that deem an input unmatched. An *unmatched* input is an input that is not used when performing a memo lookup. The IFL language supports introduction and elimination forms for unmatched input using *question* types.

The static semantics of IFL is a combination of the static semantics AFL and MFL extended with question types.

The dynamic semantics combines those of MFL and AFL and extends it to support adaptive memoization. The dynamic semantics of AFL is preserved but the semantics of MFL has been extended to support adaptive memoization and the limited form of memoization allowed here. One critical change is the omission of memo-tables. Instead, we extend the AFL traces with memoized computations. During change propagation, memo lookups inspect the trace of the currently re-executed read for a possible match.

4.1 Abstract Syntax.

The abstract syntax of IFL is given in Figure 11. Meta-variables x, y, z and their variants range over an unspecified set of variables, Meta-variables a, b, c and variants range over an unspecified set of resources. Meta variable l and variants range over a unspecified set of locations. Meta variable m ranges over a unspecified set of memo-function identifiers. Variables, resources, locations, memo-function identifiers are mutually disjoint. The syntax of IFL is restricted to "2/3-cps" or "named form" to streamline the presentation of the dynamic semantics.

The types of IFL includes the base type `int`, sums $\tau_1 + \tau_2$ and products $\tau_1 \times \tau_2$, bang $!\,\tau$ and question $?\,\tau$ types, the stable function types, $\tau_1 \xrightarrow{\mathsf{s}} \tau_2$, changeable function types $\tau_1 \xrightarrow{\mathsf{c}} \tau_2$, memoized-stable function types $\tau_1 \xrightarrow{\mathsf{ms}} \tau_2$, and memoized-changeable function types $\tau_1 \xrightarrow{\mathsf{mc}} \tau_2$. Extending IFL with recursive or polymorphic types presents no fundamental difficulties but omitted here for the sake of brevity.

The underlying type of a bang type $!\,\tau$ is required to be an indexable type. An *indexable type* accepts an injective *index* function into integers [4]. Operationally, the index function is used to determine equality. Any type can be made indexable by supplying an index function based on boxing or tagging [4]. Since this is completely standard and well understood, we do not have a separate category for indexable types to keep the language simple.

The abstract syntax is structured into *terms* and *expression*, which in turn are partitioned into *changeable* and *stable*. Terms evaluate independent of their contexts, as in ordinary functional programming, whereas expression are evaluated with respect to a memo table. Terms and expression divided into two categories, the *stable* and the *changeable*. The value of a stable expression or term is not sensitive to

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \texttt{int} \mid \texttt{!}\,\tau \mid \texttt{?}\,\tau \mid \\
 & & & \tau\ \texttt{mod} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \\
 & & & \tau_1 \xrightarrow{\mathsf{s}} \tau_2 \mid \tau_1 \xrightarrow{\mathsf{c}} \tau_2 \mid \tau_1 \xrightarrow{\mathsf{ms}} \tau_2 \mid \tau_1 \xrightarrow{\mathsf{mc}} \tau_2 \\
\textit{Values} & v & ::= & n \mid x \mid a \mid l \mid m \mid \texttt{!}\,v \mid \texttt{?}\,v \mid (v_1,v_2) \mid \\
 & & & \texttt{inl}_{\tau_1+\tau_2} v \mid \texttt{inr}_{\tau_1+\tau_2} v \mid \\
 & & & \texttt{s\_fun}\ f(x:\tau_1):\tau_2\ \texttt{is}\ t_s\ \texttt{end} \mid \\
 & & & \texttt{c\_fun}\ (x:\tau_1):\tau_2\ \texttt{is}\ t_c\ \texttt{end} \mid \\
 & & & \texttt{ms\_fun}_m\ f(a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_s\ \texttt{end} \\
 & & & \texttt{mc\_fun}_m\ f(a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_c\ \texttt{end} \\
\textit{Operators} & o & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots \\
\textit{St. Expr} & e_s & ::= & \texttt{return}(t_s) \mid \\
 & & & \texttt{let}\ a{:}\tau\ \texttt{be}\ t_s\ \texttt{in}\ e_s\ \texttt{end} \mid \\
 & & & \texttt{let !}\,x{:}\tau\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} \mid \\
 & & & \texttt{let ?}\,x{:}\tau\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} \mid \\
 & & & \texttt{let}\ a_1{:}\tau_1{\times}a_2{:}\tau_2\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} \mid \\
 & & & \texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\,(a_1{:}\tau_1) \Rightarrow e_s \\
 & & & \qquad\qquad \mid\ \texttt{inr}\,(a_2{:}\tau_2) \Rightarrow e_s \\
\textit{Ch. Expr} & e_c & ::= & \texttt{return}(t_c) \mid \\
 & & & \texttt{let}\ a{:}\tau\ \texttt{be}\ t_s\ \texttt{in}\ e_c\ \texttt{end} \mid \\
 & & & \texttt{let !}\,x{:}\tau\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} \mid \\
 & & & \texttt{let ?}\,x{:}\tau\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} \mid \\
 & & & \texttt{let}\ a_1{:}\tau_1{\times}a_2{:}\tau_2\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} \mid \\
 & & & \texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\,(a_1{:}\tau_1) \Rightarrow e_c \\
 & & & \qquad\qquad \mid\ \texttt{inr}\,(a_2{:}\tau_2) \Rightarrow e'_c \\
\textit{St. Terms} & t_s & ::= & v \mid o(v_1,\ldots,v_n) \mid \\
 & & & \texttt{ms\_fun}\ f\ (a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_s\ \texttt{end} \mid \\
 & & & \texttt{mc\_fun}\ f\ (a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_c\ \texttt{end} \mid \\
 & & & \texttt{s\_app}(v_1,v_2) \mid \texttt{ms\_app}(v_1,v_2) \mid \\
 & & & \texttt{let}\ x\ \texttt{be}\ t_s\ \texttt{in}\ t'_s\ \texttt{end} \mid \texttt{mod}_\tau\ t_c \mid \\
 & & & \texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\,(x_1{:}\tau_1) \Rightarrow t_s \\
 & & & \qquad\qquad \mid\ \texttt{inr}\,(x_2{:}\tau_2) \Rightarrow t'_s \\
\textit{Ch. Terms} & t_c & ::= & \texttt{write}(v) \mid \\
 & & & \texttt{c\_app}(v_1,v_2) \mid \texttt{mc\_app}(v_1,v_2) \mid \\
 & & & \texttt{let}\ x\ \texttt{be}\ t_s\ \texttt{in}\ t_c\ \texttt{end} \mid \\
 & & & \texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ t_c\ \texttt{end} \mid \\
 & & & \texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\,(x_1{:}\tau_1) \Rightarrow t_c \\
 & & & \qquad\qquad \mid\ \texttt{inr}\,(x_2{:}\tau_2) \Rightarrow t'_c
\end{array}
$$

Figure 11: The abstract syntax of IFL.

the modifications to the input, whereas the the value of a changeable expression or term may be affected by them.

Stable and Changeable Terms. Familiar mechanism of functional programming are embedded in IFL in the form of stable terms. Ordinary functions arise in IFL as stable functions. The body of a stable function must be a stable term; the application of a stable function is correspondingly stable. The stable term $\texttt{mod}_\tau\ t_c$ allocates a new modifiable reference whose value is determined by the changeable term t_c. Note that the modifiable itself is stable, even though its contents is subject to change.

Changeable terms are written in destination-passing style with an implicit target. The changeable term `write`(v) writes the value v into the target. The changeable term **`read`** v **`as`** x **`in`** t_c **`end`** binds the contents of the modifiable v to the variable x, then continues evaluation of t_c. A **`read`** is considered changeable because the contents of the modifiable on which it depends is subject to change. A changeable function itself is stable, but its body is changeable; correspondingly, the application of a changeable function is a changeable term. The sequential `let` construct allows for the inclusion of stable sub-computations in changeable mode. Case expressions with changeable branches are changeable.

Memoized stable and changeable functions are function whose bodies are stable or changeable expressions. As with stable and changeable functions, memoized functions are stable terms. Applications of memoized stable functions are stable and applications of memoized changeable functions are changeable.

Stable and Changeable Expression. Expression are evaluated in the context of a memo table and are divided into stable and changeable. Stable and changeable expressions are symmetric except for the body of the `return` construct. Stable terms are included in stable expressions, and changeable terms are included in changeable expressions via a `return`. The constructs except for `return` inspect the arguments of a function and express precise dependences between the input and the output of the function. The `return` returns a value based on the those parts of the argument that have been made available by the preceding constructs.

4.2 Static Semantics

The static semantics of the language combines the static semantics of AFL and MFL and extends them with question types. The extensions are relatively straightforward. In particular, the question types are symmetric to bang types of selective memoization [4]. The full type system is provided in the companion tech-report [3].

4.3 Dynamic Semantics

The dynamic semantics consists of four separate evaluation judgments corresponding to stable and changeable terms and stable and changeable expressions. All evaluation judgments take place with respect to a state $\sigma = (\alpha, \mu, \chi, \mathtt{T})$ consisting of a location store α, a memoized-function identifier store μ, a set of changed locations χ, and a re-use trace $\mathtt{T}$. The location store is where modifiables are allocated, the memoized-function identifier store dispenses unique identifiers for memoized functions that are used for memo lookups. The set of changed location contains the locations that has been changed since the previous execution. The re-use trace is the trace available for re-use by the memo functions.

The term evaluation judgments consists of changeable and stable evaluation forms. The judgment $\sigma, t_s \Downarrow^{\mathsf{s}} v, \sigma', \mathtt{T}_s$ states that evaluation of the stable term t_s with respect to the state σ yields value v, state σ', and the trace $\mathtt{T}_s$. The judgment $\sigma, l \leftarrow t_c \Downarrow^{\mathsf{c}} \sigma', \mathtt{T}_c$ states that evaluation of the changeable term t_c with respect to the state σ writes to destination l and yields the state σ', and the trace $\mathtt{T}_c$.

The expression evaluation judgments consists of changeable and stable evaluation forms. The judgment $\sigma, m{:}\beta, e_s \Downarrow^{\mathsf{s}} \sigma', v, \mathtt{T}_s$ states that the evaluation of the stable expression with respect to state σ, branch β, and memo identifier m yields the state σ', the value v and the trace $\mathtt{T}_s$. The judgment $\sigma, m{:}\beta, l \leftarrow e_c \Downarrow^{\mathsf{c}} \sigma', \mathtt{T}$ states that the evaluation of the changeable expression with respect to state σ, branch β, and memo identifier m writes to target l and yields the state σ' and the trace $\mathtt{T}$.

Evaluation of a term or an expression records its activity in a *trace*. Traces are divided into stable and changeable. The abstract syntax of traces is given by the following grammar, where $\mathtt{T}$ stands for a trace, $\mathtt{T}_s$ stands for a stable trace and $\mathtt{T}_c$ stands for a changeable trace.

$$\frac{\begin{array}{c}(v_1 = \mathtt{ms\_fun}_m\, f(a\!:\!\tau_1)\!:\!\tau_2\ \mathtt{is}\ e_s\ \mathtt{end})\\ \sigma, m\!:\!\varepsilon, [v_1/f, v_2/a]\ e_s \Downarrow^{\mathsf{s}} v, \sigma', \mathtt{T}_s\end{array}}{\sigma, \mathtt{ms\_app}(v_1, v_2)\ \Downarrow^{\mathsf{s}} v, \sigma', \mathtt{T}_s}\ \textbf{(st. memo apply)}$$

$$\frac{\begin{array}{c}(v_1 = \mathtt{mc\_fun}_m\, f(a\!:\!\tau_2)\!:\!\tau\ \mathtt{is}\ e_c\ \mathtt{end})\\ \sigma, m\!:\!\,!\,l, l \leftarrow [v_1/f, v_2/a]\ e_c \Downarrow^{\mathsf{c}} \sigma', \mathtt{T}\end{array}}{\sigma, l \leftarrow \mathtt{mc\_app}(v_1, v_2)\ \Downarrow^{\mathsf{c}}\ \sigma', \mathtt{T}}\ \textbf{(ch. memo apply)}$$

Figure 12: Stable and changeable memoized applications.

$$\begin{array}{lcl}\mathtt{T} & ::= & \mathtt{T}_s \mid \mathtt{T}_c\\ \mathtt{T}_s & ::= & \epsilon \mid \langle \mathtt{T}_c \rangle_{l:\tau} \mid \mathtt{T}_s\ ;\ \mathtt{T}_s \mid \{\,\mathtt{T}_s\,\}^{m:\beta}_{(v,(l_1,\ldots,l_n))}\\ \mathtt{T}_c & ::= & \mathtt{W}_\tau \mid R^{x.t}_l(\mathtt{T}_c) \mid \mathtt{T}_s\ ;\ \mathtt{T}_c \mid \{\,\mathtt{T}_c\,\}^{m:\beta}_{(l_1,\ldots,l_n)}\end{array}$$

When writing traces, we adopt the convention that "`;`" is right-associative.

A stable trace records the sequence of allocations of modifiables that arise during the evaluation of a stable term or expression. The trace $\langle \mathtt{T}_c \rangle_{l:\tau}$ records the allocation of the modifiable, l, its type, τ, and the trace of the initialization code for l. The trace $\mathtt{T}_s\ ;\ \mathtt{T}'_s$ results from evaluation of a `let` expression in stable mode, the first trace resulting from the bound expression, the second from its body. The trace $\{\,\mathtt{T}_s\,\}^{m:\beta}_{(v,(l_1,\ldots,l_n))}$ arises from the evaluation of a stable memoized function application; m is the identifier, β is the branch expressing the input-output dependences, the value v is the result of the evaluation, $l_1 \ldots l_n$ are the unmatched modifiables, and $\mathtt{T}_s$ is the trace of the body of the function.

A changeable trace has one of four forms. A write, $\mathtt{W}_\tau$, records the storage of a value of type τ in the target. A sequence $\mathtt{T}_s\ ;\ \mathtt{T}_c$ records the evaluation of a `let` expression in changeable mode, with $\mathtt{T}_s$ corresponding to the bound stable expression, and $\mathtt{T}_c$ corresponding to its body. A read $R^{x.t}_l(\mathtt{T}_c)$ trace specifies the location read, l, the context of use of its value, $x.e$, and the trace, $\mathtt{T}_c$, of the remainder of evaluation with the scope of that read. This records the dependency of the target on the value of the location read. The memoized changeable trace $\{\,\mathtt{T}_c\,\}^{m:\beta}_{(l_1,\ldots,l_n)}$ arises from the evaluation of a changeable memoized function; m is the identifier, β is the branch expressing the input-output dependences, $l_1 \ldots l_n$ are the unmatched modifiables, and $\mathtt{T}_c$ is the trace of the body of the function. Since changeable function write their result to the store, the trace has no result value.

The dynamic dependency graph and the memo table described in Section 3 may be seen as an efficient representation of traces. Time stamps may be assigned to each read and write operation in the trace in left-to-right order. These correspond to the time stamps in the dynamic dependence representation. The containment hierarchy is directly represented by the structure of the trace. Specifically, the trace $\mathtt{T}_c$ (and any read in $\mathtt{T}_c$) is contained within the read trace $R^{x.t}_l(\mathtt{T}_c)$. Memo tables represent the traces of the form $\{\,\mathtt{T}_s\,\}^{m:\beta}_{(v,(l_1,\ldots,l_n))}$ and $\{\,\mathtt{T}_c\,\}^{m:\beta}_{(l_1,\ldots,l_n)}$. The identifier m identifies a memo table, the branch β is the lookup key, v is the result being stored if any, and the trace $\mathtt{T}_c$ or $\mathtt{T}_s$ along with the unmatched modifiables $l_1, \ldots, l_n$ is an encapsulated adaptive computation with inputs $l_1, \ldots, l_n$. An explicit result is not stored for memoized changeable functions because they write to their target which must match for the memo to be re-used.

Due to space restrictions the complete dynamic semantics is provided in the companion tech-report [3]. In the rest of the section, we briefly walk through some the more interesting rules.

Term evaluation. Figure 12 shows the memoized stable and changeable function applications. Memoized changeable and stable applications evaluate some expression in the context of an identifier m and a branch β. As in selective memoization, the branch collects the precise dependencies between the input and the output. For stable applications the branch starts out empty (ε). For changeable applications it is initialized to the target—since a changeable expressions writes to its target, the target must be identical for the "result" to be re-used.

Expression Evaluation. Expression evaluation takes place in the context of a re-use trace. The incremental evaluation constructs (`let!`, `let?`, *etc.*) create a branch, denoted β. The branch and the identifier m is used by the `return` construct to lookup the re-use trace for a match. If a match is found, the result is returned and the body of `return` is skipped. Otherwise, the body of the return is executed.

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma\\ ([l_1, \ldots, l_n], \beta') = \mathtt{split}\,(\beta)\\ \mathtt{find}\,(m\!:\!\beta', \mathtt{T}) = \mathtt{NONE}\\ \alpha' = \alpha[l'_1 \mapsto \alpha[l_1], \ldots, l'_n \mapsto \alpha[l_n]],\\ \text{where } l'_1 \notin \mathrm{dom}(\alpha), \ldots, l'_n \notin \mathrm{dom}(\alpha), l'_i \neq l'_j\\ \sigma' = (\alpha', \mu, \chi, T)\\ \sigma', [l'_1/l_1, \ldots l'_n/l_n]t_s\ \Downarrow^{\mathsf{s}} v, \sigma'', \mathtt{T}_s\\ \mathtt{T}'_s = \langle R^{x.\mathtt{write}(x)}_{l_1} \mathtt{W}_{\tau_1} \rangle_{l'_1:\tau_1} \cdots \langle R^{x.\mathtt{write}(x)}_{l_n} \mathtt{W}_{\tau_n} \rangle_{l'_n:\tau_n}\end{array}}{\sigma, m\!:\!\beta, \mathtt{return}(t_s) \Downarrow^{\mathsf{s}} v, \sigma'', \left(\mathtt{T}'_s\,; \{\,\mathtt{T}_s\,\}^{m:\beta}_{(v,(l'_1,\ldots,l'_n))}\right)}\ (\times)$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma\\ ([l_1, \ldots, l_n], \beta') = \mathtt{split}\,(\beta)\\ \mathtt{find}\,(m\!:\!\beta', \mathtt{T}) = \mathtt{SOME}(\{\,\mathtt{T}_s\,\}^{m:\beta'}_{(v,(l'_1,\ldots l'_n))}, \mathtt{T}')\\ \alpha' = \alpha[l'_1 \mapsto \alpha[l_1], \ldots, l'_n \mapsto \alpha[l_n]]\\ \mathtt{T}'_s = \langle R^{x.\mathtt{write}(x)}_{l_1} \mathtt{W}_{\tau_1} \rangle_{l'_1:\tau_1} \cdots \langle R^{x.\mathtt{write}(x)}_{l_n} \mathtt{W}_{\tau_n} \rangle_{l'_n:\tau_n}\\ (\chi' = \chi \cup \{l'_1, \ldots, l'_n\})\\ \sigma' = (\alpha', \mu, \chi', \mathtt{T}')\end{array}}{\sigma, m\!:\!\beta, \mathtt{return}(t_s) \Downarrow^{\mathsf{s}} v, \sigma', \left(\mathtt{T}'_s\,; \{\,\mathtt{T}_s\,\}^{m:\beta}_{(v,(l'_1,\ldots,l'_n))}\right)}\ (\checkmark)$$

$$\frac{\sigma, m\!:\,!\,v \cdot \beta, [v/x]e_s \quad \Downarrow^{\mathsf{s}} \quad v', \sigma', \mathtt{T}_s}{\sigma, m\!:\!\beta, \mathtt{let}\,!\,x : \tau\ \mathtt{be}\ !\,v\ \mathtt{in}\ e_s\ \mathtt{end} \Downarrow^{\mathsf{s}} v', \sigma', \mathtt{T}_s}\ \textbf{(let!)}$$

$$\frac{\sigma, m\!:\,?v \cdot \beta, [v/x]e_s \quad \Downarrow^{\mathsf{s}} \quad v', \sigma', \mathtt{T}_s}{\sigma, m\!:\!\beta, \mathtt{let}\,?\,x : \tau\ \mathtt{be}\ ?\,v\ \mathtt{in}\ e_s\ \mathtt{end} \Downarrow^{\mathsf{s}} v', \sigma', \mathtt{T}_s}\ \textbf{(let?)}$$

Figure 13: Sample stable-expression evaluation.

Figure 13 shows some sample stable expression evaluation rules. Changeable expressions are evaluated similarly except that a target is threaded through the changeable expressions. The evaluation $\sigma, m\!:\!\beta, e_s \Downarrow^{\mathsf{s}} v, \sigma', \mathtt{T}_s$ states that the evaluation of stable expressions e_s in the context of the state σ, with memo function identifier m and branch β yields the value v, the state σ' and the trace $\mathtt{T}_s$.

Adaptive memoization permits result re-use based on a subset of the values that the result of a function depends for. The unmatched dependences are expressed by the `let?` construct which adds them to the branch as such. The type system ensures that all unmatched arguments are modifiables. During a memo lookup, unmatched modifiables are separated from other dependences by the `split`$(\cdot)$ that splits a branch into a list of the unmatched modifiables and a branch β'. In Figure 13 the top two rules are the memo lookups. Unmatched modifiables are denoted as l_i's. The `find`$(m:\beta',\mathtt{T})$ performs the memo lookup with the filtered branch β' and the identifier m in the re-use trace `T`. If a match is not found `find` returns `NONE`, otherwise it returns the trace of the memoized function and the tail of the re-use trace following the match. If no results are found (the top rule), then the body of the `return` is evaluated after substituting unmatched modifiables with fresh modifiables. The trace returned by the evaluation is encapsulated by the branch, the identifier, the result, and returned along with a copy trace for copying the unmatched modifiables. If the result is found in the memo (the second rule from the top), the body of `return` is skipped and the found trace is re-used after copying the unmatched modifiables to the local copies.

5 Applications

We describe how to make Insertion Sort and Quick Sort incremental under insertions and deletions to the input. We prove strong performance bounds. For insertion sort, we show that an insertion or deletion is handled in expected-case $\Theta(n)$ time with adaptive memoization. For Quicksort, we consider insertions and deletions at random locations and show an expected $\Theta(\log^2 n)$ bound by using the orthogonal combination. We improve this to expected $\Theta(\log n)$ by using adaptive memoization. The expectations are over internal randomness for hashing used in memo tables. For Quicksort the expectation is also over all permutations of the input, as usual.

We present the code for the applications by using an extended version of the IFL language that support lists. For brevity, we also use pattern matching on the bang and question mark types, and do not apply the named-form restriction.

Both algorithms operate on modifiable lists defined as

```
datatype 'a mlist = NIL | CONS ('a * ('a mlist) mod)
type 'a modlist = ('a mlist) mod.
```

Due to space restrictions the proofs of the theorems in this section are provided in the companion tech-report [3].

5.1 Incremental Insertion Sort

Figure 14 shows the code for incremental insertion sort. The function `iSort` inserts the keys in the input list `l` into an initially empty accumulator `a`. As indicated by the `!` and `?`, the result is memoized based on the input list and adaptively memoized on the accumulator. This means that a result will be found in the memo when the input lists are identical even though the accumulators are not. The function `insert` inserts a given key `i` into the list `t`. It is memoized based on `i` and the previously inspected key `h`, and adaptively memoized with respect to `t`. This ensures that the same result will be returned as long as the content of the lists (`t`'s) are the same even if they contain different cons cells.

```
insert:  (!int * (!int*?int modlist))->int modlist
ms_fun insert (!i,(!h,?t)) =
  return mod (
    read t as vt in
      case vt of
        NIL => CONS (i,t)
      | CONS(hh,tt) =>
        if (i < hh) then
          CONS(i,t)
        else
          CONS(hh, ms_app(insert, (!i,(!hh,?tt))))
    end)

mc_fun iSort (!l:int modlist,?a:int modlist) =
  return
    read l as vl in
      case vl of
        NIL => write a
      | CONS(h,t) =>
        let aa = ms_app (insert (!h, (!h,?a))) in
          mc_app(iSort, (!t, ?aa))
        end
    end

s_fun insSort (l:int modlist):(int modlist) mod =
  mod (mc_app(iSort,(!l,?(mod (write NIL)))))
```

Figure 14: Insertion sort with adaptive memoization.

As discussed in Section 2 without using adaptive memoization, insertion takes $\Theta(n^2)$ time even with the orthogonal combination of adaptivity and memoization. Adaptive memoization improves performance to expected $\Theta(n)$ time.

Theorem 1
Insertion sort (shown in Figure 14) updates its result in expected $\Theta(n)$ time when its input is changed by an insertion or deletion anywhere in the list.

5.2 Incremental Quicksort

We consider two versions of Quicksort using the orthogonal combination and adaptive memoization. The table below compares their performance for a single insertion or deletion at the beginning, at the end, and at a random location in the list to the performance with memoization or adaptivity only. All bounds are expected case with expectations taken over all possible permutations of the input; for random insertions, expectations are taken over all possible locations in the input with uniform probability.

	beginning	end	random
Adaptive Memo	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Orthogonal	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(\log^2 n)$
Memoized	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Adaptive	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(n \log n)$

Quicksort with Orthogonal Combination. Figure 15 shows the code for incremental Quicksort using the orthogonal combination. The code avoids appends by using an accumulator and is very similar to the adaptive Quicksort analyzed in previous work [2]. The only difference is that the filter function `fil` is memoized based on the pivot, the function for filtering, and the input list.

Theorem 2
The Quicksort with the orthogonal combination takes expected $\Theta(n \log n)$ time for insertions at the head of the input, expected $\Theta(\log n)$ time for insertions at the end of the

```
fil:(!int*!(int*int->bool)*!int modlist)->int modlist
ms_fun fil (!p, !f, !l) =
  return mod (
    read l as ll in
      case ll of
        NIL => write NIL
        CONS(h,t) =>
        if (f h) then
          write CONS(h,ms_app(fil, (!p,!f,!t)))
        else
          read (ms_app(fil, (!p,!f,!t))) as tt in
            write tt
          end
    end)

c_fun qs(l:int modlist, rest:int mlist) =
  read l as vl in
    case vl of
      NIL => write rest
    | CONS(h,t) =>
      let
        val g = ms_app(fil, (!h, !(fn x => x > h),!t))
        val gs = mod (c_app (qs, (g,rest)))
        val s = ms_app(fil, (!h, !(fn x => x < h),!t))
      in
        c_app (qs, (s,CONS(h,gs)))
      end
  end

s_fun qsort (l:int modlist):int modlist =
  mod (c_app (qs, (l,NIL)))
```

Figure 15: Quicksort with the orthogonal combination.

```
fil:(!int*!(int*int->bool)*!int modlist)->int modlist
ms_fun fil (!p,!f,(!h,?t)) =
  return mod (
    read t as vt in
      case vt of
        NIL => write NIL
        CONS(hh,tt) =>
        if (f hh) then
          write CONS(hh,ms_app(fil, (!p,!f,(!hh,!tt))))
        else
          read (ms_app(fil, (!p,!f,(!hh,!tt)))) as vtt in
            write vtt
          end
    end)

c_fun qs(l:int modlist,rest:int mlist) = ...
```

Figure 16: Quicksort with adaptive memoization.

input, and expected $\Theta(\log^2 n)$ time for insertions at a (uniformly) randomly chosen position. Expectations are over all permutations of the input list. The same bounds apply to deletions.

Quicksort with Adaptive Memoization. Figure 16 shows the code for Quicksort with adaptive memoization. The difference between this version and the version using orthogonal combination is that `fil` is not memoized based on the input list. It now takes a separate head and tail and is memoized based only on the head. This ensures that `fil` generates the same output when its input consists of keys that are a subset of the previous input—even if the new input consists of different cons cells.

Theorem 3
The adaptively memoized Quicksort takes expected $\Theta(n)$ time for insertions at the head of the input, expected $\Theta(\log n)$ time for insertions at the end of the input, and expected $\Theta(\log n)$ time for insertions at a uniformly randomly chosen position. The expectations are over permutations of the input list. The same bounds apply to deletions.

References

[1] Martin Abadi, Butler W. Lampson, and Jean-Jacques Levy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

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

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive memoization. Technical report, Department of Computer Science, Carnegie Mellon University, 2003. Available at http://www.cs.cmu.edu/~umut/research/amemo.pdf.

[4] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.

[5] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *To Appear in ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[6] Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. Separating structure from data in dynamic trees. Technical Report CMU-CS-03-189, Department of Computer Science, Carnegie Mellon University, 2003.

[7] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

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

[9] 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.

[10] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. *ACM SIGPLAN Notices*, 35(5):311–320, 2000.

[11] Yanhong A. Liu, Scott Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

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

[13] D. Michie. 'memo' functions and machine learning. *Nature*, 218:19–22, 1968.

[14] William Pugh. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 269–276. ACM Press, 1988.

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

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

[17] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 502–510, January 1993.

[18] Thomas Reps. *Generating Language-Based Environments*. PhD thesis, Department of Computer Science, Cornell University, August 1982.

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