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

Memoization may be viewed as a mechanism for re-using a computation—if a function is re-applied to the same argument we may re-use the previous computation to determine the result, rather than perform it again. Conventional memoization is *accurate* in the sense that it only permits re-use when a computation will be precisely the same as one that has already been performed. But in many cases a computation may be largely, though not entirely, the same as one that has been previously carried out for a slightly different input. The previous computation may be re-used by permitting *inaccurate* memoization, and restoring accurancy by *adapting* the result to the variant input. This technique, which we call *adaptive memoization*, greatly increases the effectiveness of both memoization and adaptivity alone (or in orthogonal combination) for incremental computation. In particular we obtain an incremental version of Quicksort that adjusts its output in logarithmic expected time to insertions and deletions at random positions in the input, and an incremental version of Insertion Sort that adjusts its output in linear time to an insertion or deletion anywhere in its input. These results are the best possible for these algorithms, and rely crucially on adaptive memoization.

1 Introduction

Incremental computation is the ability to update the result of an algorithm as a result of a "small" change to its input, ideally in less time than would be required by a complete re-execution. The key to providing incrementality is to permit *re-use* of parts of the previous computation when revising the output in response to changes to the input. There are two general methods for achieving result re-use: memoization [7, 13, 14, 1, 16, 11, 3] and adaptivity [2, 4]. Memoization relies on remembering the results of function calls so that the previous result may be re-used when the function is called again with identical arguments. Adaptivity relies on recording the dynamic dependencies of computations on data values so that when these values change the affected computations can be re-executed to restore the result.

In earlier work on incremental computation [2, 3] we observed that adaptivity and memoization are, in a sense, duals, each addressing a separate aspect of incrementality. When combined "orthogonally" these two methods can be used to obtain incremental versions of some algorithms, but we have found that in many cases a more subtle combination of these methods is required. The fundamental issue is that conventional memoization only allows re-use of *accurate* results in the sense that the result of a function call can only be re-used in place of a function call that will evaluate to the *same* result. This limits the effectiveness of memoization by preventing the result of a function call to be re-used when computing a similar, but slightly different, result.

We introduce a new technique, called *adaptive memoization*, that permits re-use of *inaccurate* results. In the case of multi-argument functions this is achieved by a designating a subset of the arguments to be those for which an exact match is required to trigger memoization. If that subset includes all of the arguments, then this is just standard memoization, but if the subset is proper, then the re-used result may not be correct when all arguments are considered. But accuracy may be restored if the recovered computation is required to be adaptive in the remaining arguments, for then we may simply change those arguments to their new values and use adaptivity to recover the correct result.

As examples, we consider Quicksort and Insertion Sort. We show that Quicksort handles its insertions/deletions anywhere in the list in expected $O(\log n)$ time (expectation is over all possible positions of insertion or deletion). For insertion sort, we show expected $O(n)$ bound for any insertion or deletion. These results are within an expected constant factor of the optimal for these algorithms and crucially rely on inaccurate result re-use.[1] Besides the examples to be presented below, we also have used these methods to build efficient incremental algorithms in computational geometry, including kinetic data structures [6] and line-sweep techniques [5]. In general we believe the methods are very broadly applicable.

2 Background and Related Work

2.1 Memoization

Memoization [7, 14, 13] 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

*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.

[1] The expected constant factor overhead is due to the maintenance of memo tables using hashing.

```
type 'a list = nil
             | cons ('a*'a list)

map5:  int list -> int list
fun map5 l =

 case l of
   nil => nil
 | cons(h,t) =>
     cons(h+5,map5 t)
```

```
type 'a list = nil
             | cons ('a*'a list)

m_map5:  !(int list) -> int list
mfun m_map5 (!l) =

  case l of
    nil => nil
  | cons(h,t) =>
      cons(h+5,m_map5 !t)
```

```
type 'a mlist = NIL
              | CONS ('a*('a mlist) modref)
type 'a modlist = 'a mlist modref
a_map5:  int modlist -> int modlist
fun a_map5 l =
  mod (read l (fn vl =>
    case vl of
      NIL => write(NIL)
    | CONS(h,t) =>
        write(CONS(h+5,a_map5 t))))
```

Figure 1: The code for standard (left), memoized (middle), and adaptive (right) `map5`.

divide-and-conquer algorithms using so-called stable decompositions can be made incremental efficiently by using memoization. Liu, Stoller, and Teitelbaum [12] presented systematic techniques for developing incremental programs using memoization. Their techniques automatically determine what results should be memoized and use transformations to make a standard program incremental.

Since in general the result of a function call may not depend on all its arguments, caching results based on precise input-output dependences can dramatically improve result re-use. Several techniques for identifying precise input-output dependences have been proposed by Abadi, Lampson, and Levy [1], by Heydon, Levin, and Yu [11], and by the authors [3].

A common characteristics of all previous work on memoization is that they provide for re-use of accurate result. The result of a function call can only be re-used in place of a call only if their results are identical. To the best of our knowledge no previous work enables re-use of inaccurate results as we do in this paper.

2.2 Dependence Graph Techniques

Static dependence graph techniques for incremental computation were introduced by Demers, Reps, and Teitelbaum [9, 18] and have been successfully applied to many applications [17]. Static dependence graphs represent data dependences in a computation in such a way that when an input is changed, all data that depends on the change can be updated by performing a change propagation on the graph. The key limitation of static dependence graphs is that the dependence structure remains the same during change propagation. As Pugh points out [15] this limits the kinds of applications that can be made incremental using static dependence graphs.

To overcome the limitations of static dependence graphs, we introduced *dynamic dependence graphs* [2]. The key difference between static and dynamic dependence graphs is that with dynamic dependence graphs, change propagation updates the dependence structure according to the input change. Dynamic dependence graphs can therefore be used to make any purely functional program incremental.

Dynamic dependence graphs yield 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 [4]. 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 [20].

2.3 Adaptivity

In earlier work [2], where we introduce dynamic dependence graphs, we also present language facilities for writing what we call *adaptive* programs that update their output when their input changes. Since the rest of this paper builds upon adaptivity techniques, we present a short overview.

The language facilities are based 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. Modifiable reference are created via the `mod` construct, their values are read via `read` construct, and are written to via `write` construct. 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; this modifiable is called the *target* of the read.

As an adaptive program executes, it implicitly 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. To obtain good performance, it is crucial that the control dependences are represented efficiently. We do this by using time stamping each edge with the time interval in which it executes and maintaining the time stamps using the constant-time order maintenance data structure of Dietz and Sleator [10].

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 according to their time stamps. Re-executing a reader re-establishes the relationship between its source and target by updating the value of the target, which can affect 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 dependence structure of can change dramatically during change propagation.

3 Combining Memoization and Dynamic Dependence Graphs

This section discusses the limitations of memoization and dynamic dependence graphs when used in isolation and shows how they can be combined to obtain a general purpose technique for incremental computation. We consider two different combinations. The *orthogonal combination* combines memoization and dynamic dependence graphs without changing their semantics. *Adaptive memoization* combines memoization and dynamic dependence graph integrally to provide for re-use of inaccurate results.

The *orthogonal combination* is a straightforward extension of our previous work [2, 3]. It relies on memoizing results along with their dynamic dependence graphs. When a memo match occurs both the result and the dynamic dependence graph are re-used—the result is returned and the dependence graph is linked into the current context. Although the orthogonal combination can be effective for certain simple program, it remains ineffective in general, because, like conventional memoization, it only allows re-use on accurate result. In the case of insertion sort, for example, the orthogonal combination does not improve the asymptotic performance of an incremental version over a a from-scratch re-execution. We describe this combination anyway both as motivation and as a step towards the more complex integral combination that we call adaptive memoization.

The *integral combination* or *adaptive memoization* provides for re-use of inaccurate results in addition to accurate results (orthogonal combination is a special case of adaptive memoization). The key idea is to memoize *adaptive computations* [2] along with their arguments so that computations can not only be re-used but also adapted according to different arguments. As we show, inaccurate result re-use is critical to obtain efficient incremental programs. Indeed adaptive memoization dramatically improves the performance incremental insertion sort to expected $O(n)$ and that of Quicksort to $O(\log n)$ from $O(n^2)$ and $O(\log^2 n)$ with the orthogonal combination.

In the remainder of this section we review the limitations of adaptivity and memoization considered in isolation by considering a simple version of the `map` function. The orthogonal combination overcomes these limitations in the case of `map`, but for insertion sort even this is not sufficient to obtain an efficient incremental version. By examining the source of the difficulty we arrive at the integrated combination, called adaptive memoization.

Section 4 formalizes adaptive memoization and Section 5 describes how it is implemented. Section 6 analyses the performance of insertion sort. Section 6 presents Quicksort as another example and shows how adaptive memoization yields an incremental Quicksort that is optimal for any insertion or deletion.

3.1 Limitations of Memoization and Adaptivity

Consider the simple function, `map5`, given in Figure 1, which adds 5 to every element of a list. The memoized version, `m_map5`, memoizes the result for each recursive call. The code is based on our selective memoization techniques [3]. The `mfun` keyword indicates that `m_map5` is memoized; and assigning its input list a bang type (`!`) indicates that memoization is based on the input list. This means that if the function is ever called with the same input list, then the result will be re-used. For simplicity, we use pattern matching to accesses underlying values of banged types. The adaptive version, `a_map5`, is based on our adaptive functional programming techniques [2]. The transformation from `map5` to `a_map5` is relatively straightforward and involves changing the input list type to a modifiable list and then updating the body of the code by using the `mod`, `read`, and `write` constructs.

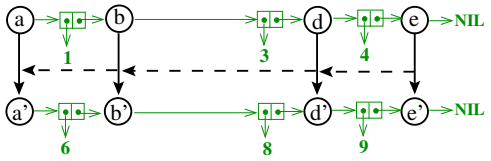

Figure 2: DDG for `a_map5` on input $[1, 3, 4]$

Consider evaluating the memoized version `m_map5` with an input list l and now with a second list $l' = k :: l$ that has been obtained from l by inserting the new key k at the head. Since calls to `m_map5` are memoized, the result will be found in the memo at the second recursive call (with l) and the result will be updated in constant time. Suppose, instead, that l is changed by adding a new key at the very end. After this change, none of the recursive calls to `m_map5` will find their result in the memo because the input to all recursive calls is now a different list. It will therefore take linear time to update the output. Thus, in general, `m_map5` will re-build the list up to the point of insertion and will take linear time on average. As this example illustrates, memoization is good in handling *shallow* changes that affect some function call close to the roots of the call tree of the evaluation (*e.g.*, insertion at the head of the list). Memoization performs poorly for changes that affect calls *deep* in the call tree (*e.g.*, insertion at the end of the list).

Consider evaluating the adaptive version `a_map5` with an input list $[1, 3, 4]$. This evaluation will be represented using the dependence graph shown in Figure 2. Each circle corresponds to a modifiable and the edges between modifiables corresponds to reads of modifiables. Cons cells are lightly shaded (or colored green). Each read is contained in a read to the left of it, as determined by recursive calls. The containment edges are shown with horizontal and dashed edges. Consider changing the input by inserting the key 2 at the second position by creating a new modifiable `c` and a new cons cell holding the value 2. Figure 3 show this change and the dynamic dependence graph after a change propagation. Change propagation updates the output by recursively computing the tail of the list. The deleted edges and nodes are shown with faded dashed lines. Since change propagation recomputes the result for the tail of the list following the insertion, an insertion at the very end of the list, a deep change, requires constant time; a change at the head of the list, a shallow change, requires linear time. Thus, like memoization, dynamic dependence graphs will take linear time in general.

3.2 Orthogonal Combination

The `map5` example of the previous section illustrates the problem with memoization and dynamic dependence graphs when applied in isolation. They both perform well under particular kinds of input changes: memoization performs well for shallow changes, and dynamic dependence graph perform well for deep changes. For a large class of input changes, that affect some call in the middle of the call tree, both techniques can, and typically do, fail to improve performance asymptotically over a from-scratch re-execution. In

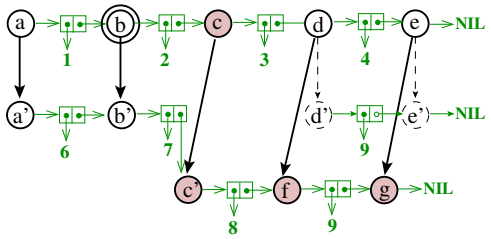

Figure 3: DDG for `a_map5` after changing input to $[1, \mathbf{2}, 3, 4]$.

```
am_map5:  !(int modlist) -> int modlist
mfun am_map5 (!l) =
  mod (read l (fn vl =>
    case vl of
      NIL => write(NIL)
    | CONS(h,t) =>
        write(CONS(h+5,am_map5 !t))))
```

Figure 4: The code for memoized and adaptive `map5`.

the case of `map5`, for example, insertions around the middle of the list will take linear time with both techniques. We now present an orthogonal combination of memoization and dynamic dependence graphs that solves certain problems, and the `map5` example in particular, in constant time.

The code for the `map5` function using the orthogonal combination is shown in Figure 4. The code is written by memoizing the adaptive version `a_map5` (Figure 1) using selective memoization techniques. The bang in front of the parameter to `am_map5` indicates that memoization is based on the input list `l`. A memo match will occur if `am_map5` is called with the same list again.

The `am_map5` function based on the orthogonal combination handles any insertion/deletion to the input in constant time. As an example, consider evaluating `am_map5` with the input list $[1, 3, 4]$ and changing the input to $[1, \mathbf{2}, 3, 4]$ by inserting 2. This change creates a new modifiable `c` and a new cons cell holding the value 2 and making the modifiable `b` point to it. Now, performing a change propagation will update the output by inserting the key 7 to the output and performing a recursive call to `am_map5`. Since the modifiable list `d` has been previously computed, a memo match will occur and the result and the dependence graph for the recursive call will be re-used from the memo. Change propagation (adpativity), along with memoization, will therefore update the output in constant time. The dynamic dependence graph after the update is shown in Figure 5. Note that the only difference between the revised dynamic dependence graph (Figure 5) and the original (Figure 2) are the two new nodes `c` and `c'` and the edge between them.

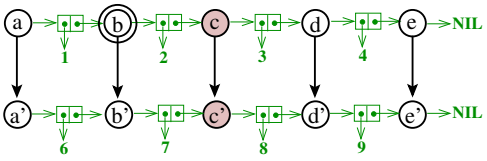

Figure 5: DDG for `am_map5` after inserting 2.

3.3 Integral Combination: Adaptive Memoization

Although the orthogonal combination yields good performance for certain applications, it nevertheless remains ineffective in general. This section uses insertion sort as an example to demonstrate the limitations of the orthogonal combination and illustrates how adaptive memoization can be used to overcome these limitations.

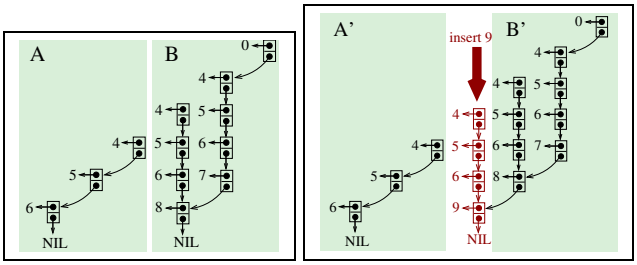

Figure 7: The accumulators for insertion sort with inputs $[6, 5, 4, 8, 7, 0]$ and $[6, 5, 4, \mathbf{9}, 8, 7, 0]$

Figure 6 shows the standard accumulator-based code for insertion sort. The function `iSort`, iterates over the input list `l`, inserting each key into the accumulator `a`. As a concrete example consider sorting the list $[6, 5, 4, 8, 7, 0]$ using insertion sort. The left box in Figure 7 shows the accumulators created during the evaluation. Each column represents an accumulator with time advancing from left to right. An accumulator is obtained from the preceding accumulator by inserting the next key from the input. Since `insert` recreates the accumulator up to the position where the key is placed and re-uses the tail, some tails are shared—curved arrows show such sharing. For example the first column on the left is the accumulator $[6]$, the second column is $[5, 6]$, the third column is $[4, 5, 6]$ *etc.*

Consider now evaluation of insertion Sort on the input $[6, 5, 4, \mathbf{9}, 8, 7, 0]$ obtained from the original list by inserting 9. The box on the right in Figure 7 shows the accumulators for this input. To compare the two evaluations, divide the computation into two boxes, A, B, and A', B', corresponding to the parts before and after the call to `insert` where the newly key 9 is inserted to the accumulator. Note that box A and A' are identical in both computations. Box B and B' are slightly different because the accumulators in box B' end with the new key 9.

Consider a version of insertion sort that uses the orthogonal combination described in Section 3.2. Consider evaluating insertion sort on $[6, 5, 4, 8, 7, 0]$ and then changing the input to $[6, 5, 4, \mathbf{9}, 8, 7, 0]$ by inserting 9. Performing a change propagation after this change will create the computation shown on the right in Figure 7 from the computation on the left. During change propagation, computation in box A will be re-used because none of the calls to `iSort` before key 9 is affected by the change (*i.e.*, A'=A)—more precisely change propagation will skip over box A because none of the modifiables read will be changed. The function `iSort` however will be evaluated when 9 is considered and 9 will be inserted into the accumulator creating the new accumulator $[4, 5, 6, 9]$. The thick arrow in Figure 7 points to this new accumulator. Now since this accumulator has never been seen before, the recursive calls to `iSort` will not find their result in the memo and box B' will be recomputed from scratch. Thus the orthogonal combination of memoization and dynamic dependence graphs fails to provide an asymp-

```
insert:int * int list -> int list
fun insert (k,a) =

  case a of
    nil => cons (p,nil)
  | cons(h,t) =>
      if (k < h) then
        cons(k,t)
      else
        cons(h,insert(k,t))


iSort:  int list * int list -> int list
fun iSort (l,a) =

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


insSort:  int modlist -> int modlist
fun insSort (l) =
  iSort (l,[])
```

```
insert:  (!int * ?(int modlist) * !int)->int modlist
mfun m_insert (!k,?a,!v) =
  mod (read a as va in
    case va of
      NIL => write (CONS (k,a))
    | CONS(h,t) =>
      if (k < h) then
        write (CONS(k,t))
      else
        write (CONS(h, m_insert (!k,?t,!v)))
  end)

m_iSort:!(int modlist) * ?(int modlist) -> int modlist
mfun m_iSort (!l,?a) =
  read l as vl in
    case vl of
      NIL => write a
    | CONS(h,t) =>iSort (!t, ?(m_insert (!h, ?a, !h)))
  end

m_insSort:  int modlist -> int modlist
fun m_insSort (l) =
  mod (m_iSort (!l,?(mod (write NIL))))
```

Figure 6: Standard insertion sort (left) and its adaptively memoized version (right).

totic improvement over re-execution, when averaging over all insertion positions.

To see how we can do better, note that boxes B and B' (Figure 7) are very similar. In particular the only structural difference between the boxes B and B' are the links to cons cell containing 9. Thus if we can re-use the computation in box B and change the first accumulator by inserting 9 and perform a change propagation, we can hope to construct the computation in Box B'. Memoization, however, will not provide for such result re-use because calls to `iSort` that construct box B' all depend on the new accumulator. Since this accumulator is not part of the computation before the input change, no memo matches will take place. This is consistent with the fact that memoization can only re-use accurate results. What we would like is a technique for re-using the inaccurate result in box B and transforming it to the accurate result B' efficiently by taking advantage of their similarity.

Adaptive memoization provides re-use of such inaccurate results. The code for the adaptively memoized version of insertion sort is shown on the right in Figure 6. The code is obtained by first making insertion sort adaptive by changing its input to modifiable lists and memoizing the calls to the `m_insert` and `m_iSort` functions.

The key difference from the conventional memoization, or the orthogonal combination, is that `m_iSort` and `m_insert` are not memoized based on the accumulator (`a`). This is indicated by assigning the accumulator the *question* type `?`, which marks types that are *unmatched* during memo lookup. This means calling `m_iSort` with the same list `l` will cause a memo match regardless of the value of the accumulator `a`. Referring back to our example in Figure 7, this kind of memoization will enable us to re-use box B. The function `m_insert` is somewhat subtle: it is memoized based on the key `k` being inserted and the key of the accumulator that has been previously visited `v`. Memoizing `m_insert` based on the keys of the accumulator will enable re-use of results from an evaluation with some other list with the same keys, even when the two lists consists of different cons cells. Referring back to our example in Figure 7, the calls to `m_insert` will synchronize the accumulator created by the insertion of 9 (pointed by the thick arrow on the right) with the first accumulator of box B and will insert 9 into the latter. Adaptive memoization will update all inaccurate result by running a change propagation. The technique thus enables re-use of box B, and updates it to obtain B'.

Insertion sort is an example program where a data structure (the accumulator) is threaded through the program. The problem is that making a small change to the input can make a small change to the accumulator preventing re-use of previously computed results however similar they might be. By not memoizing results based on the accumulator the adaptively memoized insertion sort takes advantage of the structural similarity of the computations before and after a change. This suffices to obtain the linear time bound for insertions/deletions proven in Section 6. Note that this bound is within an expected constant factor of the optimal for insertion sort (because otherwise insertion sort can be made to run faster than $O(n^2)$ by successive insertions).

In the rest of this section, we present a high level presentation of the techniques for supporting adaptive memoization. These ideas are are formalized in Section 4 by presenting a functional language for adaptive memoization and studying its static and dynamic semantics.

The idea of adaptive memoization is to remember *adaptive computations* along with their arguments so that they can not only be re-used but also adapted according to different argument values. As with orthogonal memoization, adaptive memoization remembers both the result and the dynamic dependence graph for a memoized call. In addition, adaptive memoization remembers the unmatched arguments, *i.e.*, arguments with questions types. Since memo lookups only compare matched arguments, a memo match can return an inaccurate result. All re-used results are therefore made accurate by forcing the unmatched arguments to match. This is done by changing the values of unmatched arguments to the values of the unmatched arguments of the current call and performing a change propagation. Since unmatched arguments can be changed, they must be modifiables. This is one of properties that the type system described in Section 4 for adaptive memoization ensures.

The key challenge to realize this idea is to represent and reuse adaptive computations while also supporting change propagation so that inaccurate results can be made accurate according to the unmatched arguments. The difficulty is that adjusting some part of the computation by change propagation can affect other parts of the computation. To overcome this challenge, adaptive memoization relies on a

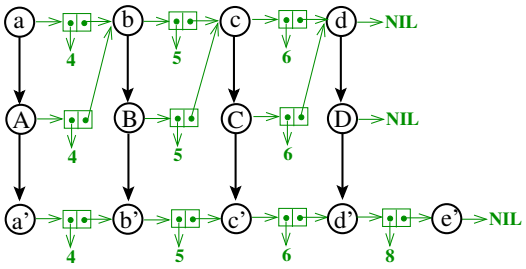

Figure 8: DDGs of `m_insert` (!8, ?[4, 5, 6], !0).

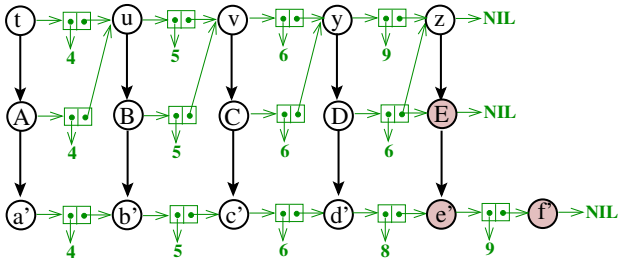

Figure 9: DDGs of `m_insert` (!8, ?[4, 5, 6, **9**], !0)).

copying technique. When a memoized function is being evaluated, the underlying systems makes copies of the unmatched arguments and performs evaluation after substituting these copies instead of the original arguments. In other words, the system creates a "stand-alone" adaptive computation encapsulated with its "inputs", that is, the unmatched arguments. When a result is re-used, its unmatched arguments are changed and the output is updated by change propagation.

Consider a concrete example from insertion sort to make these ideas clear. Figure 8 shows the dependence graph for the call `m_insert` (!8, ?[4, 5, 6], !0) that inserts 8 to [4, 5, 6]. The input consists of a modifiable list (the accumulator) consisting of the modifiables `a,b,c,d`. Since the accumulator is unmatched (a question type), the system makes a copy of the accumulator and evaluates the body of the function `m_insert` with this copy. In other words, the modifiables `a,b,c,d` are copied to the modifiables `A,B,C,D`. We will refer to these modifiables as the *local copies*. Copying the modifiables `a,b,c,d` to `A,B,C,D` requires reading each modifiable and then writing the value read to the corresponding local copy. Explicitly reading the unmatched arguments creates a data dependence between the copies and the unmatched arguments—this dependence relation ensures that unmatched arguments and local copies remain consistent under change propagation. Local copies enable isolating and adapting dynamic dependence graphs by change propagation.

As an example of how adaptive memoization enables re-use of an inaccurate result, consider performing the call `m_insert` (!8, ?[4, 5, 6, **9**], !0) after the call `m_insert` (!8, ?[4, 5, 6], !0). Assume also that the lists [4, 5, 6] and [4, 5, 6, **9**] consists of entirely different cons cells. Now with conventional memoization none of the result from the first call would be re-used in computing the second—since the accumulators are different none of the recursive calls will match in the memo. With adaptive memoization however the accumulators will not be matched (they have type question) and therefore each recursive call performed by `minsert`(!8, ?[4, 5, 6, **9**], !0) will match a previous call by `m_insert` (!8, ?[4, 5, 6], !0) except for the last call for key 9. When a match occurs the local copy of the memoized computation is updated by the value of the unmatched argument by reading the unmatched argument and writing its value to the local copy—this establishes the consistency of the local copy and the unmatched argument. The dynamic dependence graph obtained by `m_insert` (!8, ?[4, 5, 6, **9**], !0) is shown in Figure 9. Note how the modifiables `t,u,v,y,z` of the new accumulator [4, 5, 6, **9**], are synchronized with the local copies `A,B,C,D,E` from the previous evaluation (Figure 8). Note also that and a large piece of the dependence graph from Figure 8 is re-used. The changed pieces of the graph are shaded in color and consist of the nodes `e,e',f'` and the edges between them. Referring back to our example in Figure 7, this is how boxes B and B' are "synchronized" by matching the new accumulator [4, 5, 6, **9**] to [4, 5, 6] and updating it with the newly inserted key **9**.

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 [3] with support for adaptive memoization.

Our implementation of the IFL language, described in Section 5, 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 [3] 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. 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 10. 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, $\tau_1 \xrightarrow{s} \tau_2$, changeable function types $\tau_1 \xrightarrow{c} \tau_2$, memoized-stable function types $\tau_1 \xrightarrow{ms} \tau_2$, and

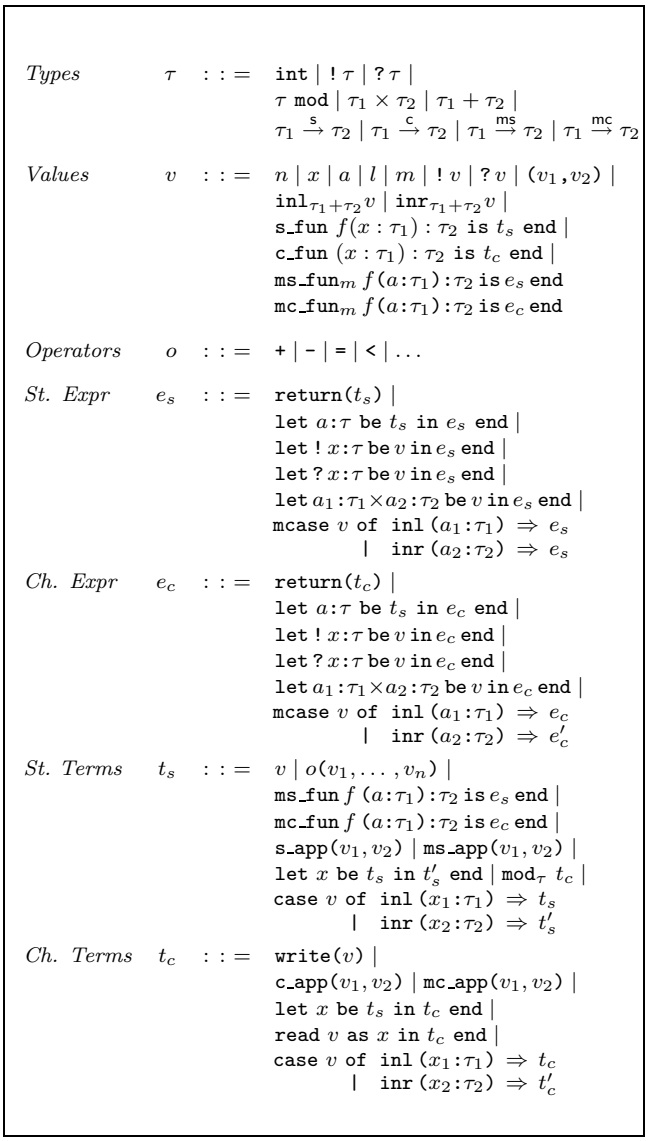

Figure 10: The abstract syntax of IFL.

memoized-changeable function types $\tau_1 \xrightarrow{mc} \tau_2$. 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 [3]. 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 [3]. 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 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. This section presents an overview of the static semantics, the full type system is provided in Appendix B.

Each typing judgment takes place under three contexes: Δ for resources, Λ for locations, and Γ for ordinary variables. We distinguish two modes, stable and changeable. Stable terms and expressions are typed in the stable mode and changeable terms are typed in the changeable mode.

The judgment $\Delta; \Lambda; \Gamma \Vdash t : \tau$ states that t is a well formed stable term of type τ relative to Δ, Λ and Γ. The judgment $\Delta; \Lambda; \Gamma \Vdash e : \tau$ states that e is a well formed stable expression of type τ relative to Δ, Λ and Γ.

The judgment $\Delta; \Lambda; \Gamma \vDash t : \tau$ states that t is a well formed changeable term of type τ relative to Δ, Λ and Γ. The judgment $\Delta; \Lambda; \Gamma \vDash e : \tau$ states that e is a well formed changeable expression of type τ relative to Δ, Λ and Γ.

Figure 11 shows some sample typing judgments for stable and changeable terms. Figure 12 shows some sample typing judgments for stable and changeable expressions.

To support adaptive memoization we use the *question types* `?(`τ `mod)`. The `?` construct introduces a question type and `let?` construct eliminates it. One non-orthogonal re-

quirement about question types is that their underlying type must be a modifiable type. This is an artifact of the interaction between memoization and adaptivity. The typing rules for the bang types and the `?` types, shown in Figure 12 are otherwise symmetric.

$$\frac{\Delta, a{:}\tau_1; \Lambda; \Gamma,\ f{:}\tau_1 \overset{\mathsf{ms}}{\to} \tau_2 \Vdash e_s : \tau_2}{\Delta; \Lambda; \Gamma \Vdash \texttt{ms\_fun}\, f\ (a{:}\tau_1){:}\tau_2\ \texttt{is}\, e_s\ \texttt{end} : \tau_1 \overset{\mathsf{ms}}{\to} \tau_2} \quad \textbf{(st. mfun)}$$

$$\frac{\Delta, a{:}\tau_1; \Lambda; \Gamma,\ f{:}\tau_1 \overset{\mathsf{mc}}{\to} \tau_2 \vDash e_c : \tau_2}{\Delta; \Lambda; \Gamma \Vdash \texttt{mc\_fun}\, f\ (a{:}\tau_1){:}\tau_2\ \texttt{is}\, e_c\ \texttt{end} : \tau_1 \overset{\mathsf{mc}}{\to} \tau_2} \quad \textbf{(ch. mfun)}$$

$$\frac{\Delta; \Lambda; \Gamma \Vdash v_1 : \tau_1 \overset{\mathsf{ms}}{\to} \tau_2 \quad \Delta; \Lambda; \Gamma \Vdash v_2 : \tau_1}{\Delta; \Lambda; \Gamma \Vdash \texttt{ms\_app}(v_1, v_2) : \tau_2} \quad \textbf{(memo apply)}$$

$$\frac{\Delta; \Lambda; \Gamma \Vdash v_1 : (\tau_1 \overset{\mathsf{mc}}{\to} \tau_2) \quad \Delta; \Lambda; \Gamma \Vdash v_2 : \tau_1}{\Delta; \Lambda; \Gamma \vDash \texttt{mc\_app}(v_1, v_2) : \tau_2} \quad \textbf{(memo apply)}$$

Figure 11: Some typing judgments for stable (top) and changeable (bottom) terms.

$$\frac{\Delta; \Lambda; \Gamma \Vdash v : !\,\tau_1 \quad \Delta; \Lambda; \Gamma, x{:}\tau_2 \Vdash e_s : \tau_2}{\Delta; \Lambda; \Gamma \Vdash \texttt{let !}\, x{:}\tau_1\, \texttt{be}\, v\, \texttt{in}\, e_s\, \texttt{end} : \tau_2} \quad \textbf{(let!)}$$

$$\frac{\Delta; \Lambda; \Gamma \Vdash v : \texttt{?}\,(\tau_1\ \texttt{mod}) \quad \Delta; \Lambda; \Gamma, x{:}\tau_1\ \texttt{mod} \Vdash e_s : \tau_2}{\Delta; \Lambda; \Gamma \Vdash \texttt{let ?}\, x{:}(\tau_1\ \texttt{mod})\, \texttt{be}\, v\, \texttt{in}\, e_s\, \texttt{end} : \tau_2} \quad \textbf{(let?)}$$

$$\frac{\Delta; \Lambda; \Gamma \Vdash v : !\,\tau \quad \Delta; \Lambda; \Gamma, x{:}\tau \vDash e_c : \tau}{\Delta; \Lambda; \Gamma \Vdash \texttt{let !}\, x{:}\tau\, \texttt{be}\, v\, \texttt{in}\, e_c\, \texttt{end} : \tau} \quad \textbf{(let!)}$$

$$\frac{\Delta; \Lambda; \Gamma \Vdash v : \texttt{?}\,(\tau_1\ \texttt{mod}) \quad \Delta; \Lambda; \Gamma, x{:}\tau_1\ \texttt{mod} \vDash e_c : \tau_2}{\Delta; \Lambda; \Gamma \vDash \texttt{let ?}\, x{:}(\tau_1\ \texttt{mod})\, \texttt{be}\, v\, \texttt{in}\, e_c\, \texttt{end} : \tau_2} \quad \textbf{(let?)}$$

Figure 12: Some typing judgements for stable (top) and changeable (bottom) expressions.

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 $\sigma, m{:}\beta, e_s \Downarrow\!\Downarrow^{\mathsf{s}} \sigma', v, \mathtt{T}_s$ 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 $\sigma, m{:}\beta, l \leftarrow e_c \Downarrow\!\Downarrow^{\mathsf{c}} \sigma', \mathtt{T}$ 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.

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

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 $R_l^{x.t}(\mathtt{T}_c)$ 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 5 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. Specifi-

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

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

Figure 13: Stable and changeable memoized applications.

cally, the trace $\mathtt{T}_c$ (and any read in $\mathtt{T}_c$) is contained within the read trace $R_l^{x.t}(\mathtt{T}_c)$. 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.

The complete dynamic semantics is provided Appendix C. In the rest of the section, we briefly walk through some the more interesting rules.

Term evaluation. Figure 13 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 made accurate by change propagation and returned—the body of `return` is skipped. Otherwise, the body of the return is executed.

Figure 14 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 $\texttt{split}\,(\cdot)$ that splits a branch into a list of the unmatched modifiables and a branch β'. In Figure 14 the top two rules are the memo lookups. Unmatched modifiables are denoted as l_i's. The $m : \beta', \mathtt{T} \rightsquigarrow$ relation performs the memo lookup with the filtered branch

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ ([l_1, \ldots, l_n], \beta') = \texttt{split}\,(\beta) \\ m : \beta', \mathtt{T} \rightsquigarrow \epsilon, \_ \\ \alpha' = \alpha[l'_1 \mapsto \alpha[l_1], \ldots, l'_n \mapsto \alpha[l_n]], \\ \text{where } l'_1 \notin \mathrm{dom}(\alpha), \ldots, l'_n \notin \mathrm{dom}(\alpha), l'_i \neq l'_j \\ \sigma' = (\alpha', \mu, \chi, T) \\ \sigma', [l'_1/l_1, \ldots l'_n/l_n]t_s\ \Downarrow^{\mathsf{s}} v, \sigma'', \mathtt{T}_s \\ \mathtt{T}'_s = \langle R^{x.\mathtt{write}(x)}_{l_1} \mathtt{W}_{\tau_1} \rangle_{l'_1:\tau_1} \cdots \langle R^{x.\mathtt{write}(x)}_{l_n} \mathtt{W}_{\tau_n} \rangle_{l'_n:\tau_n}\end{array}}{\sigma, m{:}\beta, \texttt{return}(t_s) \Downarrow^{\mathsf{s}} v, \sigma'', \left(\mathtt{T}'_s\,;\{\,\mathtt{T}_s\,\}^{m:\beta}_{(v,(l'_1,\ldots,l'_n))}\right)} \quad (\times)$$

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

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

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

Figure 14: Sample stable-expression evaluation.

β' and the identifier m in the re-use trace $\mathtt{T}$. If a match is not found it returns the trace of the memoized function and the tail of the re-use trace following the match, otherwise it returns an empty trace. 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), then the body of `return` is skipped. The re-used trace is updated with change propagation and returned along with the trace of the performed copies.

5 Implementation

This section describes the implementation of a library for supporting the IFL language described in Section 4. The implementation follows the semantics of the IFL language and builds upon our implementations of adaptivity [2] and selective memoization [3]. Carlsson [8] presented an implementation of the adaptivity libraries in the Haskell language.

As with our previous work, the implementation is efficient: its overhead is expected constant over a standard semantics. The expected constant overhead is due to the use of hash tables for implementing memo tables. Another key property of the implementation is that it does not rely on

complex or problem specific cache management strategies. The semantics makes clear what result should be cached and when results should be evicted. In particular, all invalidated computations are evicted from cache. Since all cached computations are valid, the size of the memo tables never exceeds the size of the dynamic dependence graphs assuming for program that has no non-adaptive memoized functions.

The main idea behind the implementation is to memoize adaptive computations instead of just results. To do this, we implement the memo table of each function as a hash table mapping branches to the local copies of unmatched arguments, the result, and the time-interval at which the call took place. The time-interval representation of adaptive computations rely on representing traces (Section 4) via time stamps maintained in order maintenance data structure [10]. This representation based on time-stamps is a critical component of the implementation and described in more detail in our previous work [2].

When a memo match occurs, adaptive computation represented by the time-interval is joined into the current computation, the values of unmatched arguments are copied, a change propagation is performed to make accurate the result, and finally the result is returned. Since adaptive memoization re-uses computations and not just values, a memo table entry can only be re-used once. The implementation ensures this by restricting all memo searches to within the time interval being re-evaluated during change propagation. In other words, only results from the part of the computation being invalidated are re-used. This is consistent with the dynamic semantics presented in Section 4.

Re-use of computations interacts with time-stamps and the order maintenance data structure in subtle ways. Since re-using a computation effectively splices in a whole set of time-stamps at current point in the computation, it can change the ordering of the time-stamps. To maintain the ordering the implementation invalidates all computations that take place between the current time and the start time of the re-used computation. This is easily done by deleting the time stamps between the current time and the start of the re-used computation. Since a computation is re-used only when it is to be invalidated, this causes no problems. This is consisten with the dynamic semantics (Section 4).

6 Applications

We describe how to make Insertion Sort and Quick Sort incremental under insertions and deletions to the input and prove strong performance bounds. For insertion sort, we show that an insertion or deletion is handled in expected-case $O(n)$ time with adaptive memoization. For Quicksort, we consider insertions and deletions at random locations and show an expected $O(\log^2 n)$ bound by using the orthogonal combination. We improve this to expected $O(\log n)$ 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. The $O(n)$ and $O(\log n)$ bounds are optimal for these algorithms.

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.
```

The proofs of the theorems in this section are provided in Appendix A.

6.1 Incremental Insertion Sort

Figure 15 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 => write (CONS (i,t))
      | CONS(hh,tt) =>
        if (i < hh) then
          write (CONS(i,t))
        else
          write (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 15: 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 $O(n)$ time.

Theorem 1
Insertion sort (shown in Figure 15) updates its result in expected $O(n)$ time when its input is changed by an insertion or deletion anywhere in the list.

Proof: The proof is given in Appendix A. ■

6.2 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.

```
fil:(!(int->bool)*!int modlist)->int modlist
ms_fun fil (!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, (!f,!t)))
        else
          read (ms_app(fil, (!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, (!(fn x => x > h),!t))
        val gs = mod (c_app (qs, (g,rest)))
        val s = ms_app(fil, (!(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 16: Quicksort with the orthogonal combination.

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

Quicksort with Orthogonal Combination. Figure 16 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].

Theorem 2
The Quicksort with the orthogonal combination takes expected $O(n \log n)$ time for insertions at the head of the input, expected $O(\log n)$ time for insertions at the end of the input, and expected $O(\log^2 n)$ time for insertions at a (uniformly) randomly chosen position. Expectations are over all permutations of the input list. The same bounds apply to deletions.

Proof: The proof is given in Appendix A. ■

Quicksort with Adaptive Memoization. Figure 17 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 $O(n)$ time for insertions at the head of the input, expected $O(\log n)$ time for insertions at the end of the input, and expected $O(\log n)$ time for insertions at a uniformly randomly chosen position. The expectations are over permutations of the input list. The same bounds apply to deletions.

```
fil:(!(int->bool)*(!int*?int modlist))->int modlist
ms_fun fil (!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, (!f,(!hh,?tt))))
        else
          read (ms_app(fil, (!f,(!hh,?tt)))) as vtt in
            write vtt
          end
    end)

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

Figure 17: Quicksort with adaptive memoization.

Proof: The proof is given in Appendix A. ■

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. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.

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

[5] Umut A. Acar, Guy E. Blelloch, Srinath Sridhar, and Virgina Vassilevska. Dynamic pointer machine and applications, 2004. In preperation.

[6] Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. Convex hulls for dynamic data, 2004. In preperation.

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

[8] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 26–35. ACM Press, 2002.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[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] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

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

A Proofs for Insertion Sort and Quicksort

For our results, we assume that inputs to the applications do not contain multiple instances of the same key. Uniqueness can easily be ensured by associating a unique identifier with each element of the input.

The time for updating the output of an incremental computation is affected by the kind and the size of the priority queue employed for change propagation [2, 4]. Although a general purpose, logarithmic time priority queue works for all applications, it is not efficient for many applications. For example, we showed that a certain class of parallel computation can use a constant-time priority queue [4]. In insertion sort and the Quicksort with orthogonal combination, the size of the queue is bounded by a constant, and thus a general purpose priority queue can be used. For the adaptively memoized version of Quicksort however the queue size can be linear in the size of the input. Thus a general purpose priority queue does not work well. Instead, we use a constant-time doubly ended priority queue. Insertions to the queue are done either at the front or the back and deletions are always done at the front. An insertion checks if the priority of the inserted key is higher than the key at the front, if so the key inserted at the front, otherwise it is inserted at the back.

Theorem 4 (Insertion Sort)
Insertion sort (shown in Figure 15) updates its result in expected $O(n)$ time when its input is changed by an insertion or deletion anywhere in the list. The expectation is over internal randomization of the choice of hash function used for memo tables.

Proof: We consider `iSort` and `insert` in isolation. Inserting a new key k will change some modifiable in the list and insert a new modifiable m for the tail. Since the tail of m will not be affected, `iSort` will synchronize with the previous execution after two calls to `insert`–even though the accumulator has changed.

To insert the new key k to the result, function `iSort` will call `insert`. Since k has never been seen before, `insert` will insert k to the accumulator returning a new accumulator. Since the accumulator has now changed, the subsequent calls to `insert` will need to be re-executed. Since the contents of the accumulator are the same as before, except for the location where k is inserted, each re-executed read of `insert` will synchronize with the previous execution by returning the same result. At most n reads will involve the new key k, and therefore the accumulators will be synchronized after $O(n)$ re-execution of the sole read in `insert`. Since each re-execution take expected constant time, the result will be updated in expected $O(n)$ time ■

Theorem 5 (Quicksort with Orhogonal Combination)
The Quicksort using the orthogonal combination of adaptivity and memoization (Figure 16) updates its output in $O(n \log n)$ time for insertions at the head of the input, $O(\log n)$ time for insertions at the end of the input, and expected $O(\log^2 n)$ time for insertions at a (uniformly) randomly chosen position. All bounds are expected with expectations over permutations of the input list. Same bound applies to deletions.

Proof: Inserting a key at the head of the input list re-executes the first call to `qs` and thus takes expected $O(n \log n)$ time. In previous work, we showed that insertions at the end of the input take $O(\log n)$ expected time using adaptivity only, thus the same result applies.

Consider inserting a new key k anywhere in the list. Key k will be propagated down the recursion tree by re-executing calls to `fil` along some path until k becomes the pivot. When k becomes the pivot, the corresponding call to `qs` will be re-executed. Since `fil` is memoized based on the input list, a single insertion to the input will take expected constant time to handle at each level. Consequently, the total time is the no more than the depth of the tree plus $O(m \log m)$ where m is the number of keys in the input when k becomes pivot. The depth of the tree is expected $O(\log n)$. Thus we need to show is that $O(m \log m)$ is $O(\log^2 n)$ in the expected case.

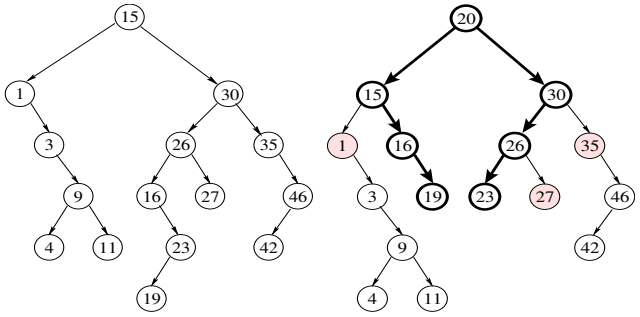

Figure 18: The recursion tree for Quicksort with inputs $L = [15, 30, 26, 1, 3, 16, 27, 9, 35, 4, 46, 23, 11, 42, 19]$ (left) and $L' = [20, L]$ (right).

The expected time for `qs` on an input list of size m is, $E[T(m)] = O(m \log m)$ with expectation over all permutations of the input. We are interested in the expectation of this for a random insertion position. Thus $E[E[T(m)]] = E[O(m \log m)]$, since $m \leq n$, we have $E[E[T(m)]] = (\log n)E[m]$. The expected value m is $O(\log n)$ by using the well known isometry between the pivot-tree of Quicksort and Treaps [19]. Thus a uniformly random insertion takes $O(\log^2 n)$ time. ■

Theorem 6 (Quicksort with Adaptive Memoization)
The adaptively memoized Quicksort (Figure 16) updates its output in expected $O(n)$ time for insertions at the head of the input, expected $O(\log n)$ time for insertions at the end of the input, and expected $O(\log n)$ time for insertions at a uniformly randomly chosen position. The expectations are over permutations of the input list. Same bound applies to deletions.

Proof: The result for inserting at the end of the input list relies only on adaptivity and our previous result applies [2]. To prove the $O(n)$ bound for insertions at the head, we use an argument that is similar to that with memoization Quicksort [3]. The result for a random insertion will then follow by the fact that the expected size of a randomly selected subtree has size $O(\log n)$.

Consider the recursion tree for Quicksort on some input where each recursive call is marked with its pivot. Now consider the recursion tree of Quicksort on the input changed by inserting a new key k at the head. Figure 18 shows the pivots trees for two such inputs. Since the new key k is inserted at the head of the input list, it will become root. Consider the rightmost spine of the left subtree of the root and the leftmost spine of the right subtree of the root—these spines are marked with bold edges in Figure 18. The following properties are true as shown in previous work [3].

1. The subtree connected to the vertices of the spine are identical in both recursion trees.
2. The sum of the sizes of the subtrees of vertices on the spines is expected $O(n)$, where the expectation is over all permutation of the input.

Consider any call that is not on the spine. By property (1) the input to that call is identical before and after the insertion. The call to filters at each such node will therefore find its result in the memo and take expected constant time. Since calls to filter are performed in reverse sorted order in both computations, re-use of a result will not invalidate some other result. Thus all the calls except for those at the two spines will take constant time. Note that such result re-use is not possible without adaptive memoization, because the input lists will not in fact be identical even though their contents are the same—the filters at the root

will generate all new results. The calls at each spine will take expected linear time in the size of their inputs. By property (2), the sum of the input sizes to the calls on the spines is expected $O(n)$, therefore the time for these calls is $O(n)$. This establishes the $O(n)$ bound for an insertion at the head of the input list.

For random insertions, consider inserting a new key k at a random position in the input. The newly inserted key will be propagated down the call tree by re-executing filter calls along some path. Since the `fil` is memoized each level will take expected constant time. Since the depth of the recursion tree is expected $O(\log n)$, the time for these call will be expected $O(\log n)$. When the new key becomes the pivot, it will cause the corresponding call to `qs` to re-execute. By using the result for an insertion at the of the input, this will take $O(m)$ time where m is the size of the input to that call. By using the known isometry between Treaps [19] and the call tree of Quicksort, the expected size of m if $O(\log n)$. It follows that the expected time to handle a random insertion is $O(\log n)$. ■

B Static Semantics

The section presents the complete static semantics for the IFL language.

Each typing judgment takes place under three contexes: Δ for resources, Λ for locations, and Γ for ordinary variables. We distinguish two modes, stable and changeable. Stable terms and expressions are typed in the stable mode and changeable terms are typed in the changeable mode.

The judgment $\Delta;\Lambda;\Gamma \Vdash t : \tau$ states that t is a well formed stable term of type τ relative to Δ, Λ and Γ. The judgment $\Delta;\Lambda;\Gamma \Vdash e : \tau$ states that e is a well formed stable expression of type τ relative to Δ, Λ and Γ.

The judgment $\Delta;\Lambda;\Gamma \vDash t : \tau$ states that t is a well formed changeable term of type τ relative to Δ, Λ and Γ. The judgment $\Delta;\Lambda;\Gamma \vDash e : \tau$ states that e is a well formed changeable expression of type τ relative to Δ, Λ and Γ.

To support adaptive memoization we use the *question types* $\texttt{?}(\tau\ \texttt{mod})$. The `?` construct introduces a question type and `let?` construct eliminates it. One non-orthogonal requirement about question types is that their underlying type must be a modifiable type. This is an artifact of the interaction between memoization and adaptivity. The typing rules for the bang types and the `?` types are otherwise symmetric.

The typing rules distinguish between terms and expressions and a stable and changeable context. The stable and changeable expression are almost identical except for the `return` construct. Figure 19 shows the typing rules for values, Figure 20 shows the typing rules for terms, and Figure 21 shows the typing ruels for expressions. The abstract syntax is shown in Figure 10.

$$\frac{}{\Delta;\Lambda;\Gamma \Vdash n : \texttt{int}}$$

$$\frac{(\Delta(a) = \tau)}{\Delta;\Lambda;\Gamma \Vdash a : \tau} \quad \frac{(\Lambda(l) = \tau)}{\Delta;\Lambda;\Gamma \Vdash l : \tau\ \texttt{mod}} \quad \frac{(\Gamma(x) = \tau)}{\Delta;\Lambda;\Gamma \Vdash x : \tau}$$

$$\frac{\emptyset;\Lambda;\Gamma \Vdash t : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{!}\, t : \texttt{!}\, \tau} \quad \frac{\emptyset;\Lambda;\Gamma \Vdash t : \tau\ \texttt{mod}}{\Delta;\Lambda;\Gamma \Vdash \texttt{?}\, t : \texttt{?}\, (\tau\ \texttt{mod})}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_1 : \tau_1 \quad \Delta;\Lambda;\Gamma \Vdash v_2 : \tau_2}{\Delta;\Lambda;\Gamma \Vdash (v_1, v_2) : \tau_1 \times \tau_2}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1}{\Delta;\Lambda;\Gamma \Vdash \texttt{inl}_{\tau_1+\tau_2} v : \tau_1 + \tau_2} \quad \frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{inr}_{\tau_1+\tau_2} v : \tau_1 + \tau_2}$$

$$\frac{\Delta;\Lambda;\Gamma,\ f : \tau_1 \xrightarrow{\mathsf{s}} \tau_2,\ x : \tau_1 \Vdash t_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{s\_fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ t_s\ \texttt{end} : (\tau_1 \xrightarrow{\mathsf{s}} \tau_2)}$$

$$\frac{\Delta;\Lambda;\Gamma,\ f : \tau_1 \xrightarrow{\mathsf{c}} \tau_2,\ x : \tau_1 \vDash t_c : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{c\_fun}\ (x : \tau_1) : \tau_2\ \texttt{is}\ t_c\ \texttt{end} : (\tau_1 \xrightarrow{\mathsf{c}} \tau_2)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma, f{:}\tau_1 \xrightarrow{\mathsf{ms}} \tau_2;\Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{ms\_fun}_m\ f(a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_s\ \texttt{end} : \tau_1 \xrightarrow{\mathsf{ms}} \tau_2}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma, f{:}\tau_1 \xrightarrow{\mathsf{mc}} \tau_2;\Vdash e_c : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{mc\_fun}_m\ f(a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_c\ \texttt{end} : \tau_1 \xrightarrow{\mathsf{mc}} \tau_2}$$

Figure 19: Typing of values.

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_i : \tau_i \quad (1 \le i \le n) \quad \vdash_o o : (\tau_1, \ldots, \tau_n)\ \tau}{\Delta;\Lambda;\Gamma \Vdash o(v_1, \ldots, v_n) : \tau} \textbf{ (prims)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma,\ f{:}\tau_1 \xrightarrow{\mathsf{ms}} \tau_2 \Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{ms\_fun}\ f\ (a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_s\ \texttt{end} : \tau_1 \xrightarrow{\mathsf{ms}} \tau_2} \textbf{ (st. mfun)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma,\ f{:}\tau_1 \xrightarrow{\mathsf{mc}} \tau_2 \vDash e_c : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{mc\_fun}\ f\ (a{:}\tau_1){:}\tau_2\ \texttt{is}\ e_c\ \texttt{end} : \tau_1 \xrightarrow{\mathsf{mc}} \tau_2} \textbf{ (ch. mfun)}$$

$$\frac{\Lambda;\Gamma \Vdash v_1 : (\tau_1 \xrightarrow{\mathsf{s}} \tau_2) \quad \Lambda;\Gamma \Vdash v_2 : \tau_1}{\Lambda;\Gamma \Vdash \texttt{s\_app}(v_1, v_2) : \tau_2} \textbf{ (stable apply)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_1 : \tau_1 \xrightarrow{\mathsf{ms}} \tau_2 \quad \Delta;\Lambda;\Gamma \Vdash v_2 : \tau_1}{\Delta;\Lambda;\Gamma \Vdash \texttt{ms\_app}(v_1, v_2) : \tau_2} \textbf{ (memo apply)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Lambda;\Gamma, x : \tau_1 \Vdash t'_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\ x\ \texttt{be}\ t_s\ \texttt{in}\ t'_s\ \texttt{end} : \tau_2} \textbf{ (let)}$$

$$\frac{\Delta;\Lambda;\Gamma \vDash t_c : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{mod}_\tau\ t_c : \tau\ \texttt{mod}} \textbf{ (mod)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta;\Lambda;\Gamma, x_1{:}\tau_1 & \Vdash & t_s : \tau \\ \Delta;\Lambda;\Gamma, x_2{:}\tau_2 & \Vdash & t'_s : \tau\end{array}}{\begin{array}{l}\Delta;\Lambda;\Gamma \Vdash\ \texttt{case}\ v\ \texttt{of}\ \texttt{inl}\,(x_1{:}\tau_1) \Rightarrow t_s\ : \tau \\ \qquad\qquad\quad \mid\ \texttt{inr}\,(x_2{:}\tau_2) \Rightarrow t'_s\end{array}} \textbf{ (case)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau}{\Delta;\Lambda;\Gamma \vDash \texttt{write}(v) : \tau} \textbf{ (write)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_1 : (\tau_1 \xrightarrow{\mathsf{c}} \tau_2) \quad \Delta;\Lambda;\Gamma \Vdash v_2 : \tau_1}{\Delta;\Lambda;\Gamma \vDash \texttt{c\_app}(v_1, v_2) : \tau_2} \textbf{ (apply)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_1 : (\tau_1 \xrightarrow{\mathsf{mc}} \tau_2) \quad \Delta;\Lambda;\Gamma \Vdash v_2 : \tau_1}{\Delta;\Lambda;\Gamma \vDash \texttt{mc\_app}(v_1, v_2) : \tau_2} \textbf{ (memo apply)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Delta;\Lambda;\Gamma, x : \tau_1 \vDash t_c : \tau_2}{\Delta;\Lambda;\Gamma \vDash \texttt{let}\ x\ \texttt{be}\ t_s\ \texttt{in}\ t_c\ \texttt{end} : \tau_2} \textbf{ (let)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1\ \texttt{mod} \quad \Delta;\Lambda;\Gamma, x : \tau_1 \vDash t_c : \tau_2}{\Delta;\Lambda;\Gamma \vDash \texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ t_c\ \texttt{end} : \tau_2} \textbf{ (read)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta;\Lambda;\Gamma, x_1{:}\tau_1 & \vDash & t_c : \tau \\ \Delta;\Lambda;\Gamma, x_2{:}\tau_2 & \vDash & t'_c : \tau\end{array}}{\begin{array}{l}\Delta;\Lambda;\Gamma \Vdash\ \texttt{case}\ v\ \texttt{of}\ \texttt{inl}\,(x_1{:}\tau_1) \Rightarrow t_c\ : \tau \\ \qquad\qquad\quad \mid\ \texttt{inr}\,(x_2{:}\tau_2) \Rightarrow t'_c\end{array}} \textbf{ (case)}$$

Figure 20: Typing of stable (top) and changeable (bottom) terms.

$$\frac{\emptyset;\Lambda;\Gamma \quad \Vdash \quad t_s : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{return}(t_s) : \tau} \textbf{ (return)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Delta, a{:}\tau_1;\Lambda;\Gamma \Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\ a{:}\tau_1\ \texttt{be}\ t_s\ \texttt{in}\ e_s\ \texttt{end} : \tau_2} \textbf{ (let)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : !\,\tau_1 \quad \Delta;\Lambda;\Gamma, x{:}\tau_2 \Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\,!\ x{:}\tau_1\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} : \tau_2} \textbf{ (let!)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : ?\,(\tau_1\ \texttt{mod}) \quad \Delta;\Lambda;\Gamma, x{:}\tau_1\ \texttt{mod} \Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\,?\,x{:}(\tau_1\ \texttt{mod})\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} : \tau_2} \textbf{ (let?)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1 \times \tau_2 \quad \Delta, a_1{:}\tau_1, a_2{:}\tau_2;\Lambda;\Gamma \Vdash e_s : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\ a_1{:}\tau_1 \times a_2{:}\tau_2\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} : \tau} \textbf{ (let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta, a_1{:}\tau_1;\Lambda;\Gamma & \Vdash & e_s : \tau \\ \Delta, a_2{:}\tau_2;\Lambda;\Gamma & \Vdash & e'_s : \tau\end{array}}{\begin{array}{l}\Delta;\Lambda;\Gamma \Vdash\ \texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\,(a_1{:}\tau_1) \Rightarrow e_s\ : \tau \\ \qquad\qquad\quad \mid\ \texttt{inr}\,(a_2{:}\tau_2) \Rightarrow e'_s\end{array}} \textbf{ (case)}$$

$$\frac{\emptyset;\Lambda;\Gamma \quad \vDash \quad t_c : \tau}{\Delta;\Lambda;\Gamma \vDash \texttt{return}(t_c) : \tau} \textbf{ (return)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Delta, a{:}\tau_1;\Lambda;\Gamma \vDash e_c : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\ a{:}\tau_1\ \texttt{be}\ t_s\ \texttt{in}\ e_c\ \texttt{end} : \tau_2} \textbf{ (let)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : !\,\tau \quad \Delta;\Lambda;\Gamma, x{:}\tau \vDash e_c : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\,!\ x{:}\tau\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} : \tau} \textbf{ (let!)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : ?\,(\tau_1\ \texttt{mod}) \quad \Delta;\Lambda;\Gamma, x{:}\tau_1\ \texttt{mod} \vDash e_c : \tau_2}{\Delta;\Lambda;\Gamma \vDash \texttt{let}\,?\,x{:}(\tau_1\ \texttt{mod})\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} : \tau_2} \textbf{ (let?)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1 \times \tau_2 \quad \Delta, a_1{:}\tau_1, a_2{:}\tau_2;\Lambda;\Gamma \vDash e_c : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\ a_1{:}\tau_1 \times a_2{:}\tau_2\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} : \tau} \textbf{ (let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta, a_1{:}\tau_1;\Lambda;\Gamma & \vDash & e_c : \tau \\ \Delta, a_2{:}\tau_2;\Lambda;\Gamma & \vDash & e'_c : \tau\end{array}}{\begin{array}{l}\Delta;\Lambda;\Gamma \Vdash\ \texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\,(a_1{:}\tau_1) \Rightarrow e_c\ : \tau \\ \qquad\qquad\quad \mid\ \texttt{inr}\,(a_2{:}\tau_2) \Rightarrow e'_c\end{array}} \textbf{ (case)}$$

Figure 21: Typing of stable (top) and changeable (bottom) expressions.

C 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. Re-use trace is provided by change propagation and is empty in the initial evaluation.

The term evaluation judgments consists of changeable and stable evaluation forms. The judgment $\sigma, t_s \Downarrow^{\mathtt{s}} v, \sigma', \mathtt{T}_s$ 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 $\sigma, l \leftarrow t_c \Downarrow^{\mathtt{c}} \sigma', \mathtt{T}_c$ 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 $\sigma, m\mathtt{:}\beta, e_s \Downarrow\!\!\Downarrow^{\mathtt{s}} \sigma', v, \mathtt{T}_s$ 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 $\sigma, m\mathtt{:}\beta, l \leftarrow e_c \Downarrow\!\!\Downarrow^{\mathtt{c}} \sigma', \mathtt{T}_c$ 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 $\mathtt{T}_c$.

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.

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

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 $R_l^{x.t}(\mathtt{T}_c)$ 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.

Dynamic dependency graphs and the memo tables described in Section 5 may be seen as an efficient representation of traces. Time intervals may be assigned to each read in the trace in left-to-right order. 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 $R_l^{x.t}(\mathtt{T}_c)$. Memo tables remember 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, 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 $l_1, \ldots, l_n$. Since changeable expression write their result to a modifiable, an explicit result is not stored for memoized changeable expressions.

Term evaluation. Figures 22 and 23 show the evaluation rules for stable and changeable terms. Memoized stable and memoized changeable functions are evaluated into values by generating a new memoized function identifier m. 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 memo function identifier m, a branch, and a re-use trace. The incremental evaluation constructs (`let!`, `let?`, `let*`, `mcase`) create a branch, denoted β. A *branch* is a list of *events* corresponding to "choice points" in the evaluation of an expression.

$$
\begin{array}{lcll}
\textit{Event} & \varepsilon & ::= & \mathtt{!}v \mid \mathtt{?}\,v \mid \mathtt{inl} \mid \mathtt{inr} \\
\textit{Branch} & \beta & ::= & \bullet \mid \varepsilon \cdot \beta
\end{array}
$$

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.

$$\sigma, v \;\Downarrow^{\mathsf{s}}\; v, \sigma, \varepsilon \quad \textbf{(value)} \qquad \sigma, o(v_1, \ldots, v_n) \;\Downarrow^{\mathsf{s}}\; \mathtt{app}(o, (v_1, \ldots, v_n)), \sigma, \varepsilon \quad \textbf{(primitive)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ \sigma' = (\alpha, \mu \cup \{m\}, \chi, \mathtt{T}),\ m \notin \mathrm{dom}(\mu)\end{array}}{\sigma, \mathtt{ms\_fun}\, f\ (a{:}\tau_1){:}\tau_2\ \mathtt{is}\, e_s\ \mathtt{end} \;\Downarrow^{\mathsf{s}}\; \mathtt{ms\_fun}_m\, f(a{:}\tau_1){:}\tau_2\ \mathtt{is}\, e_s\ \mathtt{end}, \sigma', \varepsilon} \quad \textbf{(memo stable fun)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ \sigma' = (\alpha, \mu \cup \{m\}, \chi, \mathtt{T}),\ m \notin \mathrm{dom}(\mu)\end{array}}{\sigma, \mathtt{mc\_fun}\, f\ (a{:}\tau_1){:}\tau_2\ \mathtt{is}\, e_c\ \mathtt{end} \;\Downarrow^{\mathsf{s}}\; \mathtt{mc\_fun}_m\, f(a{:}\tau_1){:}\tau_2\ \mathtt{is}\, e_c\ \mathtt{end}, \sigma, \varepsilon} \quad \textbf{(memo changeable fun)}$$

$$\frac{\begin{array}{c}(v_1 = \mathtt{s\_fun}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ t_s\ \mathtt{end}) \\ \sigma, [v_1/f, v_2/x]\ t_s \;\Downarrow^{\mathsf{s}}\; v, \sigma', \mathtt{T}_s\end{array}}{\sigma, \mathtt{s\_app}(v_1, v_2) \;\Downarrow^{\mathsf{s}}\; v, \sigma', \mathtt{T}_s} \quad \textbf{(stable apply)}$$

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

$$\frac{\begin{array}{lll}\sigma, t_s & \Downarrow^{\mathsf{s}} & v_1, \sigma', \mathtt{T}_s \\ \sigma', [v_1/x]\, t'_s & \Downarrow^{\mathsf{s}} & v_2, \sigma'', \mathtt{T}'_s\end{array}}{\sigma, \mathtt{let}\ x\ \mathtt{be}\ t_s\ \mathtt{in}\ t'_s\ \mathtt{end} \;\Downarrow^{\mathsf{s}}\; v_2, \sigma'', (\mathtt{T}_s\ ;\ \mathtt{T}'_s)} \quad \textbf{(let)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ \alpha' = \alpha[l \mapsto \square],\ l \notin \mathrm{dom}(\alpha) \\ (\alpha', \mu, \chi, \mathtt{T}), l \leftarrow t_c \;\Downarrow^{\mathsf{c}}\; \sigma', \mathtt{T}_c\end{array}}{\sigma, \mathtt{mod}_\tau\ t_c \;\Downarrow^{\mathsf{s}}\; l, \sigma', \langle \mathtt{T}_c \rangle_{l:\tau}} \quad \textbf{(mod)}$$

$$\frac{\sigma, t_s \;\Downarrow^{\mathsf{c}}\; v', \sigma', \mathtt{T}_s}{\sigma, \mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (x_1{:}\tau_1)\ \Rightarrow\ t_s\ |\ \mathtt{inr}\ (x_2{:}\tau_2)\ \Rightarrow\ t'_s\ \mathtt{end} \;\Downarrow^{\mathsf{c}}\; v', \sigma', \mathtt{T}_s} \quad \textbf{(case)}$$

$$\frac{\sigma, t'_s \;\Downarrow^{\mathsf{c}}\; v', \sigma', \mathtt{T}_s}{\sigma, \mathtt{case}\ \mathtt{inr}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (x_1{:}\tau_1)\ \Rightarrow\ t_s\ |\ \mathtt{inr}\ (x_2{:}\tau_2)\ \Rightarrow\ t'_s\ \mathtt{end} \;\Downarrow^{\mathsf{c}}\; v', \sigma', \mathtt{T}_s} \quad \textbf{(case)}$$

Figure 22: Evaluation of stable terms.

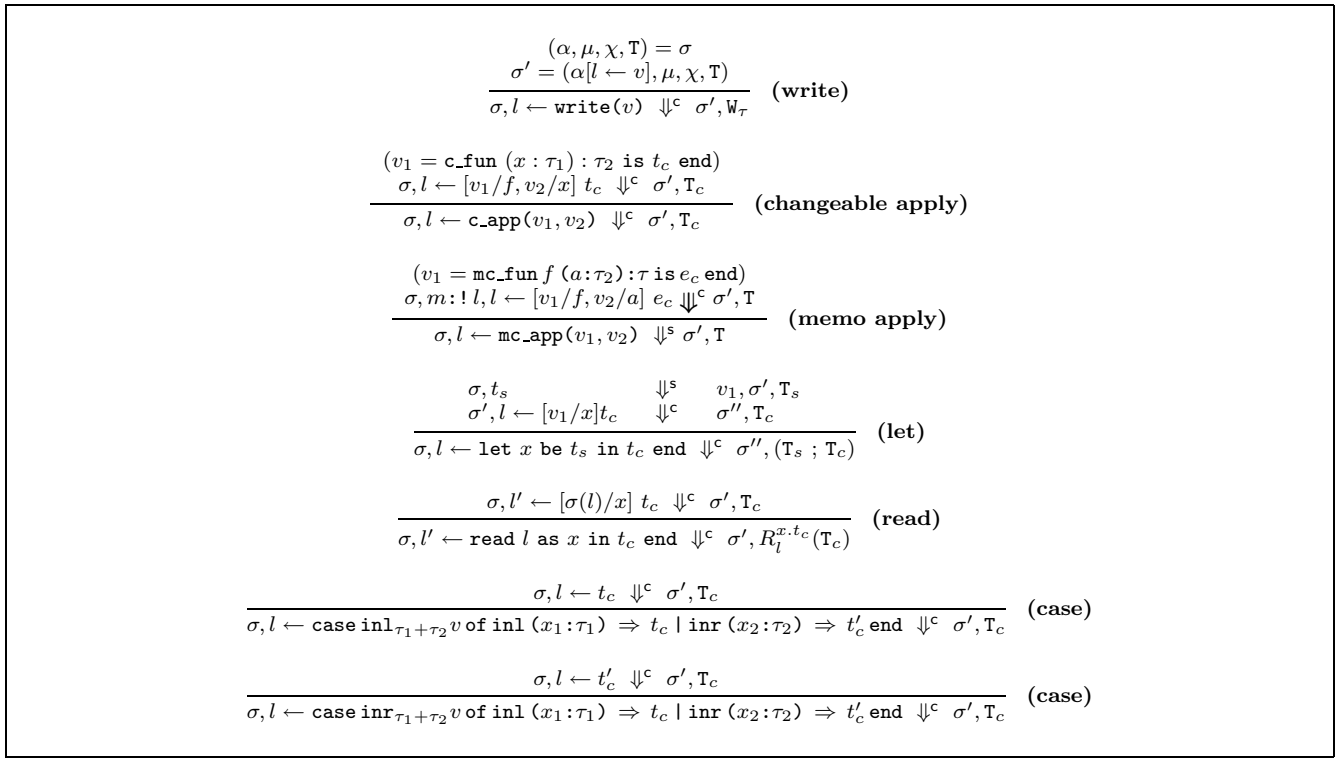

Figure 23: Evaluation of changeable terms.

Figure 24 shows the rules for stable-expression evaluation and Figure 25 shows the rules for changeable-expression evaluation. Changeable expressions are evaluated with an implicit target and the evaluation rules are otherwise similar to those of stable 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 $\mathtt{T}_s$. The evaluation $\sigma, m\!:\!\beta, l \leftarrow e_c \Downarrow^{\mathsf{c}} \sigma', \mathtt{T}_c$ states that the evaluation of changeable expression e_c in the context of the state σ, with memo function identifier m and branch β write to location l and yields the state σ' and the trace $\mathtt{T}_c$.

Adaptive memoization permits result re-use by matching a subset of the values that the result of a function depends on. The unmatched dependences are expressed by the `let?` construct. The type system ensures that all unmatched arguments are modifiables. During a memo lookup, unmatched modifiables are separated from other dependences and memo look up is performed based on the matched dependences only.

The first row of Figure 24 shows the evaluation rules for the stable `return` expression. First the unmatched modifiables $l_1 \ldots l_n$ are separated from the branch by $\mathtt{split}\,(\cdot)$ and a memo look up is performed. The memo lookup seeks for a memoized result in the re-use trace whose identifier and branch matches m and β'. If a result is not found, then the look up returns an empty trace (ε). If a result is found, then it returns the trace found and the uninspected tail of the re-use trace. Figure 26 shows the definition of a look up.

When no result is found in the memo (top, left rule in Figure 24), the unmatched modifiables $l_1, \ldots, l_n$ are copied into fresh modifiables $l'_1 \ldots l'_n$ and the body of the return is evaluated. The trace of the evaluation is then extended with the trace representing the copy operations and the result is returned.

When a match is found in the memo (top, right rule in Figure 24), the values of unmatched locations $l_1 \ldots l_n$ are copied to the local copies $l'_1 \ldots l'_n$ of the re-used computation and a change propagation is performed to update the re-used trace with respect to the values of unmatched modifiables. The trace returned by change propagation forms the result trace together with the trace of the copies. Since a result is found in the memo, the body of the `return` is skipped.

$$\frac{\begin{array}{c}(\alpha,\mu,\chi,\mathtt{T})=\sigma\\ ([l_1,\ldots,l_n],\beta')=\mathtt{split}\,(\beta)\\ m:\beta',\mathtt{T}\rightsquigarrow\epsilon,\_\\ \alpha'=\alpha[l'_1\leftarrow\alpha[l_1],\ldots,l'_n\leftarrow\alpha[l_n]],\ l'_i\notin\mathrm{dom}(\alpha),\ l'_i\neq l'_j\\ (\alpha',\mu,\chi,\mathtt{T}),[l'_1/l_1,\ldots l'_n/l_n]\ t_s\ \Downarrow^{\mathsf{s}}v,\sigma',\mathtt{T}_s\\ \mathtt{T}'_s=\langle R^{x.\mathtt{write}(x)}_{l_1}\mathtt{W}_{\tau_1}\rangle_{l'_1:\tau_1};\ldots;\langle R^{x.\mathtt{write}(x)}_{l_n}\mathtt{W}_{\tau_n}\rangle_{l'_n:\tau_n}\end{array}}{\sigma,m:\beta,\mathtt{return}(t_s)\Downarrow^{\mathsf{s}}v,\sigma',\left(\mathtt{T}'_s;\{\,\mathtt{T}_s\,\}^{m:\beta}_{(v,(l'_1,\ldots,l'_n))}\right)}$$

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

$$\frac{\sigma,t_s\ \Downarrow^{\mathsf{s}}\ \sigma',v,\mathtt{T}_s\qquad\sigma',m:\beta,[v/a]e_s\ \Downarrow^{\mathsf{s}}\ \sigma'',v',\mathtt{T}'_s}{\sigma,m:\beta,\mathtt{let}\ a:\tau\ \mathtt{be}\ t_s\ \mathtt{in}\ e_s\ \mathtt{end}\Downarrow^{\mathsf{s}}v',\sigma'',(\mathtt{T}_s;\mathtt{T}'_s)}\ \textbf{(let)}$$

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

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

$$\frac{\sigma,m:a_1\times a_2\cdot\beta,[v_1/a_1,v_2/a_2]e_s\ \Downarrow^{\mathsf{s}}\ v,\sigma',\mathtt{T}_s}{\sigma,m:\beta,\mathtt{let}\,a_1\times a_2\,\mathtt{be}\,v_1\times v_2\,\mathtt{in}\,e_s\,\mathtt{end}\Downarrow^{\mathsf{s}}v,\sigma',\mathtt{T}_s}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma,m:\mathtt{inl}\cdot\beta,[v/a_1]e_s\ \Downarrow^{\mathsf{s}}\ v',\sigma',\mathtt{T}_s}{\sigma,m:\beta,\ \begin{array}{l}\mathtt{mcase}\ \mathtt{inl}_{\tau_1+\tau_2}v\ \mathtt{of}\\ \quad\mathtt{inl}\,(a_1:\tau_1)\Rightarrow e_s\\ \quad\mid\mathtt{inr}\,(a_2:\tau_2)\Rightarrow e'_s\end{array}\ \Downarrow^{\mathsf{s}}v',\sigma',\mathtt{T}_s}\ \textbf{(case)}$$

$$\frac{\sigma,m:\mathtt{inr}\cdot\beta,[v/a_2]e_s\ \Downarrow^{\mathsf{s}}\ v',\sigma',\mathtt{T}_s}{\sigma,m:\beta,\ \begin{array}{l}\mathtt{mcase}\ \mathtt{inr}_{\tau_1+\tau_2}v\ \mathtt{of}\\ \quad\mathtt{inl}\,(a_1:\tau_1)\Rightarrow e_s\\ \quad\mid\mathtt{inr}\,(a_2:\tau_2)\Rightarrow e'_s\end{array}\ \Downarrow^{\mathsf{s}}v',\sigma',\mathtt{T}_s}\ \textbf{(case)}$$

Figure 24: Evaluation of stable expressions.

$$\frac{\begin{array}{c}(\alpha,\mu,\chi,\mathtt{T})=\sigma\\ ([l_1,\ldots,l_n],\beta')=\mathtt{split}\,(\beta)\\ m:\beta',\mathtt{T}\rightsquigarrow\epsilon,\_\\ \alpha'=\alpha[l'_1\leftarrow\alpha[l_1],\ldots,l'_n\leftarrow\alpha[l_n]],\ l'_i\notin\mathrm{dom}(\alpha),\ l'_i\neq l'_j\\ (\alpha',\mu,\chi,\mathtt{T}),l\leftarrow[l'_1/l_1,\ldots l'_n/l_n]\ t_c\ \Downarrow^{\mathsf{c}}\ \sigma',\mathtt{T}_c\\ \mathtt{T}_s=\langle R^{x.\mathtt{write}(x)}_{l_1}\mathtt{W}_{\tau_1}\rangle_{l'_1:\tau_1};\ldots;\langle R^{x.\mathtt{write}(x)}_{l_n}\mathtt{W}_{\tau_n}\rangle_{l'_n:\tau_n}\end{array}}{\sigma,m:\beta,l\leftarrow\mathtt{return}(t_c)\Downarrow^{\mathsf{c}}\sigma',\left(\mathtt{T}_s;\{\,\mathtt{T}_c\,\}^{m:\beta}_{(l'_1,\ldots,l'_n)}\right)}$$

$$\frac{\begin{array}{c}(\alpha,\mu,\chi,\mathtt{T})=\sigma\\ ([l_1,\ldots,l_n],\beta')=\mathtt{split}\,(\beta)\\ m:\beta',\mathtt{T}\rightsquigarrow\{\,\mathtt{T}_c\,\}^{m:\beta'}_{(l'_1,\ldots l'_n)},\mathtt{T}'\\ \alpha'=\alpha[l'_1\leftarrow\alpha[l_1],\ldots,l'_n\leftarrow\alpha[l_n]]\\ (\alpha',\mu),\chi\cup\{l'_1,\ldots,l'_n\},\{\,\mathtt{T}_c\,\}^{m:\beta'}_{(l'_1,\ldots l'_n)}\overset{\mathsf{c}}{\circlearrowright}\mathtt{T}'_c,\chi',(\alpha'',\mu')\\ \mathtt{T}_s=\langle R^{x.\mathtt{write}(x)}_{l_1}\mathtt{W}_{\tau_1}\rangle_{l'_1:\tau_1};\ldots;\langle R^{x.\mathtt{write}(x)}_{l_n}\mathtt{W}_{\tau_n}\rangle_{l'_n:\tau_n}\end{array}}{\sigma,m:\beta,l\leftarrow\mathtt{return}(t_c)\Downarrow^{\mathsf{c}}(\alpha'',\mu',\chi',\mathtt{T}'),(\mathtt{T}_s;\mathtt{T}'_c)}$$

$$\frac{\sigma,t_s\ \Downarrow^{\mathsf{s}}\ \sigma',v,\mathtt{T}_s\qquad\sigma',m:\beta,l\leftarrow[v/a]e_c\ \Downarrow^{\mathsf{c}}\ \sigma'',\mathtt{T}_c}{\sigma,m:\beta,l\leftarrow\mathtt{let}\ a:\tau\ \mathtt{be}\ t_s\ \mathtt{in}\ e_c\ \mathtt{end}\Downarrow^{\mathsf{s}}\sigma'',(\mathtt{T}_s;\mathtt{T}_c)}\ \textbf{(let)}$$

$$\frac{\sigma,m:!\,v\cdot\beta,l\leftarrow[v/x]e_c\ \Downarrow^{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,m:\beta,l\leftarrow\mathtt{let}\,!\,x:\tau\,\mathtt{be}\,!\,v\,\mathtt{in}\,e_c\,\mathtt{end}\Downarrow^{\mathsf{c}}\sigma',\mathtt{T}_c}\ \textbf{(let!)}$$

$$\frac{\sigma,m:?\,v\cdot\beta,l\leftarrow[v/x]e_c\ \Downarrow^{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,m:\beta,l\leftarrow\mathtt{let}\,?\,x:\tau\,\mathtt{be}\,?\,v\,\mathtt{in}\,e_c\,\mathtt{end}\Downarrow^{\mathsf{c}}\sigma',\mathtt{T}_c}\ \textbf{(let?)}$$

$$\frac{\sigma,m:a_1\times a_2\cdot\beta,l\leftarrow[v_1/a_1,v_2/a_2]e_c\ \Downarrow^{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,m:\beta,l\leftarrow\mathtt{let}\,a_1\times a_2\,\mathtt{be}\,v_1\times v_2\,\mathtt{in}\,e_c\,\mathtt{end}\Downarrow^{\mathsf{c}}v,\sigma',\mathtt{T}_c}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma,m:\mathtt{inl}\cdot\beta,l\leftarrow[v/a_1]e_c\ \Downarrow^{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,m:\beta,l\leftarrow\ \begin{array}{l}\mathtt{mcase}\ \mathtt{inl}_{\tau_1+\tau_2}v\ \mathtt{of}\\ \quad\mathtt{inl}\,(a_1:\tau_1)\Rightarrow e_c\\ \quad\mid\mathtt{inr}\,(a_2:\tau_2)\Rightarrow e'_c\end{array}\ \Downarrow^{\mathsf{c}}\sigma',\mathtt{T}_c}\ \textbf{(case)}$$

$$\frac{\sigma,m:\mathtt{inr}\cdot\beta,l\leftarrow[v/a_2]e_c\ \Downarrow^{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,m:\beta,l\leftarrow\ \begin{array}{l}\mathtt{mcase}\ \mathtt{inr}_{\tau_1+\tau_2}v\ \mathtt{of}\\ \quad\mathtt{inl}\,(a_1:\tau_1)\Rightarrow e_c\\ \quad\mid\mathtt{inr}\,(a_2:\tau_2)\Rightarrow e'_c\end{array}\ \Downarrow^{\mathsf{s}}v',\sigma',\mathtt{T}_c}\ \textbf{(case)}$$

Figure 25: Evaluation of changeable expressions.

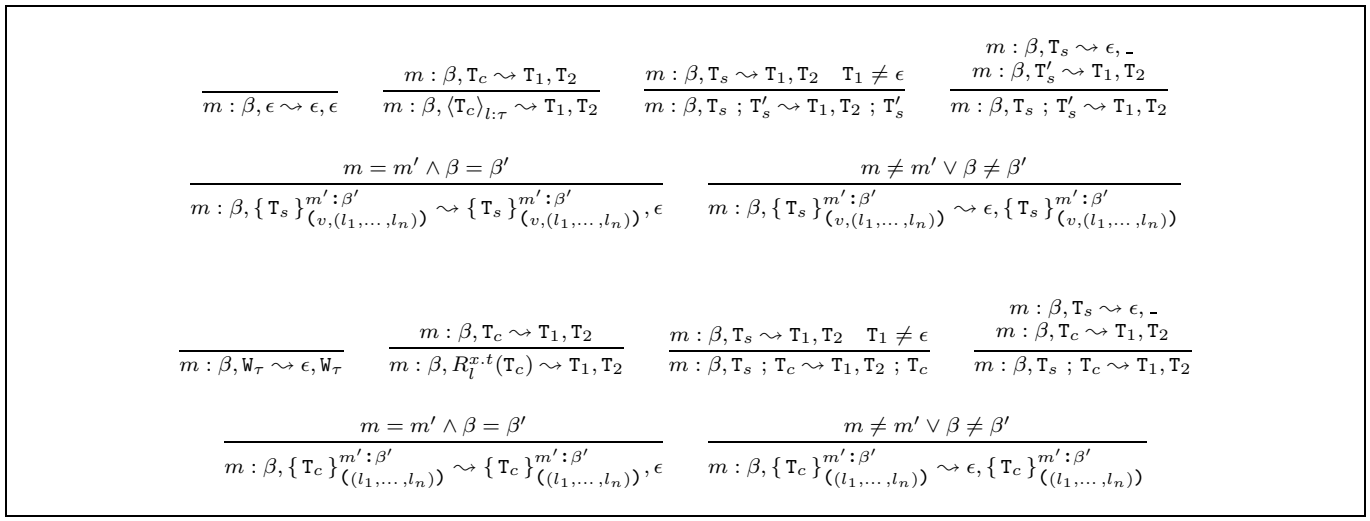

Figure 26: The rules for memo look up.

C.1 Change Propagation

We present a formal version of the change-propagation algorithm, which is informally described in Section 5. The algorithm extends the original change propagation algorithm [2] to support result re-use. Given a trace, a state ς, and a set of changed locations χ, the algorithm scans through the trace as it seeks for reads of changed locations. When such a read is found, the body of the read is re-evaluated to obtain a revised trace. Crucial point is that the re-evaluation of a read re-uses the trace of that read. In contrast, the original change propagation algorithm throws away the trace of the re-evaluated read [2]. Since re-evaluation can change the value of the target of the re-evaluated read, the target is added to the set of changed locations. Figure 27 shows the rules for change propagation.

The change propagation algorithm is given by these two judgments:

1. *Stable propagation*: $\varsigma, \chi, \mathtt{T}_s \overset{\mathsf{s}}{\curvearrowright} \mathtt{T}'_s, \chi', \varsigma'$
2. *Changeable propagation*: $\varsigma, \chi, l \leftarrow \mathtt{T}_c \overset{\mathsf{c}}{\curvearrowright} \mathtt{T}'_c, \chi', \varsigma'$

These judgments define the change-propagation for a stable trace, $\mathtt{T}_s$ (respectively, changeable trace, $\mathtt{T}_c$), with respect to a a set of changed locations χ, and state $\varsigma = (\alpha, \mu)$ consisting of a location store α, and function identifier store μ. For changeable propagation a target location, l, is maintained as in the changeable evaluation mode of IFL.

Given a trace, change propagation mimics the evaluation rule of IFL that originally generated that trace. To stress this correspondence, each rule is marked with the name of the evaluation rule to which it corresponds. For example, the propagation rule for the trace $\mathtt{T}_s$; $\mathtt{T}'_s$ mimics the `let` rule of the stable mode that gives rise to this trace.

Note that the purely functional change-propagation algorithm presented here scans the whole trace. Therefore, a direct implementation of this algorithm will run in time linear in the size of the trace. Performance can be improved by using side effects: since the change-propagation algorithm revises the trace by only replacing the changeable trace of re-evaluated reads, the re-evaluated reads can be replaced in place, while skipping the unaffected parts of the trace. This is how the ML implementation performs change propagation using a dynamic dependency graph as described in Section 5.

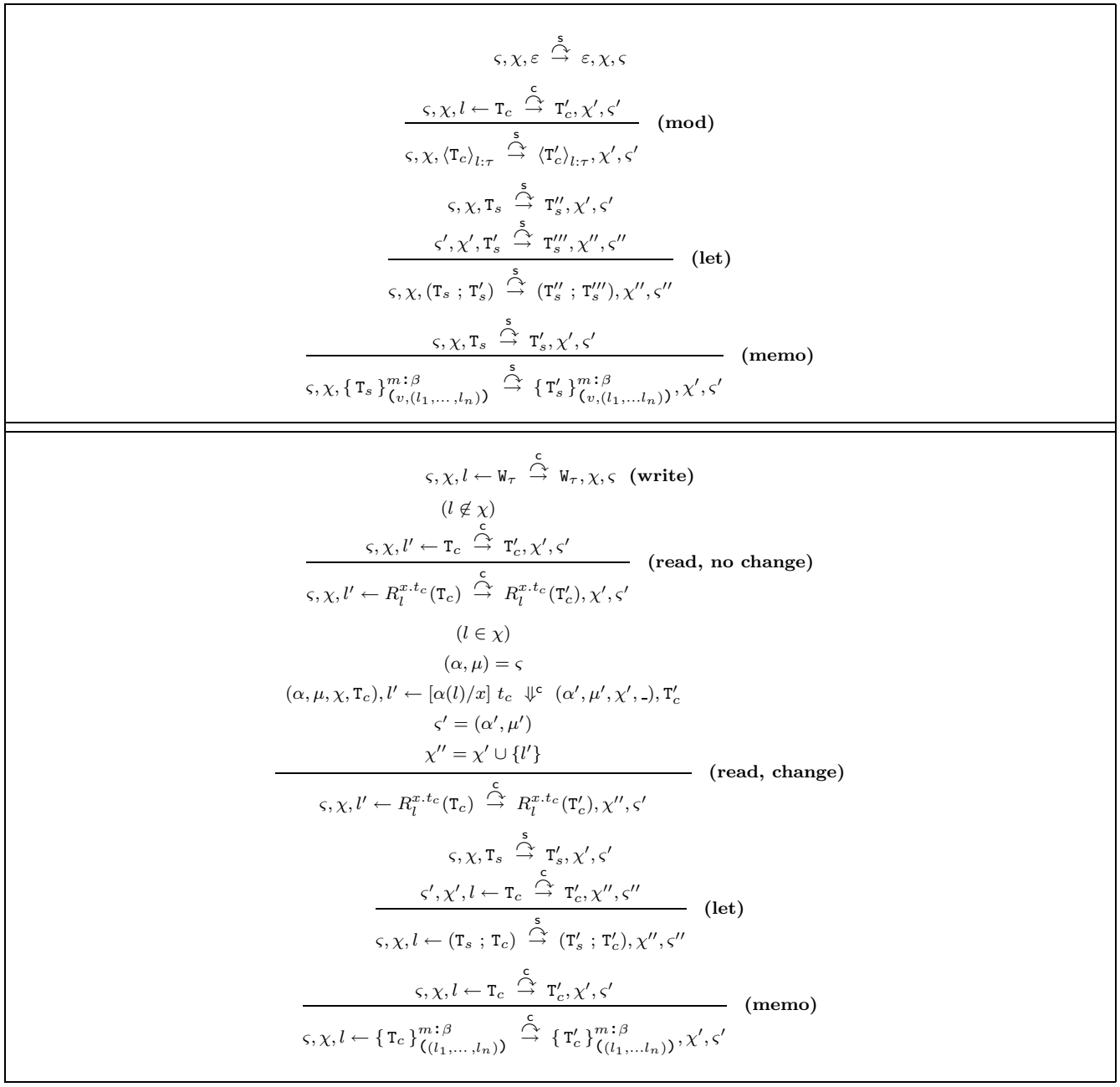

Figure 27: Change propagation for stable (top) and changeable (bottom) traces.