Adaptive Functional Programming*

Umut A. Acar† Guy E. Blelloch‡ Robert Harper§

May 5, 2006

Abstract

We present techniques for incremental computing by introducing adaptive functional programming. As an *adaptive* program executes, the underlying system represents the data and control dependences in the execution in the form of a *dynamic dependence graph*. When the input to the program changes, a change propagation algorithm updates the output and the dynamic dependence graph by propagating changes through the graph and re-executing code where necessary. Adaptive programs adapt their output to any change in the input, small or large.

We show that adaptivity techniques are practical by giving an efficient implementation as a small ML library. The library consists of three operations for making a program adaptive, plus two operations for making changes to the input and adapting the output to these changes. We give a general bound on the time it takes to adapt the output, and based on this, show that an adaptive Quicksort adapts its output in logarithmic time when its input is extended by one key.

To show the safety and correctness of the mechanism we give a formal definition of AFL, a call-by-value functional language extended with adaptivity primitives. The modal type system of AFL enforces correct usage of the adaptivity mechanism, which can only be checked at run time in the ML library. Based on the AFL dynamic semantics, we formalize the change-propagation algorithm and prove its correctness.

1 Introduction

Incremental computation concerns maintaining the input-output relationship of a program as the input of a program undergoes changes. Incremental computation is useful in situations where small input changes lead to relatively small changes in the output. In limiting cases one cannot avoid a complete re-computation of the output, but in many cases the results of the previous computation may be re-used to update output more quickly than a complete re-evaluation.

In this paper, we propose *adaptive functional programming* as a technique for incremental-computation. As an adaptive program executes, the underlying system represents the data and

*This research was supported in part by NSF grants CCR-9706572, CCR-0085982, and CCR-0122581.

†umut@tti-c.org. Toyota Technological Institute, Chicago IL.

‡blelloch@cs.cmu.edu. Carnegie Mellon University, Pittsburgh, PA.

§rwh@cs.cmu.edu. Carnegie Mellon University, Pittsburgh, PA.

control dependences in the execution via a *dynamic dependence graph*. When the input to the program changes, *a change-propagation algorithm* updates the dependence graph and the output by propagating changes through the graph and re-executing code where necessary. The input changes can take a variety of forms (insertions, deletions, etc.) and can be small or large.

Our proposed mechanism extends call-by-value functional languages with a small set of primitives to support adaptive programming. Apart from requiring that the host language be purely functional, we make no other restriction on its expressive power. In particular our mechanism is compatible with the full range of effect-free constructs found in ML. Our proposed mechanism has these strengths:

- **Generality:** It applies to any purely functional program. The programmer can build adaptivity into an application in a natural and modular way. The performance can be determined using analytical techniques but will depend on the particular application.
- **Flexibility:** It enables the programmer to control the amount of adaptivity. For example, a programmer can choose to make only one portion or aspect of a system adaptive, leaving the others to be implemented conventionally.
- **Simplicity:** It requires small changes to existing code. For example, the adaptive version of Quicksort presented in the next section requires only minor changes to the standard implementation.

Our adaptivity mechanism is based on the idea of a *modifiable reference* (or *modifiable*, for short) and three operations for creating (`mod`), reading (`read`), and writing (`write`) modifiables. A modifiable enables recording the dependence of one computation on the value of another. A modifiable reference is essentially a write-once reference cell that records the value of an expression whose value may change as a (direct or indirect) result of changes to the inputs. Any expression whose value can change must store its value in a modifiable reference; such an expression is said to be *changeable*. Expressions that are not changeable are said to be *stable*; stable expressions are not associated with modifiables.

Any expression that depends on the value of a changeable expression must express this dependence by explicitly reading the contents of the modifiable storing the value of that changeable expression. This establishes a data dependence between the expression reading that modifiable, called the *reader*, and the expression that determines the value of that modifiable, the *writer*. Since the value of the modifiable may change as a result of changes to the input, the reader must itself be deemed a changeable expression. This means that a reader cannot be considered stable, but may only appear as part of a changeable expression whose value is stored in some other modifiable.

By choosing the extent to which modifiables are used in a program, the programmer can control the extent to which it is able to adapt to change. For example, a programmer may wish to make a list manipulation program adaptive to insertions into and deletions from the list, but not under changes to the individual elements of the list. This can be represented in our framework by making only the "tail" elements of a list adaptive, leaving the "head" elements stable. However, once certain aspects are made changeable, all parts of the program that depend on those aspects are, by implication, also changeable.

The key to adapting the output to change of input is to record the dependencies between readers and writers that arise during the initial evaluation. These dependencies are maintained as a *dynamic dependence graph* where each node represents a modifiable, and each edge represents a read whose source is the modifiable being read and whose target is the modifiable being written.

Also, each edge is tagged with the corresponding reader, which is a closure. Whenever the source modifiable changes, the new value of the target is determined by re-evaluating the associated reader.

It is not enough, however, to maintain only this dependence graph connecting readers to writers. It is also essential to maintain an ordering on the edges and keep track of which edges (reads) are created during the execution of which other edges (i.e., which edges are within the dynamic scope of which other edges). We call this second relationship the *containment hierarchy*. The ordering among the edges enables us to re-evaluate readers in the same order as they were evaluated in the initial evaluation. The containment hierarchy enables us to identify and remove edges that become obsolete. This occurs, for example, when the result of a conditional inside a reader takes a different branch than the initial evaluation. One difficulty is maintaining the ordering and containment information during re-evaluation. We show how to maintain this information efficiently using time-stamps and an order-maintenance algorithm of Dietz and Sleator [11].

A key property of the proposed techniques is that the time for change propagation can be determined analytically. For example, we show in this paper that adaptive Quicksort updates its output in expected $O(\log n)$ time when its input is changed by a single insertion or deletion at the end. In other work [5], we describe an analysis technique, called *trace stability*, for bounding the time for change propagation under a class of input changes. The technique relies on representing executions via traces and measuring the distance between the traces of a program on similar inputs. A stability theorem states that the time for change propagation can be bounded in terms of the traces for the inputs before and after the change. The trace of an execution is the function call tree of the execution augmented with certain information.

2 Related Work

Many techniques have been proposed for incremental computation. The idea of using dependency graphs for incremental updates was introduced by Demers, Reps and Teitelbaum [10] in the context of attribute grammars. Reps then showed an algorithm to propagate a change optimally [29], and Hoover generalized the approach outside the domain of attribute grammars [19]. A crucial difference between this previous work and ours is that the previous work is based on static dependency graphs. Although they allow the graph to be changed by the modify step, the propagate step (i.e., the propagation algorithm) can only pass values through a static graph. This severely limits the types of adaptive computations that the technique handles [26]. Another difference is that they don't have the notion of forming the initial graph/trace by running a computation, but rather assume that it is given as input (often it naturally arises from the application). Yellin and Strom use the dependency graph ideas within the INC language [32], and extend it by having incremental computations within each of its array primitives. Since INC does not have recursion or looping, however, the dependency graphs remain static.

The idea behind memoization[8, 22, 23] is to remember function calls and re-use them when possible. Pugh [26], and Pugh and Teitelbaum [27] were the first to apply memoization to incremental computation. One motivation behind their work was a lack of general-purpose technique for incremental computation (previous techniques based on dependence graphs applied only to certain computations). Since Pugh and Teitelbaum's work, other researchers investigated applications of memoization to incremental computation [1, 21, 20, 20, 17, 18, 4]. The effectiveness of memoization critically depends on the particular application and the kinds of input changes being considered. In general, memoization alone is not likely to support efficient updates. For example, the best bound for list sorting using memoization is linear [21].

Other approaches to incremental computation are based on partial evaluation [15, 31]. Sun-

daresh and Hudak's approach [31] requires the user to fix the partition of the input that the program will be specialized on. The program is then partially evaluated with respect to this partition and the input outside the partition can be changed incrementally. The main limitation of this approach is that it allows input changes only within a predetermined partition [21, 16]. Field [16], and Field and Teitelbaum [15] present techniques for incremental computation in the context of lambda calculus. Their approach is similar to Hudak and Sundaresh's but they present formal reduction systems that use partially evaluated results optimally. We refer the reader to Ramalingam and Reps's excellent bibliography for a summary of other work on incremental computation [28].

The adaptivity techniques described in this paper have been extended and applied to some applications. Carlsson [9] gives an implementation of the ML library described in Section 4.9 in the Haskell language. Carlsson's implementation does not support laziness, but ensures certain safety properties of adaptive programs. Acar et al. [3] present an ML library that ensures similar safety properties and combines the adaptivity mechanism with memoization. As an application, Acar et. al [5] consider the dynamic-trees problem of Sleator and Tarjan [30]. They show that an adaptive version of the tree-contraction algorithm of Miller and Reif [24] yields an asymptotically efficient solution to the dynamic-trees problem. Acar, Blelloch, and Vittes [6] perform an experimental analysis of the approach by considering a broad set of applications involving dynamic trees; the results show that the approach is competitive in practice. Acar's thesis [2] describes a technique for combining adaptivity and memoization and shows that the combination can support incremental updates (asymptotically) efficiently for a reasonably broad range of applications.

3 Overview of the Paper

In Section 4 we illustrate the main ideas of adaptive functional programming in an algorithmic setting. We first describe how to implement an adaptive form of Quicksort in the Standard ML language based on the interface of a module implementing the basic adaptivity mechanisms. We then describe dynamic dependence graphs and the change-propagation algorithm and establish an upper bound for the running time of change propagation. Based on this bound, we prove the expected-$O(\log n)$ time bound for adaptive Quicksort under an extension to its input. We finish by briefly describing the implementation of the mechanism in terms of an abstract ordered list data structure. This implementation requires less than 100 lines of Standard ML code.

In Section 5 we define an adaptive functional programming language, called AFL, which is an extension of a simple call-by-value functional language with adaptivity primitives. The static semantics of AFL enforces properties that can only be enforced by run-time checks in our ML library. The dynamic semantics of AFL is given by an evaluation relation that maintains a record of the adaptive aspects of the computation, called a trace, which is used by the change propagation algorithm. Section 6 proves the type safety of AFL.

In Section 7 we present the change propagation algorithm in the framework of the dynamic semantics of AFL. The change-propagation algorithm interprets a trace to determine the correct order in which to propagate changes, and to determine which expressions need to be re-evaluated. The change-propagation algorithm also updates the containment structure of the computation, which is recorded in the trace. Using this presentation, we prove that the change-propagation algorithm is correct by showing that it yields essentially the same result as a complete re-evaluation with the changed inputs.

```
signature ADAPTIVE =
  sig
    type 'a mod
    type 'a dest
    type changeable

    val mod: ('a * 'a -> bool) ->
               ('a dest -> changeable) -> 'a mod
    val read: 'a mod * ('a -> changeable) -> changeable
    val write: 'a dest * 'a -> changeable

    val init: unit -> unit
    val change: 'a mod * 'a -> unit
    val propagate: unit -> unit
  end
```

Figure 1: Signature of the adaptive library.

4 A Framework for Adaptive Computing

We give an overview of our adaptive framework based on our ML library and an adaptive version of Quicksort.

4.1 The ML library

The signature of our adaptive library for ML is given in Figure 1. The library provides functions to create (`mod`), to read from (`read`), and to write to (`write`) modifiables, as well as meta-functions to initialize the library (`init`), change input values (`change`) and propagate changes to the output (`propagate`). The meta-functions are described later in this section. The library distinguishes between two "handles" to each modifiable: a *source* of type `'a mod` for reading from, and a *destination* of type `'a dest` for writing to. When a modifiable is created, correct usage of the library requires that it only be accessed as a destination until it is written, and then only be accessed as a source.[1] All changeable expressions have type `changeable`, and are used in a "destination passing" style—they do not return a value, but rather take a destination to which they write a value. Correct usage requires that a changeable expression ends with a `write`—we define "ends with" more precisely when we discuss time stamps. The destination written will be referred to as the *target (destination)*. The type changeable has no interpretable value.

The `mod` takes two parameters, a conservative comparison function and an *initializer*. A conservative comparison function returns `false` when the values are different but may return `true` or `false` when the values are the same. This function is used by the change-propagation algorithm to avoid unnecessary propagation. The `mod` function creates a modifiable and applies the initializer to the new modifiable's destination. The initializer is responsible for writing the modifiable. Its body is therefore a changeable expression, and correct usage requires that the body's target match the initializer's argument. When the initializer completes, `mod` returns the source handle of the modifiable it created.

[1]The library does not enforce this restriction statically, but can enforce it with run-time checks. In the following discussion we will use the term "correct usage" to describe similar restrictions where run-time checks are needed to check correctness. The language described in Section 5 enforces all these restrictions statically using a modal type system.

The `read` takes the source of a modifiable and a *reader*, a function whose body is changeable. The `read` accesses the contents of the modifiable and applies the reader to it. Any application of `read` is itself a changeable expression since the value being read could change. If a call R_a to `read` is within the dynamic scope of another call R_b to `read`, we say that R_a is *contained* within R_b. This relation defines a hierarchy on the reads, which we will refer to as the *containment hierarchy* (of reads).

```
1  datatype 'a list =
2      NIL
3    | CONS of ('a * 'a list)

4
5

6  fun filter f l =
7  let
8    fun filt(l) =
9      case l of
10       NIL => NIL
11     | CONS(h,r) =>
12         if f(h) then
13           CONS(h, filt(r))
14         else
15           filt(r)
16 in
17   filt(l)
18 end

19 fun qsort(l) =
20 let
21   fun qs(l,rest) =
22     case l of
23       NIL => rest
24     | CONS(h,r) =>
25       let
26         val l = filter (fn x => x < h) r
27         val g = filter (fn x => x >= h) r
28         val gs = qs(g,rest)
29       in
30         qs(l,CONS(h,gs))
31       end
32 in
33   qs(l,NIL)
34 end
```

```
1  datatype 'a list' =
2      NIL
3    | CONS of ('a * 'a list' mod)

4  fun modl f = mod (fn (NIL,NIL) => true
5                     | _ => false) f

6  fun filter' f l =
7  let
8    fun filt(l,d) = read(l, fn l' =>
9      case l' of
10       NIL => write(d, NIL)
11     | CONS(h,r) =>
12         if f(h) then write(d,
13           CONS(h, modl(fn d => filt(r,d))))
14         else
15           filt(r, d))
16 in
17   modl(fn d => filt(l, d))
18 end

19 fun qsort'(l) =
20 let
21   fun qs(l,rest,d) = read(l, fn l' =>
22     case l' of
23       NIL => write(d, rest)
24     | CONS(h,r) =>
25       let
26         val l = filter' (fn x => x < h) r
27         val g = filter' (fn x => x >= h) r
28         val gs = modl(fn d => qs(g,rest,d))
29       in
30         qs(l,CONS(h,gs),d)
31       end)
32 in
33   modl(fn d => qs(l,NIL,d))
34 end
```

Figure 2: The complete code for non-adaptive (left) and adaptive (right) versions of Quicksort.

4.2 Making an Application Adaptive

The transformation of a non-adaptive program to an adaptive program involves two steps. First, the input data structures are made "modifiable" by placing desired elements in modifiables. Second, the original program is updated by making the reads of modifiables explicit and placing the results of each expression that depends on a modifiable into another modifiable. This means that all values

that directly or indirectly depend on modifiable inputs are placed in modifiables. The changes to the program are therefore determined by what parts of the input data structure are made modifiable.

As an example, consider the code for a standard Quicksort, `qsort`, and an adaptive Quicksort, `qsort'`, as shown in Figure 2. To avoid linear-time concatenations, `qsort` uses an accumulator to store the sorted tail of the input list. The transformation is done in two steps. First, we make the lists "modifiable" by placing the tail of each list element into a modifiable as shown in lines 1,2,3 in Figure 2. (If desired, each element of the list could have been made modifiable as well; this would allow changing an element without changing the list structurally). The resulting structure, *a modifiable list*, allows the user to insert and delete items to and from the list. Second, we change the program so that the values placed in modifiables are accessed explicitly via a `read`. The adaptive Quicksort uses a `read` (line 21) to determine whether the input list `l` is empty and writes the result to a destination `d` (line 23). This destination belongs to the modifiable that is created by a call to `mod` (through `modl`) in line 28 or 33. These modifiables form the output list, which now is a modifiable list. The function `filter` is similarly transformed into an adaptive one, `filter'` (lines 6-18). The `modl` function takes an initializer and passes it to the `mod` function with a constant-time, conservative comparison function for lists. The comparison function returns `true`, if and only if both lists are `NIL` and returns `false` otherwise. This comparison function is sufficiently powerful to prove the $O(\log n)$ bound for adaptive Quicksort.

```
1  fun newElt(v) = modl(fn d => write(d,v))

2  fun fromList(nil) =
3       let val m = newElt(NIL)
4       in (m,m)
5       end
6  | fromList(h::r) =
7       let val (l,last) = fromList(r)
8       in (newElt(CONS(h,l)),last)
9       end

10 fun test(lst,v) =
11 let
12    val _ = init()
13    val (l,last) = fromList(lst)
14    val r = qsort'(l)
15 in
16    (change(last,CONS(v,newElt(NIL)));
17     propagate();
18     r)
19 end
```

Figure 3: Example of changing input and change propagation for Quicksort.

4.3 Adaptivity

An adaptive programs allows the programmer to change the input to the program and update the result by running change propagation. This process can be repeated as desired. The library provides the meta-function `change` to change the value of a modifiable and the meta-function `propagate` to propagate these changes to the output. Figure 3 illustrates an example. The function `fromList` converts a list to a modifiable list, returning both the modifiable list and its last element. The `test`

function first performs an initial evaluation of the adaptive Quicksort by converting the input list `lst` to a modifiable list `l` and sorting it into `r`. It then changes the input by adding a new key `v` to the end of `l`. To update the output `r`, `test` calls `propagate`. The update will result in a list identical to what would have been returned if `v` was added to the end of `l` before the call to `qsort`. In general, any number of inputs could be changed before running `propagate`.

4.4 Dynamic Dependence Graphs

The crucial issue is to support change propagation efficiently. To do this, an adaptive program, as it evaluates, creates a record of the adaptive activity in the form of a dependence graph augmented with additional information regarding the containment hierarchy and the evaluation order of reads. The augmented dependence graph is called a *dynamic dependence graph*. In a dynamic dependence graph, each node represents a modifiable and each edge represents a read. An evaluation of `mod` adds a node, and an evaluation of `read` adds an edge to the graph. In a `read`, the node being read becomes the source, and the target of the read (the modifiable that the reader finished by writing to) becomes the target. Each edge is tagged with the reader function, represented as a *closure* (i.e., a function and an environment that maps free variables to their values).

When the input to an adaptive program changes, a change-propagation algorithm updates the output and the dynamic dependence graph by propagating changes through the graph and re-executing the reads affected by the change. When re-evaluated with a changed source, a read can, due-to conditionals, take a different evaluation path than before. For example, it can create new reads and decide to skip previously evaluated reads. It is therefore critical for correctness that the newly created reads are inserted into the graph and the previously created reads are deleted from the graph. Insertions are performed routinely by the `read`'s. To support deletions, dynamic-dependence graphs maintain a *containment hierarchy* between reads. A read e is *contained* within another read e' if e was created during the execution of e'. During change propagation, the reads contained in a re-evaluated read are removed from the graph.

Containment hierarchy is represented using time-stamps. Each edge and node in the dynamic dependence graph is tagged with a *time-stamp* corresponding to its execution "time" in the sequential execution order. Time stamps are generated by the `mod` and `read` expressions. The time stamp of an edge is generated by the corresponding `read`, before the reader is evaluated, and the time stamp of a node is generated by the `mod` after the initializer is evaluated (the time corresponds to the initialization time). Correct usage of the library requires that the order of time stamps is independent of whether the `write` or `mod` generate the time stamp for the corresponding node. This is what we mean by saying that a changeable expression must end with a `write` to its target.

The time stamp of an edge is called its *start time* and the time stamp of the target of the edge is called the edge's *stop time*. The start and the stop time of the edge define the *time span* of the edge. Time spans are then used to identify the containment relationship of reads: a read R_a is contained in a read R_b if and only if the time span of the edge associated with R_a is within the time span of the edge associated with R_b. For now, we will represent time stamps with real numbers. We will, subsequently, show how the Dietz-Sleator Order-Maintenance Algorithm can be used to maintain time stamps efficiently [11].

We define a *dynamic dependence graph* (DDG) as a directed-acyclic graph (DAG) in which each edge has an associated reader and a time stamp, and each node has an associated value and time stamp. We say that a node (and corresponding modifiable) is an *input* if it has no incoming edges. Dynamic dependence graphs serve as an efficient implementation of the notion of traces that we formalize in Section 5. We therefore do not formalize dynamic dependence graphs here. A more

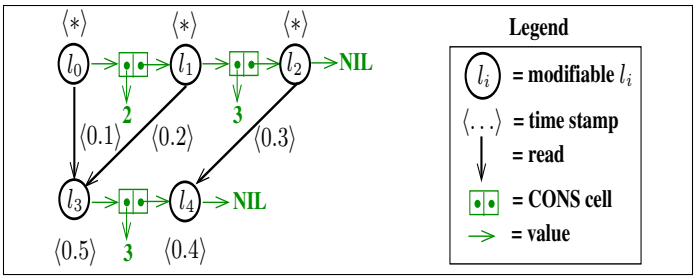

Figure 4: The DDG for an application of `filter'` to the modifiable list `2::3::nil`.

precise description of dynamic dependence graphs can be found elsewhere [5].

As an example for dynamic dependence graphs, consider the adaptive filter function `filter'` shown in Figure 2. The arguments to `filter'` consists of a function `f` and a modifiable list `l`; the results consists of a modifiable list that contains the items of `l` satisfying `f`. Figure 4 shows the dependence graph for an evaluation of `filter'` with the function `(fn x => x > 2)` and a modifiable input list of `2::3::nil`. The output is the modifiable list `3::nil`. Although not shown in the figure, each edge is also tagged with a reader. In this example, all edges have an instance of reader `(fn l' => case l' of ...)` (lines 8-15 of `qsort'` in Figure 2). The time stamps for input nodes are not relevant, and are marked with stars in Figure 4. We note that readers are closures, i.e., code with captured environments. In particular, each of the readers in our example have their source and their target in their environment.

4.5 Change Propagation

Given a dynamic dependence graph and a set of changed input modifiables, the *change-propagation algorithm* updates the DDG and the output by propagating changes through the DDG. The idea is to re-evaluate the reads that are affected by the input change in the sequential execution order. Re-executing the reads in the sequential execution order ensures that the source of an edge (read) is updated before the re-execution of that read. We say that an edge or read, is *affected* if its source has a different underlying value.

Figure 5 shows the change-propagation algorithm. The algorithm maintains a priority queue of affected edges. The queue is prioritized on the time stamp of each edge, and is initialized with the out-edges of the changed input values. Each iteration *updates an edge*, e, by re-evaluating the reader of e after deleting all nodes and edges that are contained in e from both the graph and queue. After the reader is re-evaluated the algorithm checks if the value of the target has changed (line 10) by using the conservative comparison function passed to `mod`. If the target has changed, the out-edges of the target are added to the queue to propagate that change.

As an example, consider an initial evaluation of `filter` whose dependence graph is shown in Figure 4. Now, suppose we change the modifiable input list from `2::3::nil` to `2::4::7::nil` by creating the modifiable list `4::7::nil` and changing the value of modifiable l_1 to this list, and run change propagation. The leftmost frame in Figure 6 shows the input change. The change-propagation algorithm starts by inserting the outgoing edge of l_1, (l_1,l_3), into the queue. The algorithm then removes the edge from the queue for re-execution. Before re-evaluating the reader of the edge, the algorithm establishes the current time-span as $\langle 0.2\rangle$-$\langle 0.5\rangle$, and deletes the nodes and edges contained in the edge from the DDG and the queue (which is empty) (middle frame in Figure 6). The algorithm then re-evaluates the reader `(fn l' => case l' of ...)` (8-15 in

Propagate Changes

```
    I is the set of changed inputs
    (V, E) = G is an DDG
1   Q := ⋃_{v∈I} outEdges(v)
2   while Q is not empty
3     e := deleteMin(Q)
4     (T_s, T_e) := timeSpan(e)
5     V := V − {v ∈ V | T_s < T(v) < T_e}
6     E' := {e' ∈ E | T_s < T(e') < T_e}
7     E := E − E'
8     Q := Q − E'
9     v' := apply(reader(e), val(src(e))) in time (T_s, T_e)
10    if v' ≠ val(target(e)) then
11      val(target(e)) = v'
12      Q := Q + outEdges(target(e))
```

Figure 5: The change-propagation algorithm.

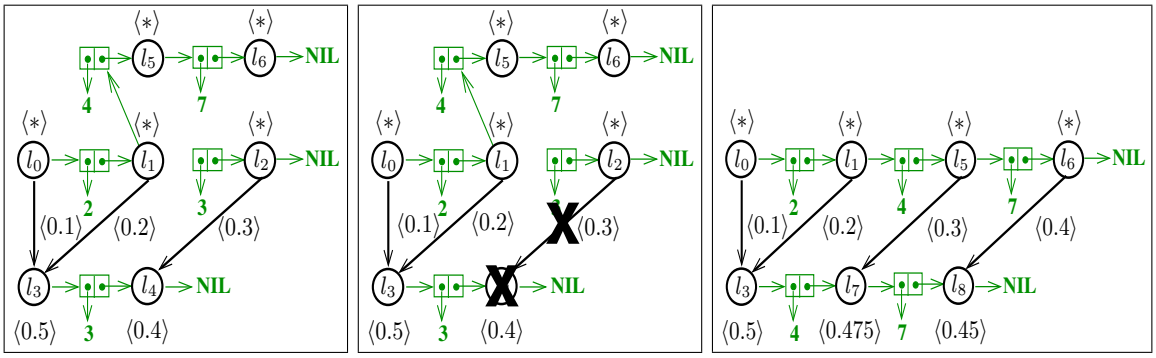

Figure 6: Snapshots of the DDG during change propagation.

Figure 2) in the time span ⟨0.2⟩-⟨0.5⟩. The reader walks through the modifiable list `4::7::nil` as it filters the items and writes the head of the result list to l_3 (the right frame in Figure 6). This creates two new edges, which are given the time stamps, ⟨0.3⟩, and ⟨0.4⟩. The targets of these edges, l_7 and l_8, are assigned the time stamps, ⟨0.475⟩, and ⟨0.45⟩, matching the order that they were initialized (these time stamps are otherwise chosen arbitrarily to fit in the range ⟨0.4⟩-⟨0.5⟩). Note that after change propagation, the modifiables l_2 and l_4 become unreachable and can be garbage collected.

The change-propagation algorithm deletes the reads that are contained in a re-evaluated read because such reads can be inconsistent with a from-scratch evaluation of the program on the changed input. Re-evaluating such a read can therefore change the semantics of the program, e.g., it can cause the result to be computed incorrectly, cause non-termination, or raise an exception (assuming the language supports exceptions as ML does). As an example, consider the `factorial` function shown in Figure 7. The program takes an integer modifiable `n` and a boolean modifiable `p` whose value is `true` if the value of `n` is positive and `false` otherwise. Consider evaluating the function with a positive `n` with `p` set to `true`. Now change `n` to negative two and `p` to `false`. This change

```
1  fun factorial (n:int mod, p:bool mod):int mod =
2    let
3      fun fact (n:int mod) =
4        if (n = 0) then 1
5        else n * fact(n-1)
6    in
7      mod (fn d =>
8           read (p, fn p' =>
9                    if (not p') then write (d,1)
10                   else read (n, fn n' => write (d, fact n'))))
11   end
```

Figure 7: The `factorial` function with a flag indicating the sign of the input.

will make the read on line 8 affected. With the change-propagation algorithm, the read on line 8 will be re-evaluated and one will be written to the result modifiable. Since the read on line 10 is contained in the read on line 8, it will be deleted; note that re-evaluating this read will result in non-termination by calling `fact` on negative two.

4.6 Implementing Change Propagation Efficiently

The change-propagation algorithm described above can be implemented efficiently using a standard representation of graphs, a standard priority-queue algorithm, and an order-maintenance algorithm for time stamps. The implementation of the DDG needs to support deleting an edge, a node, and finding the outgoing edges of a node. An adjacency list representation of DDG's where the outgoing edges of a node are maintained in a doubly-linked list supports these operations in constant time. To support the deletion of edges contained within a given time interval efficiently, the implementation maintains a time-ordered, doubly-linked list of all edges. With this representation inserting and deleting an edge, and finding the next edge all take constant time. The priority queue should support addition, deletion, and delete-minimum operations efficiently. A standard logarithmic-time priority-queue data structure is sufficient for our purposes.

A more interesting question is how to implement time-stamp operations efficiently. One option is to keep the time stamps in a list and tag each time stamp with a real number while ensuring that the list is sorted with respect to tags. The tag for a new time stamp is computed as the average of the tags of the time stamps immediately before and immediately after it. Time stamps are compared by comparing their tags. Unfortunately, this approach is not practical because it requires arbitrary precision real numbers. Another option is to drop the tags and compare two time stamps by comparing their positions in the list—the time stamp closer to the beginning of the list is smaller. This comparison operation is not efficient because it can take linear time in the length of the list. Another approach is to assign an integer rank to each time stamp such that nodes closer to the beginning of the list have smaller ranks. This enables constant time comparisons by comparing the ranks. The insertion algorithm, however, needs to some re-ranking to make space for a time stamps that is being inserted between two time stamps whose tags differ by one. Using integer ranks, Dietz and Sleator give two efficient data structures, called order-maintenance data structures, for maintaining time stamps [11]. The first data structure performs all operations in amortized constant time, the second more complicated data structures achieves worst-case constant time.

4.7 Performance of Change Propagation

We show an upper bound on the running time of change propagation. As discussed above, we assume an adjacency list representation for dynamic dependence graphs together with a time-ordered list of edges, a priority queue that can support insertions, deletions, and remove-minimum operations in logarithmic time, and an order-maintenance structure that supports insert, delete, compare operations in constant time. We present a more precise account of the performance of change propagation elsewhere [5].

We define several performance measures for change propagation. Consider running the change-propagation algorithm, and let A denote the set of all affected edges. Of these edges, some of them participate in an edge update (are re-evaluated), and the others are deleted because they are contained in an updated edge. We refer to the set of updated edges as A_u. For an updated edge $e \in A_u$, let $|e|$ denote the re-evaluation time (complexity) of the reader associated with e assuming that `mod`, `read`, `write`, take constant time, and let $||e||$ denote the number of time stamps created during the initial evaluation of e. Let q be the maximum size of the priority queue at any time during the algorithm. Theorem 1 bounds the time of a propagate step.

Theorem 1 (Propagate)
Change propagation takes time

$$O\left(\sum_{e \in A_u} (|e| + ||e||) + |A| \log q\right).$$

Proof: The time for propagate can be partitioned into four items: (1) re-evaluation of readers, (2) creation of time stamps, (3) deletion of time stamps and contained edges, and (4) insertions/deletions into/from from the priority queue.

Re-evaluation of the readers (1) takes $\sum_{e \in A_u} |e|$ time. The number of time stamps created during the re-evaluation of a reader is no greater than the time it takes to re-evaluate the reader. Since creating one time stamp takes constant time, the time spent for creating all time stamps (2) is $O(\sum_{e \in A_u} |e|)$. Determining a time stamp to delete, deleting the time stamp and the corresponding node or edge from the DDG and the time-ordered doubly-linked edge list takes constant time per edge. Thus total time for the deletions (3) is $O(\sum_{e \in A_u} ||e||)$. Since each edge is added to the priority queue once and deleted from the queue once, the time for maintaining the priority queue (4) is $O(|A| \log q)$. ■

4.8 Performance of Adaptive Quicksort

We analyze the change-propagation time for Quicksort when the input list is modified by adding a new key at the end. The analysis is based on the bound given in Theorem 1.

Figure 8 shows the intuition behind the proof. Each circle represents a recursive call to Quicksort and each rectangle represents a the two calls to `filter` along with the recursive calls; the shorter the rectangle, the smaller the input. The dark circles and squares show the calls that would be affected by some insertion at the end of the input list. The key here is that each affected call takes constant time to re-evaluate and no more than two calls to `filter` are affected at each level (assuming for counting purposes that all recursive calls to filter have the same level as the `qsort` call that calls `filter`). Only one call to `qsort` is affected at the bottom level. Figure 8 highlights the path along which affected calls occur.

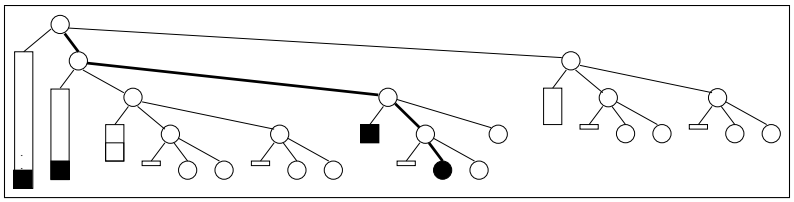

Figure 8: The function-call tree for Quicksort.

Theorem 2
Change propagation updates the output of adaptive Quicksort in $O(\log n)$ time after the input list of length n is extended with a new key at the end.

Proof: The proof is by induction on the height h of a call tree representing just the calls to `qs`. When the input is extended, the value of the last element l_n of the list is changed from `NIL` to `CONS(`v`,`l_{n+1}`)`, where the value of l_{n+1} is `NIL` and v is the new key. The induction hypothesis is that in change propagation on an input tree of height h, the number of affected reads is at most $2h$ ($|A| \leq 2h$ and $A_u = A$), each reader takes constant time to re-evaluate ($\forall e \in A, |e| = O(1)$), the time span of a reader contains no other time stamps ($\forall e \in A, ||e|| = 0$), and the maximum size of the priority queue is 4 ($q \leq 4$).

In the base case, we have $h = 1$, and the call tree corresponds to an evaluation of `qs` with an empty input list. The only read of l_n is the outer read in `qs`. The change propagation algorithm will add the corresponding edge to the priority queue, and then update it. Now that the list has one element, the reader will make two calls to `filter` and two calls to `qs'` both with empty input lists. This takes constant time and does not add any edges to the priority queue. There are no time stamps in the time span of the re-evaluated edge and the above bounds hold.

For the inductive case assume that the hypothesis holds for trees up to height $h-1$, and consider a tree with height $h > 1$. Now, consider the change propagation starting with the root call to `qs`. The list has at least one element in it, therefore the initial read does not read the tail l_n. The only two functions that use the list are the two calls to `filter'`, and these will both read the tail in their last recursive call. Therefore, during change propagation these two reads (edges) become affected and will be added to the queue. When the edges are re-evaluated, one will write `NIL` to its target and will not change it. Re-evaluating the other reader will replace `NIL` with `CONS(`v`,`l_{n+1}`)`, and therefore extend the list that becomes input to the next level recursive call. Re-evaluating both readers takes constant time and the update deletes no time stamps. Re-execution of the two edges will change the input to one of the two recursive calls to `qs`—the change will be an extension at the end. Since the call tree of the affected `qs` has depth at most $d - 1$, the induction hypothesis applies. Thus, $|e| = O(1)$ and $||e|| = 0$ for all affected edges. Furthermore, the total number of affected edges is $|A| \leq 2(d-1) + 2 = 2d$ and all edges are re-evaluated ($A_u = A$). To see that $q \leq 4$, note that the queue contains edges from at most 2 different `qs` calls and there are at most 2 edges affected from each call.

It is known that the expected height of the call tree is $O(\log n)$ (expectation is over all inputs). Thus we have: $\mathrm{E}[|A|] = O(\log n)$, $A = A_u$, $q = 4$, and $\forall e \in A, |e| = O(1), ||e|| = 0$. Thus by taking the expectation of the formula given in Theorem 1 and plugging in these values gives expected $O(\log n)$ time for propagate. ■

Note that this theorem holds only for changes at the end of the input list. Changes at the start

```
signature ORDERED_LIST = sig
  type t

  val init :  unit -> t            (* Initialize *)
  val compare:  t*t -> order       (* Compare two nodes *)
  val insert :  t ref -> t         (* Insert a new node *)
  val spliceOut:  t*t -> unit      (* Splice interval out *)
  val isSplicedOut:  t -> bool     (* Is the node spliced?  *)
end
```

Figure 9: The signature of an ordered list.

or the middle are more challenging; we show how to handle such changes efficiently elsewhere [2].

4.9 The ML Implementation

We present an implementation of our adaptive mechanism in ML. The implementation is based on a library for ordered lists, which is an instance of the order-maintenance problem, and a standard priority queue. In the ordered-list interface (shown in Figure 9), `spliceOut` deletes all time stamps between two given time stamps and `isSplicedOut` returns `true` if the time stamp has been deleted and `false` otherwise.

Figure 10 shows the code for the ML implementation. The implementation differs somewhat from the algorithm described earlier, but the asymptotic performance remains the same. The `edge` and `node` types correspond to edges and nodes in the DDG. The reader and time-span are represented explicitly in the `edge` type, but the source and destination are implicit in the reader. In particular the reader starts by reading the source, and ends by writing to the destination. The node consists of the corresponding modifiable's value (`value`), its out-edges (`outEdges`), and a write function (`wrt`) that implements writes or changes to the modifiable. A time stamp is not needed since edges keep both start and stop times. The `currentTime` is used to help generate the sequential time stamps, which are generated for the edge on line 27 and for the node on line 22 by the write operation.

Some of the tasks assigned to the change-propagate loop in Figure 5 are performed by the write operation in the ML code. This includes the functionality of lines 10–12 in Figure 5, which are executed by lines 17–20 in the ML code. Another important difference is that the deletion of contained edges is done lazily. Instead of deleting edges from the priority queue and from the graph immediately, the time stamp of the edge is marked as affected (by being removed from the ordered-list data structure), and is deleted when it is next encountered. This can be seen in line 37.

We note that the implementation given does not include sufficient run-time checks to verify "correct usage". For example, the code does not verify that an initializer writes its intended destination. The code, however, does check for a read before write.

```
1  structure Adaptive :> ADAPTIVE =
2  struct
3    type changeable = unit
4    exception unsetMod
5    type edge = {reader:(unit -> unit), timeSpan:(Time.t * Time.t)}
6    type 'a node = {value:(unit -> 'a) ref, wrt:('a -> unit) ref, outEdges:edge list ref}
7    type 'a mod = 'a node
8    type 'a dest = 'a node
9    val currentTime = ref(Time.init())
10   val PQ = ref(Q.empty)                                          (* Priority queue *)
11   fun init() = (currentTime := Time.init(); PQ := Q.empty)
12   fun mod cmp f =
13     let val value = ref(fn() => raise unsetMod)
14         val wrt = ref(fn(v) => raise unsetMod)
15         val outEdges = ref(nil)
16         val m = {value=value, wrt=wrt, outEdges=outEdges}
17         fun change t v = (if cmp(v,(!value)()) then ()
18                           else (value := (fn() => v);
19                                 List.app (fn x => PQ := Q.insert(x,!PQ)) (!outEdges);
20                                 outEdges := nil);
21                           currentTime := t)
22         fun write(v) = (value := (fn() => v); wrt := change(Time.insert(currentTime));
23         val _ = wrt := write
24     in f(m); m end
25   fun write({wrt, ...}:'a dest, v) = (!wrt)(v)
26   fun read({value, outEdges, ...}:'a mod, f) =
27     let val start = Time.insert(currentTime)
28         fun run() = (f((!value)());
29                      outEdges := {reader=run, timeSpan=(start,(!currentTime))}::(!outEdges))
30     in run() end
31   fun change(l, v) = write(l, v)
32   fun propagate'() =
33     if (Q.isEmpty(!PQ)) then ()
34     else let val (edge, pq) = Q.deleteMin(!PQ)
35              val _ = PQ := pq
36              val {reader=f,timeSpan=(start,stop)} = edge
37          in if (Time.isSplicedOut start) then propagate'()     (* Deleted read, discard.*)
38             else (Time.spliceOut(start,stop);                  (* Splice out *)
39                   currentTime := start;
40                   f();                                         (* Rerun the read *)
41                   propagate'())
42          end
43   fun propagate() =
44     let val ctime = !currentTime
45     in (propagate'(); currentTime := ctime) end
46 end
```

Figure 10: The implementation of the adaptive library.

5 An Adaptive Functional Language

In the first part of the paper, we described an adaptivity mechanism in an algorithmic setting. The purpose was to introduce the basic concepts of adaptivity and show that the mechanism can be implemented efficiently. We now turn to the question of whether the proposed mechanism is sound. To this end, we present a small, purely functional language, called AFL, with primitives for adaptive computation. AFL ensures correct usage of the adaptivity mechanism statically by using a modal type system and employing implicit "destination passing."

The adaptivity mechanisms of AFL are similar to those of the adaptive library presented in Section 4. The chief difference is that the target of a changeable expression is implicit in AFL. Implicit passing of destinations is critical for ensuring various safety properties of the language.

AFL does not include analogues of the meta-operations for making and propagating changes as in the ML library. Instead, we give a direct presentation of the change-propagation algorithm in Section 7, which is defined in terms of the dynamic semantics of AFL given here. As with the ML implementation, the dynamic semantics must keep a record of the adaptive aspects of the computation. Rather than use DDG's, however, the semantics maintains this information in the form of a trace, which guides the change propagation algorithm. The trace representation simplifies the proof of correctness of the change propagation algorithm given in Section 7.

5.1 Abstract Syntax

The abstract syntax of AFL is given in Figure 11. We use the meta-variables x, y, and z (and variants) to range over an unspecified set of variables, and the meta-variable l (and variants) to range over a separate, unspecified set of locations—the locations are modifiable references. The syntax of AFL is restricted to "2/3-cps", or "named form", to streamline the presentation of the dynamic semantics.

The types of AFL include the base types `int` and `bool`; the stable function type, $\tau_1 \xrightarrow{\mathsf{S}} \tau_2$; the changeable function type, $\tau_1 \xrightarrow{\mathsf{C}} \tau_2$; and the type τ `mod` of modifiable references of type τ. Extending AFL with product, sum, recursive, or polymorphic types presents no fundamental difficulties, but they are omitted here for the sake of brevity.

Expressions are classified into two categories, the *stable* and the *changeable*. The value of a stable expression is not sensitive to modifications to the inputs, whereas the value of a changeable expression may, directly or indirectly, be affected by them. The familiar mechanisms of functional programming are embedded in AFL as stable expressions. These include basic types such as integers and booleans, and a sequential `let` construct for ordering evaluation. Ordinary functions arise in AFL as *stable functions*. The body of a stable function must be a stable expression; the application of a stable function is correspondingly stable. The stable expression $\texttt{mod}_\tau\ e_c$ allocates a new modifiable reference whose value is determined by the changeable expression e_c. Note that the modifiable itself is stable, even though its contents is subject to change.

Changeable expressions are written in destination-passing style, with an implicit target. The changeable expression $\texttt{write}_\tau(v)$ writes the value v of type τ into the target. The changeable expression `read` v `as` x `in` e_c `end` binds the contents of the modifiable v to the variable x, then continues evaluation of e_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 expression. The sequential `let` construct allows for the inclusion of stable sub-computations in changeable mode. Finally, conditionals with changeable branches are themselves changeable.

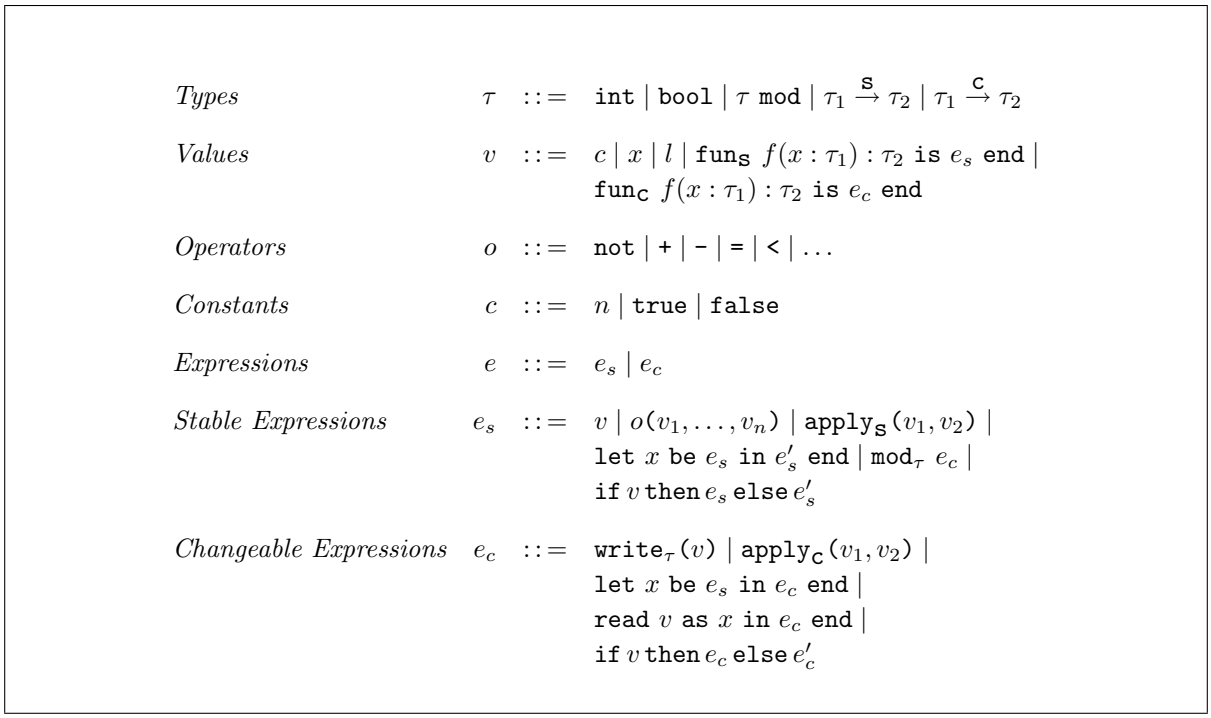

Figure 11: The abstract syntax of AFL.

```
fun sum (l:int modlist):int mod =
  let
    fun sum' (l:int modlist,
              s:int,
              dest:int mod):changeable =
      read (l, fn l' =>
        case l' of
          NIL => write (dest,s)
        | CONS(h,t) => sum' (t, s+h, dest)
      )
  in
    mod (fn d => sum' (l,0,d))
  end
```

```
funS sum (l:int modlist):int mod is
  let
    sum'' be funC sum' (l:int modlist,
                        s:int):int is

      read l as l' in
        case l' of
          NIL => writeint(s)
        | CONS(h,t) => applyC (sum',(t,s+h))
      end
  in
    modint applyC (sum'',(l,0))
  end
```

Figure 12: Function sum written with the ML library (left), and in AFL(right).

As an example, consider a function that sums up the keys in a modifiable list. Such a function could be written by traversing the list and accumulating a sum, which is written to the destination at the end. The code for this function using our ML library (Section 4) is shown in Figure 12 on the left. Note that all recursive calls to the function `sum'` share the same destination. The code for the `sum` function in AFL is shown in Figure 12 on the right assuming constructs for supporting lists and pattern matching. The critical difference between the two implementations is that in AFL, destinations are passed implicitly by making `sum'` a changeable function—all recursive calls to `sum'` share the same destination, which is created by `sum`.

The advantage to sharing of destinations is performance. Consider for example calling `sum` on some list and changing the list by an insertion or deletion at the end. Propagating this change will

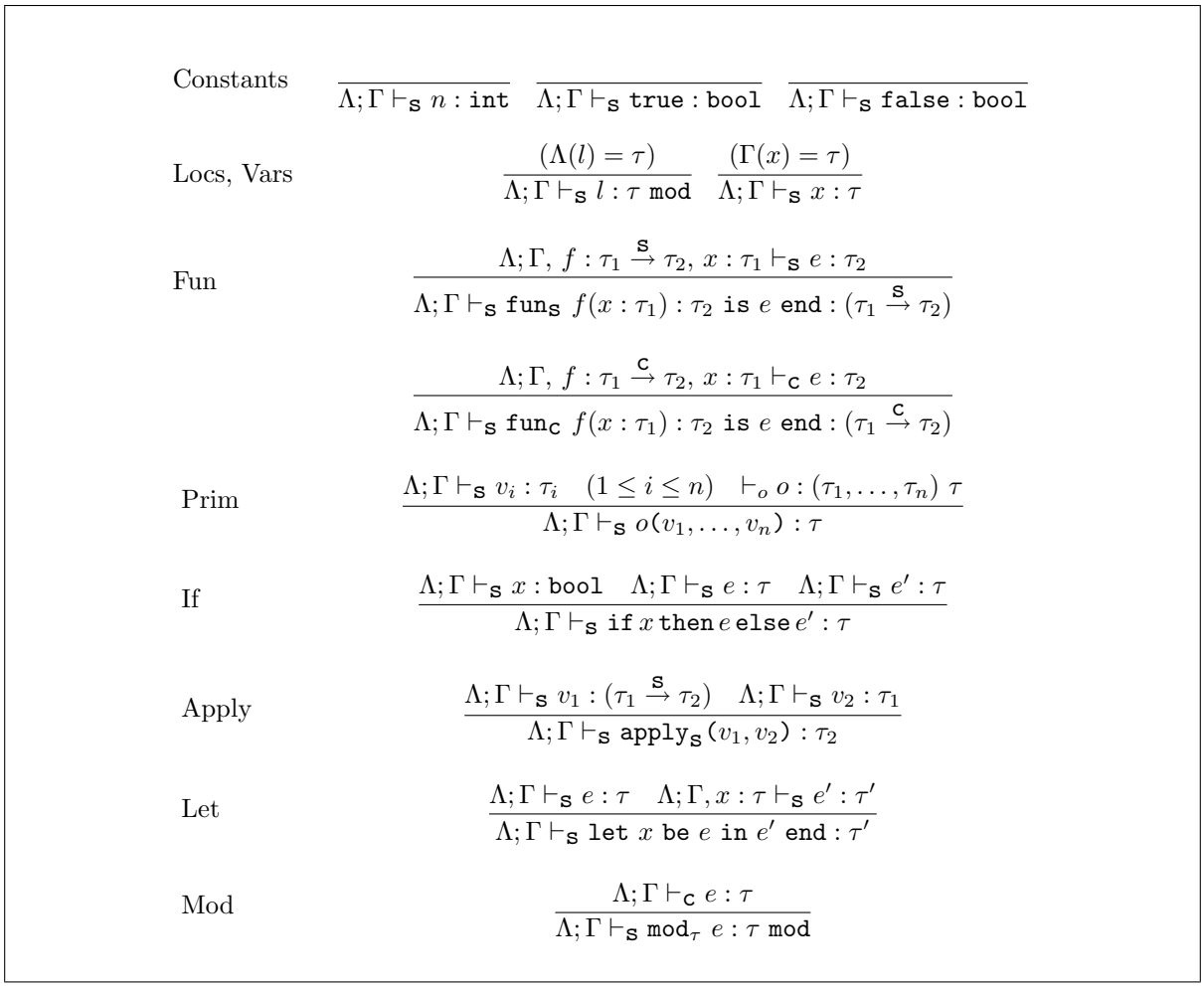

Figure 13: Typing of stable expressions.

take constant time and the result will be updated in constant time. If instead, each recursive call to `sum'` created its own destination and copied the result from the recursive call to its destination, then this change will propagate up the recursive-call tree and will take linear time. This is the motivation for including changeable functions in the AFL language.

5.2 Static Semantics

The AFL type system is inspired by the type theory of modal logic given by Pfenning and Davies [25]. We distinguish two modes, the *stable* and the *changeable*, corresponding to the distinction between terms and expressions, respectively, in Pfenning and Davies' work. However, they have no analogue of our changeable function type, and do not give an operational interpretation of their type system.

The judgement $\Lambda;\Gamma \vdash_{\mathtt{S}} e : \tau$ states that e is a well-formed stable expression of type τ, relative to Λ and Γ. The judgement $\Lambda;\Gamma \vdash_{\mathtt{C}} e : \tau$ states that e is a well-formed changeable expression of type τ, relative to Λ and Γ. Here Λ is a *location typing* and Γ is a *variable typing*; these are finite functions assigning types to locations and variables, respectively. (In Section 6 we will impose additional structure on location typings that will not affect the definition of the static semantics.)

The typing judgements for stable and changeable expressions are shown in Figures 13 and 14

$$\text{Write} \quad \frac{\Lambda;\Gamma \vdash_{\mathtt{S}} v : \tau}{\Lambda;\Gamma \vdash_{\mathtt{C}} \mathtt{write}_{\tau}(v) : \tau}$$

$$\text{If} \quad \frac{\Lambda;\Gamma \vdash_{\mathtt{S}} x : \mathtt{bool} \quad \Lambda;\Gamma \vdash_{\mathtt{C}} e : \tau \quad \Lambda;\Gamma \vdash_{\mathtt{C}} e' : \tau}{\Lambda;\Gamma \vdash_{\mathtt{C}} \mathtt{if}\, x\, \mathtt{then}\, e\, \mathtt{else}\, e' : \tau}$$

$$\text{Apply} \quad \frac{\Lambda;\Gamma \vdash_{\mathtt{S}} v_1 : (\tau_1 \xrightarrow{\mathtt{C}} \tau_2) \quad \Lambda;\Gamma \vdash_{\mathtt{S}} v_2 : \tau_1}{\Lambda;\Gamma \vdash_{\mathtt{C}} \mathtt{apply}_{\mathtt{C}}(v_1, v_2) : \tau_2}$$

$$\text{Let} \quad \frac{\Lambda;\Gamma \vdash_{\mathtt{S}} e : \tau \quad \Lambda;\Gamma, x : \tau \vdash_{\mathtt{C}} e' : \tau'}{\Lambda;\Gamma \vdash_{\mathtt{C}} \mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ e'\ \mathtt{end} : \tau'}$$

$$\text{Read} \quad \frac{\Lambda;\Gamma \vdash_{\mathtt{S}} v : \tau\ \mathtt{mod} \quad \Lambda;\Gamma, x : \tau \vdash_{\mathtt{C}} e : \tau'}{\Lambda;\Gamma \vdash_{\mathtt{C}} \mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end} : \tau'}$$

Figure 14: Typing of changeable expressions.

respectively. For primitive functions, we assume a typing relation o. For stable expression, the interesting rules are the `mod` and the changeable functions. The bodies of these expressions are changeable expressions and therefore they are typed in the changeable mode. For changeable expressions, the interesting rule is the `let` rule. The body of `let` is a changeable expression and thus typed in the changeable mode; the expression bound, however, is a stable expression and thus typed in the stable mode. The `mod` and `let` rules therefore provide inclusion between two modes.

5.3 Dynamic Semantics

The evaluation judgements of AFL have one of two forms. The judgement $\sigma, e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}_s$ states that evaluation of the stable expression e, relative to the input store σ, yields the value v, the trace $\mathtt{T}_s$, and the updated store σ'. The judgement $\sigma, l \leftarrow e \Downarrow^{\mathtt{C}} \sigma', \mathtt{T}_c$ states that evaluation of the changeable expression e, relative to the input store σ, writes its value to the target l, and yields the trace $\mathtt{T}_c$ and the updated store σ'.

In the dynamic semantics, a *store*, σ, is a finite function mapping each location in its domain, $\mathrm{dom}(\sigma)$, to either a value v or a "hole" $\square$. The *defined domain*, $\mathrm{def}(\sigma)$, of σ consists of those locations in $\mathrm{dom}(\sigma)$ not mapped to $\square$ by σ. When a location is created, it is assigned the value $\square$ to reserve that location while its value is being determined. With a store σ, we associate a location typing Λ and write $\sigma : \Lambda$, if the store satisfies the typing Λ. This is defined formally in Section 6.

A *trace* is a finite data structure recording the adaptive aspects of evaluation. The abstract syntax of traces is given by the following grammar:

$$\begin{array}{lllll} \textit{Trace} & \mathtt{T} & ::= & \mathtt{T}_s \mid \mathtt{T}_c \\ \textit{Stable} & \mathtt{T}_s & ::= & \epsilon \mid \langle \mathtt{T}_c \rangle_{l:\tau} \mid \mathtt{T}_s\ ;\ \mathtt{T}_s \\ \textit{Changeable} & \mathtt{T}_c & ::= & \mathtt{W}_\tau \mid R_l^{x.e}(\mathtt{T}_c) \mid \mathtt{T}_s\ ;\ \mathtt{T}_c \end{array}$$

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 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.

A changeable trace has one of three 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 $R_l^{x.e}(\mathtt{T}_c)$ 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.

We define the *domain* $\mathrm{dom}(\mathtt{T})$ of a trace $\mathtt{T}$ as the set of locations read or written in the trace $\mathtt{T}$. The *defined domain* $\mathrm{def}(\mathtt{T})$ of a trace $\mathtt{T}$ is the set of locations written in the trace $\mathtt{T}$. Formally the domain and the defined domain of traces are defined as

$$
\begin{array}{lcllcl}
\mathrm{def}(\varepsilon) & = & \emptyset & \mathrm{dom}(\varepsilon) & = & \emptyset \\
\mathrm{def}(\langle \mathtt{T}_c \rangle_{l:\tau}) & = & \mathrm{def}(\mathtt{T}_c) \cup \{l\} & \mathrm{dom}(\langle \mathtt{T}_c \rangle_{l:\tau}) & = & \mathrm{dom}(\mathtt{T}_c) \cup \{l\} \\
\mathrm{def}(\mathtt{T}_s \; ; \; \mathtt{T}'_s) & = & \mathrm{def}(\mathtt{T}_s) \cup \mathrm{def}(\mathtt{T}'_s) & \mathrm{dom}(\mathtt{T}_s \; ; \; \mathtt{T}'_s) & = & \mathrm{dom}(\mathtt{T}_s) \cup \mathrm{dom}(\mathtt{T}'_s) \\
\mathrm{def}(\mathtt{W}_\tau) & = & \emptyset & \mathrm{dom}(\mathtt{W}_\tau) & = & \emptyset \\
\mathrm{def}(R_l^{x.e}(\mathtt{T}_c)) & = & \mathrm{def}(\mathtt{T}_c) & \mathrm{dom}(R_l^{x.e}(\mathtt{T}_c)) & = & \mathrm{dom}(\mathtt{T}_c) \cup \{l\} \\
\mathrm{def}(\mathtt{T}_s \; ; \; \mathtt{T}_c) & = & \mathrm{def}(\mathtt{T}_s) \cup \mathrm{def}(\mathtt{T}_c) & \mathrm{dom}(\mathtt{T}_s \; ; \; \mathtt{T}_c) & = & \mathrm{dom}(\mathtt{T}_s) \cup \mathrm{dom}(\mathtt{T}_c).
\end{array}
$$

The dynamic dependency graphs described in Section 4 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 DDG representation. The containment hierarchy is directly represented by the structure of the trace. Specifically, the trace $\mathtt{T}_c$ (and any read in $\mathtt{T}_c$) is contained within the read trace $R_l^{x.e}(\mathtt{T}_c)$.

Stable Evaluation. The evaluation rules for stable expressions are given in Figure 15. Most of the rules are standard for a store-passing semantics. For example, the `let` rule sequences evaluation of its two expressions, and performs binding by substitution. Less conventionally, it yields a trace consisting of the sequential composition of the traces of its sub-expressions.

The most interesting rule is the evaluation of $\mathtt{mod}_\tau \; e$. Given a store σ, a fresh location l is allocated and initialized to $\square$ prior to evaluation of e. The sub-expression e is evaluated in changeable mode, with l as the target. Pre-allocating l ensures that the target of e is not accidentally re-used during evaluation; the static semantics ensures that l cannot be read before its contents is set to some value v.

Each location allocated during the evaluation a stable expression is recorded in the trace and is written to: If $\sigma, e \; \Downarrow^{\mathtt{S}} \; v, \sigma', \mathtt{T}_s$, then $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma) \cup \mathrm{def}(\mathtt{T}_s)$, and $\mathrm{def}(\sigma') = \mathrm{def}(\sigma) \cup \mathrm{def}(\mathtt{T}_s)$. Furthermore, all locations read during evaluation are defined in the store, $\mathrm{dom}(\mathtt{T}_s) \subseteq \mathrm{def}(\sigma')$.

Changeable Evaluation. The evaluation rules for changeable expressions are given in Figure 16. The `let` rule is similar to the corresponding rule in stable mode, except that the bound expression, e, is evaluated in stable mode, whereas the body, e', is evaluated in changeable mode. The `read` expression substitutes the binding of location l in the store σ for the variable x in e, and continues evaluation in changeable mode. The read is recorded in the trace, along with the expression that employs the value read. The `write` rule simply assigns its argument to the target. Finally, application of a changeable function passes the target of the caller to the callee, avoiding the need to allocate a fresh target for the callee and a corresponding read to return its value to the caller.

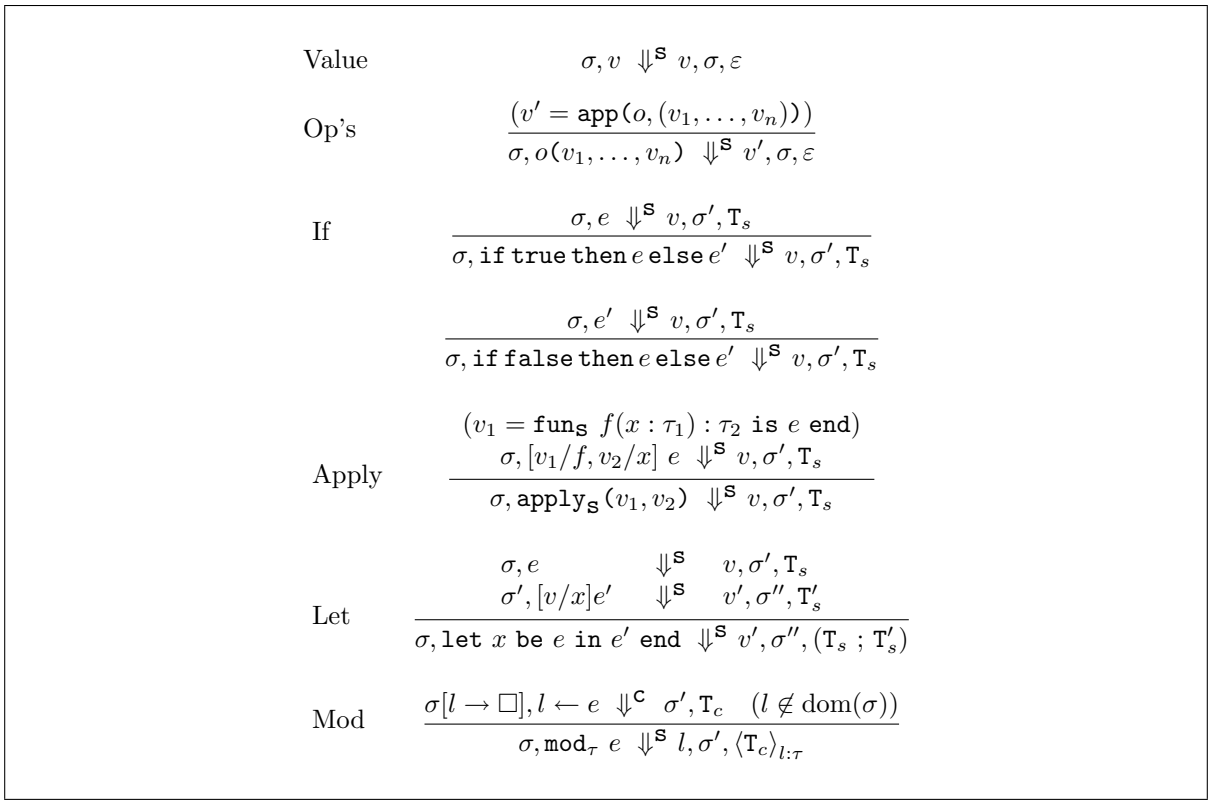

Figure 15: Evaluation of stable expressions.

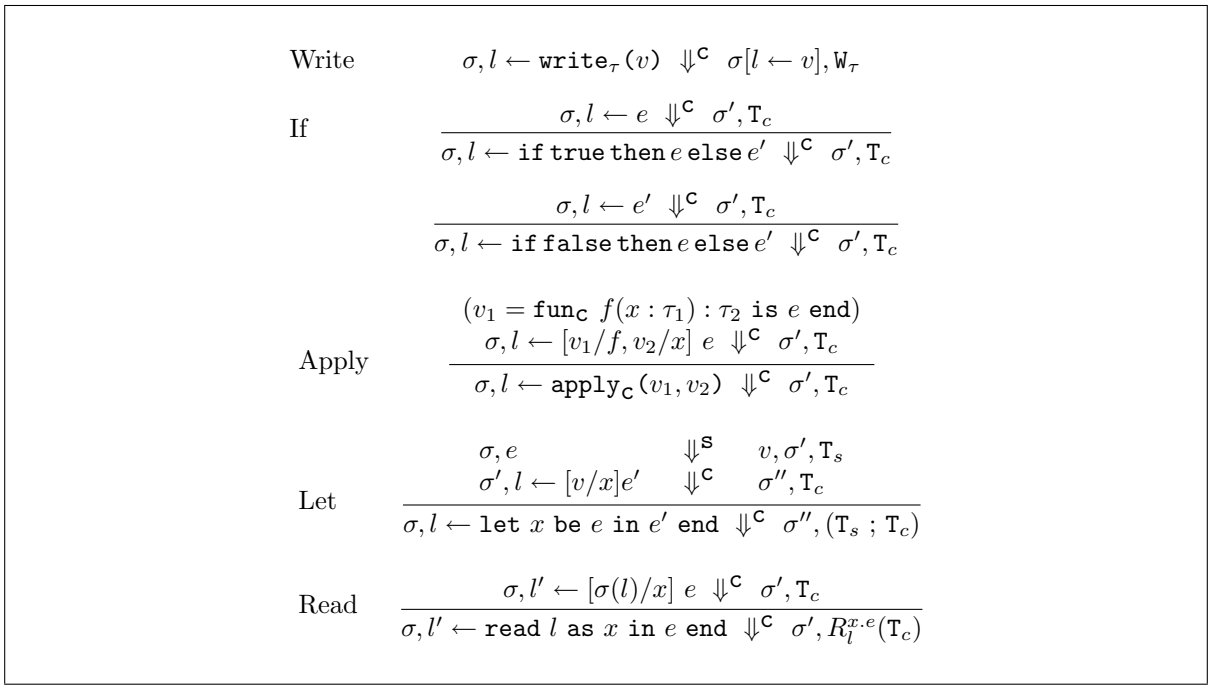

Figure 16: Evaluation of changeable expressions.

Each location allocated during the evaluation a changeable expression is recorded in the trace and is written; the destination is also written: If $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}_c$, then $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma) \cup \mathrm{def}(\mathtt{T}_c)$, and $\mathrm{def}(\sigma') = \mathrm{def}(\sigma) \cup \mathrm{def}(\mathtt{T}_c) \cup \{l\}$. Furthermore, all locations read during evaluation are defined in the store, $\mathrm{dom}(\mathtt{T}_c) \subseteq \mathrm{def}(\sigma')$.

6 Type Safety of AFL

The static semantics of AFL ensures these five properties of its dynamic semantics: (1) each modifiable is written exactly once; (2) no modifiable is read before it is written; (3) dependencies are not lost, i.e., if a value depends on a modifiable, then its value is also placed in a modifiable; (4) the store is acyclic and (5) the data dependences (dynamic dependence graph) is acyclic. These properties are critical for correctness of the adaptivity mechanisms. The last two properties show that AFL is consistent with pure functional programming by ensuring that no cycles arise during evaluation.

The write-once property (1) and no-lost-dependencies property (3) are relatively easy to observe. A write can only take place in the changeable mode and can write to the current destination. Since being the changeable mode requires the creation of a new destination by the `mod` construct, and only the current destination can be written, each modifiable is written exactly once. For property 3, note that dependencies are created by read operations, which take place in the changeable mode, are recorded in the trace, and end with a write. Thus, dependences are recorded and the result of a read is always written to a destination. The proof that the store is acyclic is more involved. We order locations (modifiables) of the store with respect to the times that they are written and require that the value of each expression typecheck with respect to the locations written before that expression. The total order directly implies that the store is acyclic (property 4), i.e., no two locations refer to each other. The restriction that an expression typechecks with respect to the previously written locations ensures that no location is read before it is written (property 2). This fact along with the total ordering on locations implies that there are no cyclic dependences, i.e., the dynamic dependence graph is acyclic (property 5).

The proof of type safety for AFL hinges on a type preservation theorem for the dynamic semantics. Since the dynamic semantics of AFL is given by an evaluation relation, rather than a transition system, the proof of type safety is indirect. First, we prove the type preservation theorem stating that the outcome of evaluation is type consistent, provided that the inputs are. Second, we prove a canonical forms lemma characterizing the "shapes" of closed values of each type. Third, we augment the dynamic semantics with rules stating that evaluation "goes wrong" in the case that the principal argument of an elimination form is non-canonical. Finally, we argue that, by the first two results, these rules can never apply to a well-typed program. Since the last two steps are routine, given the first two, we concentrate on preservation and canonical forms.

6.1 Location Typings

For the safety proof we will enrich location typings with a total ordering on locations that respects the order that they are written to. A location typing, Λ, consists of three parts:

1. A finite set, $\mathrm{dom}(\Lambda)$, of locations, called the *domain* of the store typing.
2. A finite function, also written Λ, assigning types to the locations in $\mathrm{dom}(\Lambda)$.
3. A linear ordering $\leq_\Lambda$ of $\mathrm{dom}(\Lambda)$.

The relation $l <_\Lambda l'$ holds if and only if $l \leq_\Lambda l'$ and $l \neq l'$. The restriction, $\leq_\Lambda \restriction L$, of $\leq_\Lambda$ to a subset $L \subseteq \mathrm{dom}(\Lambda)$ is the intersection $\leq_\Lambda \cap (L \times L)$.

As can be expected, stores are extended with respect to the total order: the *ordered extension*, $\Lambda[l'{:}\tau'{<}l]$, of a location typing Λ assigns the type τ' to the location $l' \notin \mathrm{dom}(\Lambda)$ and places l' immediately before $l \in \mathrm{dom}(\Lambda)$ such that

1. $\mathrm{dom}(\Lambda') = \mathrm{dom}(\Lambda) \cup \{\, l' \,\}$;
2. $\Lambda'(l'') = \begin{cases} \tau' & \textit{if } l'' = l' \\ \Lambda(l'') & \textit{otherwise;} \end{cases}$
3. (a) $l' \leq_{\Lambda'} l$;
   (b) if $l'' \leq_\Lambda l$, then $l'' \leq_{\Lambda'} l'$;
   (c) if $l'' \leq_\Lambda l'''$, then $l'' \leq_{\Lambda'} l'''$.

Location typings are partially ordered with respect to a containment relation by defining $\Lambda \sqsubseteq \Lambda'$ if and only if

1. $\mathrm{dom}(\Lambda) \subseteq \mathrm{dom}(\Lambda')$;
2. if $l \in \mathrm{dom}(\Lambda)$, then $\Lambda'(l) = \Lambda(l)$;
3. $\leq_{\Lambda'} \restriction \mathrm{dom}(\Lambda) = \leq_\Lambda$.

Ordered extensions yield bigger stores: if $l \in \mathrm{dom}(\Lambda)$ and $l' \notin \mathrm{dom}(\Lambda)$, then $\Lambda \sqsubseteq \Lambda[l'{:}\tau'{<}l]$.

Forcing an ordering on the locations of a store suffices to show that the store is acyclic. It does not however help ensure that locations are not read before they are written. We therefore restrict an expression to depend only on those locations that have been written. How can we know what locations are written? At any point in evaluation, we call the last allocated but not yet written location the *cursor* and require that an expression depend only on locations prior to the cursor. The cursor will be maintained such that all locations that precede the cursor have been written and those that come after have not. We therefore define the *restriction* $\Lambda \restriction l$, of a location typing Λ to a location $l \in \mathrm{dom}(\Lambda)$ is defined as the location typing Λ' such that

1. $\mathrm{dom}(\Lambda') = \{\, l' \in \mathrm{dom}(\Lambda) \mid l' <_\Lambda l \,\}$;
2. if $l' <_\Lambda l$, then $\Lambda'(l') = \Lambda(l')$;
3. $\leq_{\Lambda'} = \leq_\Lambda \restriction \mathrm{dom}(\Lambda')$.

Note that if $\Lambda \sqsubseteq \Lambda'$ and $l \in \mathrm{dom}(\Lambda)$, then $\Lambda \restriction l \sqsubseteq \Lambda' \restriction l$.

Definition 3 (Store typing)
A store σ may be assigned a location typing Λ, written $\sigma : \Lambda$, if and only if the following two conditions are satisfied.

1. *$dom(\sigma) = dom(\Lambda)$.*
2. *for each $l \in def(\sigma)$, $\Lambda \restriction l \vdash_{\mathtt{S}} \sigma(l) : \Lambda(l)$.*

The location typing, Λ, imposes a linear ordering on the locations in the store, σ, such that the values in σ store have the types assigned to them by Λ, relative to the types of its preceding locations in the ordering.

$$\frac{}{\Lambda, l_0 \vdash_{\mathtt{S}} \varepsilon \rightsquigarrow \Lambda} \qquad \frac{\Lambda, l_0 \vdash_{\mathtt{S}} \mathtt{T}_s \rightsquigarrow \Lambda' \quad \Lambda', l_0 \vdash_{\mathtt{S}} \mathtt{T}'_s \rightsquigarrow \Lambda''}{\Lambda, l_0 \vdash_{\mathtt{S}} \mathtt{T}_s \,;\, \mathtt{T}'_s \rightsquigarrow \Lambda''}$$

$$\frac{\Lambda[l{:}\tau{<}l_0], l \vdash_{\mathtt{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda' \quad (l \notin \mathrm{dom}(\Lambda))}{\Lambda, l_0 \vdash_{\mathtt{S}} \langle \mathtt{T}_c \rangle_{l:\tau} \rightsquigarrow \Lambda'}$$

$$\frac{}{\Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{W}_\tau : \tau \rightsquigarrow \Lambda} \qquad \frac{\Lambda, l_0 \vdash_{\mathtt{S}} \mathtt{T}_s \rightsquigarrow \Lambda' \quad \Lambda', l_0 \vdash_{\mathtt{S}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda''}{\Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{T}_s \,;\, \mathtt{T}_c : \tau \rightsquigarrow \Lambda''}$$

$$\frac{\Lambda \restriction l_0; x{:}\tau \vdash_{\mathtt{C}} e : \tau' \quad \Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{T}_c : \tau' \rightsquigarrow \Lambda' \quad (l <_\Lambda l_0,\ \Lambda(l) = \tau)}{\Lambda, l_0 \vdash_{\mathtt{C}} R_l^{x.e}(\mathtt{T}_c) : \tau' \rightsquigarrow \Lambda'}$$

Figure 17: Typing of Traces.

6.2 Trace Typing

The formulation of the type safety theorem requires a notion of typing for traces. The judgement $\Lambda, l_0 \vdash_{\mathtt{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$ states that the stable trace $\mathtt{T}_s$ is well-formed relative to the input location typing Λ and the cursor $l_0 \in \mathrm{dom}(\Lambda')$. The output location typing Λ' is an extension of Λ with typings for the locations allocated by the trace; these will all be ordered prior to the cursor. When Λ' is not important, we simply write $\Lambda \vdash_{\mathtt{S}} \mathtt{T}_s\,\mathsf{ok}$ to mean that $\Lambda \vdash_{\mathtt{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$ for some Λ'.

Similarly, the judgement $\Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$ states that the changeable trace $\mathtt{T}_c$ is well-formed relative to Λ and $l_0 \in \mathrm{dom}(\Lambda)$. As with stable traces, Λ' is an extension of Λ with the newly-allocated locations of the trace. When Λ' is not important, we write $\Lambda \vdash_{\mathtt{C}} \mathtt{T}_c : \tau$ for $\Lambda \vdash_{\mathtt{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$ for some Λ'.

The rules for deriving these judgements are given in Figure 17. The input location typing specifies the active locations, of which only those prior to the cursor are eligible as subjects of a read; this ensure a location is not read before it is written. The cursor changes when processing an allocation trace to make the allocated location active, but unreadable, thereby ensuring that no location is read before it is allocated. The output location typing determines the ordering of locations allocated by the trace relative to the ordering of the input locations. Specifically, the ordering of the newly allocated locations is determined by the trace, and is such that they are all ordered to occur immediately prior to the cursor. The ordering so determined is essentially the same as that used in the implementation described in Section 4.

The following invariants hold for traces and trace typings.

1. $\forall l.\, l \in \mathrm{def}(\mathtt{T})$, l is written exactly once: l appears once in a write position in $\mathtt{T}$ of the form $\langle \mathtt{T}_c \rangle_{l:\tau}$ for some $\mathtt{T}_c$.
2. If $\Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$, then $\mathrm{dom}(\Lambda') = \mathrm{dom}(\Lambda) \cup \mathrm{def}(\mathtt{T}_c)$ and $\mathrm{dom}(\mathtt{T}_c) \subseteq \mathrm{dom}(\Lambda')$.
3. If $\Lambda, l_0 \vdash_{\mathtt{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$, then $\mathrm{dom}(\Lambda') = \mathrm{dom}(\Lambda) \cup \mathrm{def}(\mathtt{T}_s)$ and $\mathrm{dom}(\mathtt{T}_s) \subseteq \mathrm{dom}(\Lambda')$.

6.3 Type Preservation

For the proof of type safety we shall make use of a few technical lemmas. First, typing is preserved by addition of typings of "irrelevant" locations and variables.

Lemma 4 (Weakening)
If $\Lambda \sqsubseteq \Lambda'$ *and* $\Gamma \subseteq \Gamma'$, *then*

1. *if* $\Lambda;\Gamma \vdash_{\mathtt{S}} e : \tau$, *then* $\Lambda';\Gamma' \vdash_{\mathtt{S}} e : \tau$;
2. *if* $\Lambda;\Gamma \vdash_{\mathtt{C}} e : \tau$, *then* $\Lambda';\Gamma' \vdash_{\mathtt{C}} e : \tau$;
3. *if* $\Lambda \vdash_{\mathtt{S}} T_s\,\mathsf{ok}$, *then* $\Lambda' \vdash_{\mathtt{S}} T_s\,\mathsf{ok}$;
4. *if* $\Lambda \vdash_{\mathtt{C}} T_c : \tau$, *then* $\Lambda' \vdash_{\mathtt{C}} T_c : \tau$.

Second, typing is preserved by substitution of a value for a free variable of the same type as the value.

Lemma 5 (Value Substitution)
Suppose that $\Lambda;\Gamma \vdash_{\mathtt{S}} v : \tau$.

1. *If* $\Lambda;\Gamma, x{:}\tau \vdash_{\mathtt{S}} e' : \tau'$, *then* $\Lambda;\Gamma \vdash_{\mathtt{S}} [v/x]e' : \tau'$.
2. *If* $\Lambda;\Gamma, x{:}\tau \vdash_{\mathtt{C}} e' : \tau'$, *then* $\Lambda;\Gamma \vdash_{\mathtt{C}} [v/x]e' : \tau'$.

The type preservation theorem for AFL states that the result of evaluation of a well-typed expression is itself well-typed. The location l_0, called the cursor, is the current allocation point. All locations prior to the cursor are written to, and location following the cursor are allocated but not yet written. All new locations are allocated prior to l_0 in the ordering and the newly allocated location becomes the cursor. The theorem requires that the input expression be well-typed relative to those locations preceding the cursor so as to preclude forward references to locations that have been allocated, but not yet initialized. In exchange the result is assured to be sensible relative to those locations prior to the cursor, all of which are allocated and initialized. This ensures that no location is read before it has been allocated and initialized.

Theorem 6 (Type Preservation)

1. *If*
   (a) $\sigma, e \;\Downarrow^{\mathtt{S}}\; v, \sigma', T_s$,
   (b) $\sigma : \Lambda$,
   (c) $l_0 \in dom(\Lambda)$,
   (d) $l <_\Lambda l_0$ *implies* $l \in def(\sigma)$,
   (e) $\Lambda \restriction l_0 \vdash_{\mathtt{S}} e : \tau$,

   then there exists $\Lambda' \sqsupseteq \Lambda$ *such that*

   (f) $\Lambda' \restriction l_0 \vdash_{\mathtt{S}} v : \tau$,
   (g) $\sigma' : \Lambda'$, *and*
   (h) $\Lambda, l_0 \vdash_{\mathtt{S}} T_s \rightsquigarrow \Lambda'$.
2. *If*
   (a) $\sigma, l_0 \leftarrow e \;\Downarrow^{\mathtt{C}}\; \sigma', T_c$,
   (b) $\sigma : \Lambda$,
   (c) $\Lambda(l_0) = \tau_0$,

(d) $l <_\Lambda l_0$ *implies* $l \in \mathit{def}(\sigma)$,

(e) $\Lambda \restriction l_0 \vdash_{\mathtt{C}} e : \tau_0$,

then

(f) $l_0 \in \mathit{def}(\sigma')$,

and there exists $\Lambda' \sqsupseteq \Lambda$ *such that*

(g) $\sigma' : \Lambda'$, *and*

(h) $\Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{T}_c : \tau_0 \rightsquigarrow \Lambda'$.

Proof: Simultaneously, by induction on evaluation. We will consider several illustrative cases.

- Suppose that

(1a) $\sigma, \mathtt{mod}_\tau\ e \ \Downarrow^{\mathtt{S}}\ l, \sigma'', \langle \mathtt{T}_c \rangle_{l:\tau}$;

(1b) $\sigma : \Lambda$;

(1c) $l_0 \in \mathrm{dom}(\Lambda)$;

(1d) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$;

(1e) $\Lambda \restriction l_0 \vdash_{\mathtt{S}} \mathtt{mod}_\tau\ e : \tau\ \mathtt{mod}$.

Since the typing and evaluation rules are syntax-directed, it follows that

(1a(i)) $\sigma[l \rightarrow \Box], l \leftarrow e \ \Downarrow^{\mathtt{C}}\ \sigma'', \mathtt{T}_c$, where $l \notin \mathrm{dom}(\sigma)$, and

(1e(i)) $\Lambda \restriction l_0 \vdash_{\mathtt{C}} e : \tau$.

Note that $l \notin \mathrm{dom}(\Lambda)$, by (1b). Let $\sigma' = \sigma[l \rightarrow \Box]$ and let $\Lambda' = \Lambda[l{:}\tau{<}l_0]$. Note that $\Lambda \sqsubseteq \Lambda'$ and that $\Lambda'(l) = \tau$.

Then we have

(2a′) $\sigma', l \leftarrow e \ \Downarrow^{\mathtt{C}}\ \sigma'', \mathtt{T}_c$, by (1a(i));

(2b′) $\sigma' : \Lambda'$. Since $\sigma : \Lambda$ by (1b), we have $\Lambda \restriction l' \vdash_{\mathtt{S}} \sigma(l') : \Lambda(l')$ for every $l' \in \mathrm{def}(\sigma)$. Now $\mathrm{def}(\sigma') = \mathrm{def}(\sigma)$, so for every $l' \in \mathrm{def}(\sigma')$, $\sigma'(l') = \sigma(l')$ and $\Lambda'(l') = \Lambda(l')$. Therefore, by Lemma 4, we have $\Lambda' \restriction l' \vdash_{\mathtt{S}} \sigma'(l') : \Lambda'(l')$, as required.

(2c′) $\Lambda'(l) = \tau$, by definition;

(2d′) $l' <_{\Lambda'} l$ implies $l' \in \mathrm{def}(\sigma')$, since $l' <_{\Lambda'} l$ implies $l' <_\Lambda l_0$ and (1d);

(2e′) $\Lambda' \restriction l \vdash_{\mathtt{C}} e : \tau$ since $\Lambda' \restriction l = \Lambda \restriction l_0$ and (1e(i)).

Therefore, by induction,

(2f′) $l \in \mathrm{def}(\sigma'')$;

and there exists $\Lambda'' \sqsupseteq \Lambda'$ such that

(2g′) $\sigma'' : \Lambda''$;

(2h′) $\Lambda', l \vdash_{\mathtt{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda''$.

Hence we have

(1f) $\Lambda'' \restriction l_0 \vdash_{\mathtt{S}} l : \tau$, since $(\Lambda'' \restriction l_0)(l) = \Lambda'(l) = \tau$;

(1g) $\sigma'' : \Lambda''$ by (2g′);

(1h) $\Lambda, l_0 \vdash_{\mathtt{S}} \langle \mathtt{T}_c \rangle_{l:\tau} : \tau \rightsquigarrow \Lambda''$, by (2h′).

- Suppose that

(2a) $\sigma, l_0 \leftarrow \mathtt{write}_\tau(v) \Downarrow^{\mathtt{C}} \sigma[l_0 \leftarrow v], \mathtt{W}_\tau$;

(2b) $\sigma : \Lambda$;

(2c) $\Lambda(l_0) = \tau$;

(2d) $l <_\Lambda l_0$ implies $l \in \mathrm{def}(\sigma)$;

(2e) $\Lambda \restriction l_0 \vdash_{\mathtt{C}} \mathtt{write}_\tau(v) : \tau$.

By the syntax-directed nature of the typing rules it follows that

(2e(i)) $\Lambda \restriction l_0 \vdash_{\mathtt{S}} v : \tau$.

Let $\Lambda' = \Lambda$ and $\sigma' = \sigma[l_0 \leftarrow v]$. Then we have:

(2f) $l_0 \in \mathrm{def}(\sigma')$, by definition of σ';

(2g) $\sigma' : \Lambda'$, since $\sigma : \Lambda$ by (2b), $\Lambda \restriction l_0 \vdash_{\mathtt{S}} v : \tau$ by (2e(i)), and $\Lambda(l_0) = \tau$ by (2c).

(2h) $\Lambda', l_0 \vdash_{\mathtt{C}} \mathtt{W}_\tau : \tau \rightsquigarrow \Lambda'$ by definition.

- Suppose that

(2a) $\sigma, l_0 \leftarrow \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end} \Downarrow^{\mathtt{C}} \sigma', R_l^{x.e}(\mathtt{T}_c)$;

(2b) $\sigma : \Lambda$;

(2c) $\Lambda(l_0) = \tau_0$;

(2d) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$;

(2e) $\Lambda \restriction l_0 \vdash_{\mathtt{C}} \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end} : \tau_0$.

By the syntax-directed nature of the evaluation and typing rules, it follows that

(2a(i)) $\sigma, l_0 \leftarrow [\sigma(l)/x]\ e \Downarrow^{\mathtt{C}} \sigma', \mathtt{T}_c$;

(2e(i)) $\Lambda \restriction l_0 \vdash_{\mathtt{S}} l : \tau\ \mathtt{mod}$, hence $(\Lambda \restriction l_0)(l) = \Lambda(l) = \tau$, and so $l <_\Lambda l_0$ and $\Lambda(l) = \tau$;

(2e(ii)) $\Lambda \restriction l_0; x{:}\tau \vdash_{\mathtt{C}} e : \tau_0$.

Since $l <_\Lambda l_0$, it follows that $\Lambda \restriction l \sqsubseteq \Lambda \restriction l_0$.

Therefore,

(2a′) $\sigma, l_0 \leftarrow [\sigma(l)/x]\ e \Downarrow^{\mathtt{C}} \sigma', \mathtt{T}_c$ by (2a(i));

(2b′) $\sigma : \Lambda$ by (2b);

(2c′) $\Lambda(l_0) = \tau_0$ by (2c);

(2d′) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$ by (2d).

Furthermore, by (2b′), we have $\Lambda \restriction l \vdash_{\mathtt{S}} \sigma(l) : \Lambda(l)$, hence $\Lambda \restriction l_0 \vdash_{\mathtt{S}} \sigma(l) : \Lambda(l)$ and so by Lemma 5 and 2e(ii),

(2e′) $\Lambda \restriction l_0 \vdash_{\mathtt{C}} [\sigma(l)/x]e : \tau_0$.

It follows by induction that

(2f′) $l_0 \in \mathrm{def}(\sigma')$

and there exists $\Lambda' \sqsupseteq \Lambda$ such that

(2g′) $\sigma' : \Lambda'$;

(2h′) $\Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$.

Therefore we have

(2f) $l_0 \in \mathrm{def}(\sigma')$ by (2f′);

(2g) $\sigma' : \Lambda'$ by (2g′);

(2h) $\Lambda, l_0 \vdash_{\mathtt{C}} R_l^{x.e}(\mathtt{T}_c) : \tau_0 \rightsquigarrow \Lambda'$, since

(a) $\Lambda \restriction l_0, x{:}\Lambda(l) \vdash_{\mathtt{C}} e : \tau_0$ by (2e(ii));

(b) $\Lambda, l_0 \vdash_{\mathtt{C}} \mathtt{T}_c : \tau_0 \rightsquigarrow \Lambda'$ by (2h′);

(c) $l \leq_\Lambda l_0$ by (2e(i)).

■

6.4 Type Safety for AFL

Type safety follows from the canonical forms lemma, which characterizes the shapes of closed values of each type.

Lemma 7 (Canonical Forms)
Suppose that $\Lambda \vdash_{\mathtt{S}} v : \tau$. Then

- *If $\tau =$ `int`, then v is a numeric constant.*
- *If $\tau =$ `bool`, then v is either `true` or `false`.*
- *If $\tau = \tau_1 \xrightarrow{\mathtt{C}} \tau_2$, then $v =$ `fun`$_{\mathtt{C}}$ $f(x : \tau_1) : \tau_2$ `is` e `end` with $\Lambda; f{:}\tau_1 \xrightarrow{\mathtt{C}} \tau_2, x{:}\tau_1 \vdash_{\mathtt{C}} e : \tau_2$.*
- *If $\tau = \tau_1 \xrightarrow{\mathtt{S}} \tau_2$, then $v =$ `fun`$_{\mathtt{S}}$ $f(x : \tau_1) : \tau_2$ `is` e `end` with $\Lambda; f{:}\tau_1 \xrightarrow{\mathtt{S}} \tau_2, x{:}\tau_1 \vdash_{\mathtt{S}} e : \tau_2$.*
- *If $\tau = \tau'$ `mod`, then $v = l$ for some $l \in dom(\Lambda)$ such that $\Lambda(l) = \tau'$.*

Theorem 8 (Type Safety)
Well-typed programs do not "go wrong".

Proof (sketch): Instrument the dynamic semantics with rules that "go wrong" in the case of a non-canonical principal argument to an elimination form. Then show that no such rule applies to a well-typed program, by appeal to type preservation and the canonical forms lemma. ■

7 Change Propagation

We formalize the change-propagation algorithm and the notion of an input change and prove the type safety and correctness of the change-propagation algorithm.

We represent input changes via difference stores. A *difference store* is a finite map assigning values to locations. Unlike a store, a difference store may contain "dangling" locations that are not defined within the difference store. The process of modifying a store with a difference store is defined as follows.

Definition 9 (Store modification)
Let $\sigma : \Lambda$ be a well-typed store for some Λ and let δ be a difference store. The modification of σ by δ, written $\sigma \oplus \delta$, is a well-typed store $\sigma' : \Lambda'$ for some $\Lambda' \sqsupseteq \Lambda$ and such that

$$\sigma' = \sigma \oplus \delta = \delta \cup \{\, (l, \sigma(l)) \mid l \in dom(\sigma) \wedge l \notin dom(\delta) \,\}.$$

Note that the definition requires the result store be well typed and the types of modified locations be preserved.

Modifying a store σ with a difference store δ changes the (values of the) locations in the set $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\delta)$. This set consists of *changed* locations and is called the *changed-set*. Throughout, we use χ and its variants to denote changed-sets.

Figure 18 gives the semantics of the change-propagation algorithm that was previously described in an algorithmic setting (Section 4). In the rest of this section, the term "change-propagation algorithm" refers to the semantics. The change-propagation algorithm takes a modified store, a trace obtained by evaluating an AFL program with respect to the original store, and a changed-set. The algorithm scans the trace as it seeks for reads of changed locations. When such a read is found, the body of the read is re-evaluated with the new value to update the trace and the store. Since re-evaluation can change the target of the re-evaluated read, the target is added to the changed-set.

The change-propagation algorithm is given by these two judgments:

1. *Stable propagation*: $\sigma, \mathtt{T}_s, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{s}}\; \sigma', \mathtt{T}'_s, \chi'$;
2. *Changeable propagation*: $\sigma, l \leftarrow \mathtt{T}_c, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma', \mathtt{T}'_c, \chi'$;

The judgments define the change propagation for a stable trace $\mathtt{T}_s$ and a changeable trace $\mathtt{T}_c$, with respect to a store σ, and a changed-set $\chi \subseteq \mathrm{dom}(\sigma)$. For changeable propagation a target location, l, is maintained as in the changeable evaluation mode of AFL.

The rules for change propagation are given in Figure 18. Given a trace, change propagation mimics the evaluation rule of AFL 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.

The most interesting rule is the `read` rule. This rule mimics a `read` operation, which evaluates an expression after binding its specified variable to the value of the location read. The read rule takes different actions depending on whether the location being read, i.e., the source, is in the changed-set or not. If source is not in the changed-set, then the read need not be re-evaluated and change propagation continues to scan the rest of the trace. If source is in the changed-set, then the body of the read is re-evaluated after substituting the new value of the sources for the bound variable. Re-evaluation yields a revised store and a new trace. The new trace is obtained by replacing the trace for the re-evaluated read ($\mathtt{T}_c$) with the trace returned by the re-evaluation

$$\sigma, \varepsilon, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{s}}\; \sigma, \varepsilon, \chi$$

$$\text{Mod} \qquad \frac{\sigma, l \leftarrow \mathtt{T}_c, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma', \mathtt{T}'_c, \chi'}{\sigma, \langle \mathtt{T}_c \rangle_{l:\tau}, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{s}}\; \sigma', \langle \mathtt{T}'_c \rangle_{l:\tau}, \chi'}$$

$$\text{Let} \qquad \frac{\begin{array}{ccc}\sigma, \mathtt{T}_s, \chi & \Downarrow^{\mathtt{p}}_{\mathtt{s}} & \sigma', \mathtt{T}''_s, \chi' \\ \sigma', \mathtt{T}'_s, \chi' & \Downarrow^{\mathtt{p}}_{\mathtt{s}} & \sigma'', \mathtt{T}'''_s, \chi''\end{array}}{\sigma, (\mathtt{T}_s \;;\; \mathtt{T}'_s), \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{s}}\; \sigma'', (\mathtt{T}''_s \;;\; \mathtt{T}'''_s), \chi''}$$

$$\text{Write} \qquad \sigma, l \leftarrow \mathtt{W}_\tau, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{s}}\; \sigma, \mathtt{W}_\tau, \chi$$

$$\text{Read} \qquad \frac{\sigma, l' \leftarrow \mathtt{T}_c, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma', \mathtt{T}'_c, \chi'}{\sigma, l' \leftarrow R^{x.e}_l(\mathtt{T}_c), \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma', R^{x.e}_l(\mathtt{T}'_c), \chi'} \; (l \notin \chi)$$

$$\frac{\sigma, l' \leftarrow [\sigma(l)/x]e \;\Downarrow^{\mathtt{c}}\; \sigma', \mathtt{T}'_c}{\sigma, l' \leftarrow R^{x.e}_l(\mathtt{T}_c), \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma', R^{x.e}_l(\mathtt{T}'_c), \chi \cup \{l'\}} \; (l \in \chi)$$

$$\text{Let} \qquad \frac{\begin{array}{ccc}\sigma, \mathtt{T}_s, \chi & \Downarrow^{\mathtt{p}}_{\mathtt{s}} & \sigma', \mathtt{T}'_s, \chi' \\ \sigma', l' \leftarrow \mathtt{T}_c, \chi' & \Downarrow^{\mathtt{p}}_{\mathtt{c}} & \sigma'', \mathtt{T}'_c, \chi''\end{array}}{\sigma, l' \leftarrow (\mathtt{T}_s \;;\; \mathtt{T}_c), \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma'', (\mathtt{T}'_s \;;\; \mathtt{T}'_c), \chi''}$$

Figure 18: Change propagation rules (stable and changeable).

$(\mathtt{T}'_c)$. Since re-evaluating the read may change the value written to the target, the target location is added to the changed-set.

The purely functional change-propagation algorithm presented here scans the whole trace. A direct implementation of this algorithm will therefore run in time linear in the size of the trace. Note, however, that the change-propagation algorithm revises the trace by replacing the traces of re-evaluated reads. Thus, if one is content with updating the trace with side effects, then traces of re-evaluated reads can be replaced in place, while skipping over the unchanged parts of the trace. This is the main idea behind the dynamic dependence graphs (Section 4.4). The ML implementation performs change propagation using dynamic dependence graphs (Section 4.9).

7.1 Type Safety

The change-propagation algorithm also enjoys a type preservation property stating that if the initial state is well-formed, so is the result state. This ensures that the results of change propagation can subsequently be used as further inputs. For the preservation theorem to apply, the store modification must respect the typing of the store being modified.

Theorem 10 (Type Preservation)
Suppose that def(σ) = dom(σ).

1. *If*

   (a) $\sigma, \mathtt{T}_s, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{s}}\; \sigma', \mathtt{T}'_s, \chi'$,
   (b) $\sigma : \Lambda$,
   (c) $l_0 \in dom(\Lambda)$,
   (d) $\Lambda, l_0 \vdash_{\mathtt{s}} \mathtt{T}_s\,\mathsf{ok}$, *and*
   (e) $\chi \subseteq dom(\Lambda)$,

   then for some $\Lambda' \sqsupseteq \Lambda$,

   (f) $\sigma' : \Lambda'$,
   (g) $\Lambda, l_0 \vdash_{\mathtt{s}} \mathtt{T}'_s \rightsquigarrow \Lambda'$,
   (h) $\chi' \subseteq dom(\Lambda)$.

2. *If*

   (a) $\sigma, l_0 \leftarrow \mathtt{T}_c, \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma', \mathtt{T}'_c, \chi'$,
   (b) $\sigma : \Lambda$,
   (c) $\Lambda(l_0) = \tau_0$,
   (d) $\Lambda, l_0 \vdash_{\mathtt{c}} \mathtt{T}_c : \tau_0$, *and*
   (e) $\chi \subseteq dom(\Lambda)$,

   then there exists $\Lambda' \sqsupseteq \Lambda$ *such that*

   (f) $\sigma' : \Lambda'$,
   (g) $\Lambda, l_0 \vdash_{\mathtt{c}} \mathtt{T}'_c : \tau_0 \rightsquigarrow \Lambda'$, *and*
   (h) $\chi' \subseteq dom(\Lambda)$.

Proof: By induction on the definition of the change propagation relations, making use of Theorem 6. We consider the case of a re-evaluation of a read. Suppose that $l \in \chi$ and

(2a) $\sigma, l_0 \leftarrow R_l^{x.e}(\mathtt{T}_c), \chi \;\Downarrow^{\mathtt{p}}_{\mathtt{c}}\; \sigma', R_l^{x.e}(\mathtt{T}'_c), \chi \cup \{l_0\}$;

(2b) $\sigma : \Lambda$;

(2c) $\Lambda(l_0) = \tau_0$;

(2d) $\Lambda, l_0 \vdash_{\mathtt{c}} R_l^{x.e}(\mathtt{T}_c) : \tau_0$;

(2e) $\chi \subseteq \mathrm{dom}(\Lambda)$.

By the syntax-directed nature of the change propagation and trace typing rules, it follows that

(2a(i)) $\sigma, l_0 \leftarrow [\sigma(l)/x]e \;\Downarrow^{\mathtt{c}}\; \sigma', \mathtt{T}'_c$;

(2b(i) $\Lambda \restriction l_0 \vdash_{\mathtt{s}} \sigma(l) : \Lambda(l)$, by (2b);

(2d(i)) $l <_\Lambda l_0$ and $\Lambda(l) = \tau$ for some type τ;

(2d(ii)) $\Lambda \restriction l_0; x{:}\tau \vdash_{\mathsf{C}} e : \tau_0$;

(2d(iii)) $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau_0$.

Therefore

(2a′) $\sigma, l_0 \leftarrow [\sigma(l)/x]e \ \Downarrow^{\mathsf{C}} \ \sigma', \mathtt{T}'_c$ by (2a(i));

(2b′) $\sigma : \Lambda$ by (2b);

(2c′) $\Lambda(l_0) = \tau_0$ by (2c);

(2d′) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$ by assumption that $\mathrm{def}(\sigma) = \mathrm{dom}(\sigma)$;

(2e′) $\Lambda \restriction l_0 \vdash_{\mathsf{C}} [\sigma(l)/x]e : \tau_0$ by (2d(ii)), (2b(i)), and Lemma 5.

Hence, by Theorem 6,

(2f′) $l \in \mathrm{def}(\sigma')$;

and there exists $\Lambda' \sqsupseteq \Lambda$ such that

(2g′) $\sigma' : \Lambda'$;

(2h′) $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}'_c : \tau_0 \rightsquigarrow \Lambda'$.

Consequently,

(2f) $\sigma' : \Lambda'$ by (2g′);

(2g) $\Lambda, l_0 \vdash_{\mathsf{C}} R_l^{x.e}(\mathtt{T}'_c) : \tau_0$ by (2d(i) and (ii)), (2h′), and Lemma 4;

(2h) $C \cup \{ l_0 \} \subseteq \mathrm{dom}(\Lambda')$ since $l_0 \in \mathrm{dom}(\Lambda)$ and $\Lambda' \sqsupseteq \Lambda$.

■

7.2 Correctness

Change propagation simulates a complete re-evaluation by re-evaluating only the affected sub-expressions of an AFL program. This sections shows that change propagation is correct by proving that it yields the same output and the trace as a complete re-evaluation.

Consider evaluating an adaptive program (a stable expression) e with respect to an initial store σ_i; call this the *initial evaluation*. As shown in Figure 19, the initial evaluation yields a value v_i, an extended store σ'_i, and a trace $\mathtt{T}^i_s$. Now modify the initial store with a difference store δ as $\sigma_s = \sigma_i \oplus \delta$ and re-evaluate the program with this store in a *subsequent evaluation*. To simulate the subsequent evaluation via a change propagation apply the modifications δ to σ'_i and let σ_m denote the modified store, i.e., $\sigma_m = \sigma'_i \oplus \delta$. Now perform change propagation with respect to σ_m, the trace of the initial evaluation $\mathtt{T}^i_s$, and the set of changed locations $\mathrm{dom}(\sigma'_i) \cap \mathrm{dom}(\delta)$. Change propagation yields a revised store σ'_m and trace $\mathtt{T}^m_s$. For the change-propagation to work properly, δ must change only input locations, i.e., $\mathrm{dom}(\sigma'_i) \cap \mathrm{dom}(\delta) \subseteq \mathrm{dom}(\sigma_i)$.

To prove correctness, we compare the store and the trace obtained by the subsequent evaluation, σ'_s and $\mathtt{T}^s_s$, to those obtained by the change propagation, σ'_m and $\mathtt{T}^m_s$, (see Figure 19). Since locations (names) are chosen arbitrarily during evaluation, subsequent evaluation and change propagation

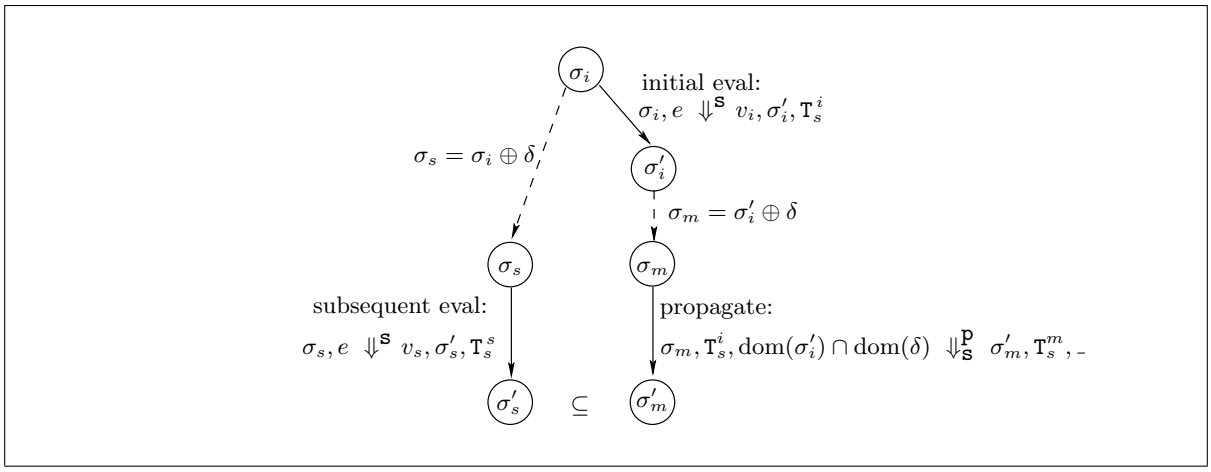

Figure 19: Change propagation simulates a complete re-evaluation.

can choose different locations (names). We therefore show that the traces are identical modulo the choice of locations and the the store σ'_s is contained in the store σ'_m modulo the choice of locations.

To study relations between traces and stores modulo the choice of locations we use an equivalence relation for stores and traces that matches different locations via a partial bijection. A *partial bijection* is a one-to-one mapping from a set of locations D to a set of locations R that may not map all the locations in D.

Definition 11 (Partial bijection)
B is a partial bijection from set D to set R if it satisfies the following:

1. $B \subseteq \{\, (a, b) \mid a \in D, b \in R \,\}$,
2. *if* $(a, b) \in B$ *and* $(a, b') \in B$ *then* $b = b'$,
3. *if* $(a, b) \in B$ *and* $(a', b) \in B$ *then* $a = a'$.

The value of a location l under the partial bijection B is denoted by $B(l)$.

A partial bijection, B, can be applied to an expression e, to a store σ, or to a trace $\mathtt{T}$, denoted $B[e]$, $B[\sigma]$, and $B[\mathtt{T}]$, by replacing, whenever defined, each location l with $B(l)$:

Definition 12 (Applications of partial bijections)

Expression: *The application of a partial bijection B to an expression e yields another expression obtained by substituting each location l in e with $B(l)$ (when defined) as shown in Figure 20.*

Hole: *The application of a partial bijection to a hole yields a hole, $B[\Box] = \Box$.*

Store: *The application of a partial bijection B to a store σ yields another store $B[\sigma]$, defined as $B[\sigma] = \{\, (B[l], B[\sigma(l)]) \mid l \in dom(\sigma) \,\}$.*

$$
\begin{array}{rcl}
B[c] = c, \quad B[x] = x & & \\
B[l] & = & \begin{cases} l' & if\ (l,l') \in B \\ l & otherwise \end{cases} \\
B[\texttt{fun}_\texttt{S}\ f(x:\tau):\tau'\ \texttt{is}\ e\ \texttt{end}] & = & \texttt{fun}_\texttt{S}\ f(x:\tau):\tau'\ \texttt{is}\ B[e_s]\ \texttt{end} \\
B[\texttt{fun}_\texttt{C}\ f(x:\tau):\tau'\ \texttt{is}\ e\ \texttt{end}] & = & \texttt{fun}_\texttt{C}\ f(x:\tau):\tau'\ \texttt{is}\ B[e_c]\ \texttt{end}
\end{array}
$$

$$
\begin{array}{rcl}
B[o(v_1,\ldots,v_n)] & = & o(B[v_1],\ldots,B[v_n]) \\
B[\texttt{apply}_\texttt{S}(v_1,v_2)] & = & \texttt{apply}_\texttt{S}(B[v_1],B[v_2]) \\
B[\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e'_s\ \texttt{end}] & = & \texttt{let}\ x\ \texttt{be}\ B[e_s]\ \texttt{in}\ B[e'_s]\ \texttt{end} \\
B[\texttt{if}\ v\ \texttt{then}\ e_s\ \texttt{else}\ e'_s] & = & \texttt{if}\ B[v]\ \texttt{then}\ B[e_s]\ \texttt{else}\ B[e'_s] \\
B[\texttt{mod}_\tau\ e_c] & = & \texttt{mod}_\tau\ B[e_c]
\end{array}
$$

$$
\begin{array}{rcl}
B[\texttt{apply}_\texttt{C}(v_1,v_2)] & = & \texttt{apply}_\texttt{C}(B[v_1],B[v_2]) \\
B[\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_c\ \texttt{end}] & = & \texttt{let}\ x\ \texttt{be}\ B[e_s]\ \texttt{in}\ B[e_c]\ \texttt{end} \\
B[\texttt{if}\ v\ \texttt{then}\ e_c\ \texttt{else}\ e'_c] & = & \texttt{if}\ B[v]\ \texttt{then}\ B[e_c]\ \texttt{else}\ B[e'_c] \\
B[\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c\ \texttt{end}] & = & \texttt{read}\ B[v]\ \texttt{as}\ x\ \texttt{in}\ B[e_c]\ \texttt{end} \\
B[\texttt{write}_\tau(v)] & = & \texttt{write}_\tau(B[v])
\end{array}
$$

Figure 20: Application of a partial bijection B to values, and stable and changeable expression.

Trace: *The application of a partial bijection to a trace is defined as*

$$
\begin{array}{rcl}
B[\epsilon] & = & \epsilon \\
B[\langle \mathtt{T}_c \rangle_{l:\tau}] & = & \langle B[\mathtt{T}_c] \rangle_{B[l]:\tau} \\
B[\mathtt{T}_s\ ;\ \mathtt{T}_s] & = & B[\mathtt{T}_s]\ ;\ B[\mathtt{T}_s] \\
B[\mathtt{W}_\tau] & = & \mathtt{W}_\tau \\
B[R_l^{x.e}(\mathtt{T}_c)] & = & R_{B[l]}^{B[x].B[e]}(B[\mathtt{T}_c]) \\
B[\mathtt{T}_s\ ;\ \mathtt{T}_c] & = & B[\mathtt{T}_s]\ ;\ B[\mathtt{T}_c].
\end{array}
$$

Destination: *Application of a partial bijection to an expression with a destination is defined as* $B[l \leftarrow e] = B[l] \leftarrow B[e]$.

The correctness theorem shows that the traces obtained by change propagation and the subsequent evaluation are identical under some partial bijection B, i.e., $B[\mathtt{T}_s^m] = \mathtt{T}_s^s$ (referring back to Figure 19). The relationship between the store σ'_s of the subsequent evaluation and σ'_m of change propagation is more subtle. Since the change propagation is performed on the store that the initial evaluation yields σ'_i, and no allocated location is ever deleted, the store after the change propagation will contain leftover unused (garbage) locations from the initial evaluation. We will therefore show that $B[\sigma'_m]$ contains σ'_s.

Definition 13 (Store containment)
We say that a store, σ, is contained *in another σ', written $\sigma \sqsubseteq \sigma'$, if*

1. $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$, and
2. $\forall l, l \in \mathrm{def}(\sigma), \sigma(l) = \sigma'(l)$.

We now state and prove the correctness theorem. The correctness theorem concerns *equal* programs or stable expressions, i.e., expression that are syntactically identical. The theorem hinges on a lemma (Lemma 16) that concerns expressions that are equal up to a partial bijection (such expressions arise due to substitution of a variable by two different locations).

Theorem 14 (Correctness)
Let e be a well-typed program with respect to a store typing Λ, $\sigma_i : \Lambda$ be an initial store such that $def(\sigma_i) = dom(\sigma_i)$, δ be a difference store, $\sigma_s = \sigma_i \oplus \delta$, and $\sigma_m = \sigma'_i \oplus \delta$ as shown in Figure 19. If

(A1) $\sigma_i, e \Downarrow^{\mathtt{S}} v_i, \sigma'_i, \mathtt{T}^i_s$, *(initial evaluation)*

(A2) $\sigma_s, e \Downarrow^{\mathtt{S}} v_s, \sigma'_s, \mathtt{T}^s_s$, *(subsequent evaluation)*

(A3) $dom(\sigma'_i) \cap dom(\delta) \subseteq dom(\sigma_i)$

then the following holds:

1. $\sigma_m, \mathtt{T}^i_s, (dom(\sigma'_i) \cap dom(\delta)) \Downarrow^{\mathtt{P}}_{\mathtt{S}} \sigma'_m, \mathtt{T}^m_s, \_$,
2. *there is a partial bijection $\mathcal{B}$ such that*
   (a) $\mathcal{B}[v_i] = v_s$,
   (b) $\mathcal{B}[\mathtt{T}^m_s] = \mathtt{T}^s_s$,
   (c) $\mathcal{B}[\sigma'_m] \sqsupseteq \sigma'_s$.

Proof: The proof is by an application of Lemma 16. To apply the lemma, define the partial bijection B (of Lemma 16) to be the identity function with domain $\mathrm{dom}(\sigma_i)$. The following hold:

1. $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma'_i)$ (follows by the definition of the store modification).
2. $\forall l, l \in (\mathrm{def}(\sigma'_i) - \mathrm{def}(\sigma_i)), \sigma_m(l) = \sigma'_i(l)$ (follows by assumption three (A3) and the definition of the store modification).
3. $B[\sigma_m] \sqsupseteq \sigma_s$ (since B is the identity, this reduces to $\sigma_m \sqsupseteq \sigma_s$, which holds because $\sigma_s = \sigma_i \oplus \delta$, and $\sigma_m = \sigma'_i \oplus \delta$ and $\sigma'_i \sqsupseteq \sigma_i$).

Applying Lemma 16 with changed locations $\mathrm{dom}(\sigma'_i) \cap \mathrm{dom}(\delta) = \{l \mid l \in \mathrm{dom}(\sigma_i) \wedge \sigma_i[l] \neq \sigma_s[l]\}$ yields a partial bijection B' such that

1. $B'[v_i] = v_s$,
2. $B'[\mathtt{T}^m_s] = \mathtt{T}^s_s$,
3. $B'[\sigma'_m] \sqsupseteq \sigma'_s$.

Thus taking $\mathcal{B} = B'$ proves the theorem.

■

We turn our attention to the main lemma. Lemma 16 considers initial and subsequent evaluations of expressions that are equivalent under some partial bijection and shows that the subsequent evaluation is identical to change propagation under an extended partial bijection. The need to consider expressions that are equivalent under some partial bijection arises because arbitrarily chosen locations (names) can be substituted for the same variable in two different evaluations. We start by defining the notion of a changed-set, the set of changed locations of a store, with respect to some modified store and a partial bijection.

Definition 15 (Changed-set)
Given two stores σ and σ', and a partial bijection B from $dom(\sigma)$ to the $dom(\sigma')$ the changed-set *of σ is*

$$\chi(B, \sigma, \sigma') = \{\, l \mid l \in dom(\sigma), B[\sigma(l)] \neq \sigma'(B[l]) \,\}.$$

The main lemma consists of two symmetric parts for stable and changeable expressions. For each kind of expression, it shows that the trace and the store of a subsequent evaluation under some partial bijection is identical to those that would be obtained by change propagation under some extended partial bijection. The lemma constructs a partial bijection by mapping locations created by change propagation to those created by the subsequent evaluation. We will assume that the expression are well-typed with respect to the stores that they are evaluated with—indeed, the correctness theorem (Theorem 14) requires that the expressions and store modifications be well typed.

The proof assumes that a changeable expression evaluates to a different value when re-evaluated, i.e., the value of the target. This assumption causes no loss of generality, and can be eliminated by additional machinery for comparing of the old and the new values of the target.

Lemma 16 (Change Propagation)
Consider the stores σ_i and σ_s and B be a partial bijection from $dom(\sigma_i)$ to $dom(\sigma_s)$. The following hold:

- *If*

$$\sigma_i, l \leftarrow e \;\Downarrow^{\mathsf{C}}\; \sigma_i', \mathtt{T}_c^i; \quad \text{and}$$
$$\sigma_s, B[l \leftarrow e] \;\Downarrow^{\mathsf{C}}\; \sigma_s', \mathtt{T}_c^s$$

  then for any store σ_m satisfying

  1. $dom(\sigma_m) \supseteq dom(\sigma_i')$,
  2. $\forall l, l \in (def(\sigma_i') - def(\sigma_i)), \sigma_m(l) = \sigma_i'(l)$, *and*
  3. $B[\sigma_m] \sqsupseteq \sigma_s$,

  there exists a partial bijection B' such that

$$\sigma_m, l \leftarrow \mathtt{T}_c^i, \chi(B, \sigma_i, \sigma_s) \;\Downarrow_{\mathsf{C}}^{\mathsf{P}}\; \sigma_m', \mathtt{T}_c^m, \chi; \quad \text{where}$$

  1. $B' \supseteq B$
  2. $dom(B') = dom(B) \cup def(\mathtt{T}_c^m)$,
  3. $B'[\sigma_m'] \sqsupseteq \sigma_s'$,
  4. $B'[\mathtt{T}_c^m] = \mathtt{T}_c^s$, *and*
  5. $\chi = \chi(B', \sigma_i', \sigma_s')$.

- *If*

$$\sigma_i, e \ \Downarrow^{\mathtt{S}} \ v_i, \sigma_i', \mathtt{T}_s^i; \quad \textit{and}$$
$$\sigma_s, B[e] \ \Downarrow^{\mathtt{S}} \ v_i', \sigma_s', \mathtt{T}_s^s,$$

then for any store σ_m *satisfying*

1. $dom(\sigma_m) \supseteq dom(\sigma_i')$,
2. $\forall l, l \in (def(\sigma_i') - def(\sigma_i)), \sigma_m(l) = \sigma_i'(l)$, *and*
3. $B[\sigma_m] \sqsupseteq \sigma_s$,

there exists a partial bijection B' *such that*

$$\sigma_m, \mathtt{T}_s^i, \chi(B, \sigma_i, \sigma_s) \ \Downarrow_{\mathtt{S}}^{\mathtt{P}} \ \sigma_m', \mathtt{T}_s^m, \chi; \quad \textit{where}$$

1. $B' \supseteq B$,
2. $dom(B') = dom(B) \cup def(\mathtt{T}_s^m)$,
3. $B'[v_i] = v_i'$,
4. $B'[\sigma_m'] \sqsupseteq \sigma_s'$,
5. $B'[\mathtt{T}_s^m] = \mathtt{T}_s^s$, *and*
6. $\chi = \chi(B', \sigma_i', \sigma_s')$.

Proof: The proof is by simultaneous induction on evaluation. Among the changeable expressions, the most interesting are `write`, `let`, and `read`. Among the stable expression, the most interesting are the `let` and `mod`.

We refer to the following properties of modified store σ_m as the *modified-store properties*:

1. $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma_i')$,
2. $\forall l, l \in (\mathrm{def}(\sigma_i') - \mathrm{def}(\sigma_i)), \sigma_m(l) = \sigma_i'(l)$, and
3. $B[\sigma_m] \sqsupseteq \sigma_s$,

- **Write:** Suppose

$$\sigma_i, l \leftarrow \mathtt{write}_\tau(v) \ \Downarrow^{\mathtt{C}} \ \sigma_i[l \to v], \mathtt{W}_\tau; \quad \text{and}$$
$$\sigma_s, B[l \leftarrow \mathtt{write}_\tau(v)] \ \Downarrow^{\mathtt{C}} \ \sigma_s[B[l] \to B[v]], \mathtt{W}_\tau$$

then for store σ_m satisfying the modified-store properties we have

$$\sigma_m, l \leftarrow \mathtt{W}_\tau, \chi(B, \sigma_i, \sigma_s) \ \Downarrow_{\mathtt{C}}^{\mathtt{P}} \ \sigma_m, \mathtt{W}_\tau, \chi(B, \sigma_i, \sigma_s).$$

The partial bijection B satisfies the following properties:

1. $B \supseteq B$
2. $\mathrm{dom}(B) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{W}_\tau)$
3. $B[\sigma_m] \sqsupseteq \sigma_s[B[l] \to B[v]]$: We know that $B[\sigma_m] \sqsupseteq \sigma_s$ and thus we must show that $B[l]$ is mapped to $B(v)$ in $B[\sigma_m']$. Observe that $\sigma_m(l) = (\sigma_i[l \to v])(l) = v$ by Modified-Store Property 2, thus $B[\sigma_m](B[l]) = B[v]$.
4. $B[\mathtt{W}_\tau] = \mathtt{W}_\tau$

5. $\chi(B, \sigma_i, \sigma_s) = \chi(B, \sigma_i[l \to v], \sigma_s[B[l] \to B[v]])$, by definition.

Thus, pick $B' = B$ for this case.

- **Apply (Changeable):** Suppose that

$$\frac{(\text{I.1}) \quad \sigma_i, l \leftarrow [v/x, \texttt{func}_\texttt{C}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e\ \texttt{end}/f]\ e\ \Downarrow^\texttt{C}\ \sigma_i', \texttt{T}_c^i}{(\text{I.2}) \quad \sigma_i, l \leftarrow \texttt{apply}_\texttt{C}(\texttt{func}_\texttt{C}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e\ \texttt{end}, v)\ \Downarrow^\texttt{C}\ \sigma_i', \texttt{T}_c^i}$$

$$\frac{(\text{S.1}) \quad \sigma_s, B[l] \leftarrow [B[v]/x, B[\texttt{func}_\texttt{C}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e\ \texttt{end}]/f]\ e]\ \Downarrow^\texttt{C}\ \sigma_s', \texttt{T}_c^s}{(\text{S.2}) \quad \sigma_s, B[l \leftarrow \texttt{apply}_\texttt{C}(\texttt{func}_\texttt{C}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e\ \texttt{end}, v)]\ \Downarrow^\texttt{C}\ \sigma_s', \texttt{T}_c^s}$$

Consider evaluations (I.1) and (S.1) and a store σ_m that satisfies the modified-store properties. By induction we have a partial bijection B_0 and

$$\sigma_m, l \leftarrow \texttt{T}_c^i, \chi(B, \sigma_i, \sigma_s)\ \Downarrow_\texttt{C}^\texttt{P}\ \sigma_m, \texttt{T}_c^m, \chi,$$

where

1. $B_0 \supseteq B$,
2. $\text{dom}(B_0) = \text{dom}(B) \cup \text{def}(\texttt{T}_c^m)$,
3. $B_0[\texttt{T}_c^m] = \texttt{T}_c^s$, and
4. $B_0[\sigma_m] \sqsupseteq \sigma_s'$.
5. $\chi = \chi(B_0, \sigma_i', \sigma_s')$,

Since (I.2) and (S.2) return the trace and store returned by (I.1) and (S.1), we pick $B' = B_0$ for this case.

- **Let:**

$$\frac{\begin{array}{llll}(\text{I.1}) & \sigma_i, e & \Downarrow^\texttt{S} & v_i, \sigma_i', \texttt{T}_s^i \\ (\text{I.2}) & \sigma_i', l \leftarrow [v_i/x]e' & \Downarrow^\texttt{C} & \sigma_i'', \texttt{T}_c^i \end{array}}{(\text{I.3}) \quad \sigma_i, l \leftarrow \texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ e'\ \texttt{end}\ \Downarrow^\texttt{C}\ \sigma_i'', (\texttt{T}_s^i\ ;\ \texttt{T}_c^i)}$$

$$\frac{\begin{array}{llll}(\text{S.1}) & \sigma_s, B[e] & \Downarrow^\texttt{S} & v_s, \sigma_s', \texttt{T}_s^s \\ (\text{S.2}) & \sigma_s', B[l] \leftarrow [v_s/x]B[e'] & \Downarrow^\texttt{C} & \sigma_s'', \texttt{T}_c^s \end{array}}{(\text{S.3}) \quad \sigma_s, B[l \leftarrow \texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ e'\ \texttt{end}]\ \Downarrow^\texttt{C}\ \sigma_s'', (\texttt{T}_s^s\ ;\ \texttt{T}_c^s)}$$

Consider any store σ_m that satisfies the modified-store properties. The following judgement shows a change propagation applied with the store σ_m on the output trace $\texttt{T}_s^i\ ;\ \texttt{T}_c^i$.

$$\frac{\begin{array}{llll}(\text{P.1}) & \sigma_m, \texttt{T}_s^i, \chi(B, \sigma_i, \sigma_s) & \Downarrow_\texttt{S}^\texttt{P} & \sigma_m', \texttt{T}_s^m, \chi \\ (\text{P.2}) & \sigma_m', l \leftarrow \texttt{T}_c^i, \chi & \Downarrow_\texttt{C}^\texttt{P} & \sigma_m'', \texttt{T}_c^m, \chi' \end{array}}{(\text{P.3}) \quad \sigma_m, l \leftarrow (\texttt{T}_s^i\ ;\ \texttt{T}_c^i), \chi(B, \sigma_i, \sigma_s)\ \Downarrow_\texttt{C}^\texttt{P}\ \sigma_m'', (\texttt{T}_s^m\ ;\ \texttt{T}_c^m), \chi'}$$

We apply the induction hypothesis on (I.1) (S.1) and (P.1) to obtain a partial bijection B_0 such that

1. $B_0 \supseteq B$,
2. $\mathrm{dom}(B_0) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_s^m)$,
3. $v_s = B_0[v_i]$,
4. $B_0[\sigma'_m] \sqsupseteq \sigma'_s$,
5. $B_0[\mathtt{T}_s^m] = \mathtt{T}_s^s$, and
6. $\chi = \chi(B_0, \sigma'_i, \sigma'_s)$.

Using these properties, we now show that we can apply the induction hypothesis on (I.2) and (S.2) with the partial bijection B_0.

- $B_0[l \leftarrow [v_i/x]e'] = B[l] \leftarrow [v_s/x]B[e']$:
  By Properties 1 and 2 it follows that $B[l] = B_0[l]$.
  By Property 3, $B_0[v_i] = v_s$.
  To show that $B[e'] = B_0[e']$, note that $\mathrm{locs}(e) \subseteq \mathrm{dom}(\sigma_i) \subseteq \mathrm{dom}(\sigma_m)$ (since e is well typed with respect to σ_i). It follows that $\mathrm{dom}(\mathtt{T}_s^m) \cap \mathrm{locs}(e) = \emptyset$. Since $\mathrm{dom}(B_0) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_s^m)$, $B[e'] = B_0[e']$.
- $\chi = \chi(B_0, \sigma'_i, \sigma''_i)$. This is true by Property 6.
- σ'_m satisfies the modified-store properties:
  1. $\mathrm{dom}(\sigma'_m) \supseteq \mathrm{dom}(\sigma'_i)$
     This is true because $\mathrm{dom}(\sigma'_m) \supseteq \mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma''_i) \supseteq \mathrm{dom}(\sigma'_i)$.
  2. $\forall l, l \in (\mathrm{def}(\sigma''_i) - \mathrm{def}(\sigma'_i)), \sigma'_m(l) = \sigma''_i(l)$
     To show that $\forall l, l \in (\mathrm{def}(\sigma''_i) - \mathrm{def}(\sigma'_i)), \sigma'_m(l) = \sigma''_i(l)$, observe that
     (a) $\forall l, l \in (\mathrm{def}(\sigma''_i) - \mathrm{def}(\sigma_i)), \sigma_m(l) = \sigma''_i(l)$,
     (b) $\mathrm{def}(\sigma''_i) - \mathrm{def}(\sigma'_i) = \mathrm{def}(\mathtt{T}_c^i) \cup \{l\}$,
     (c) $\mathrm{def}(\mathtt{T}_s^i) \cap (\mathrm{def}(\sigma''_i) - \mathrm{def}(\sigma'_i)) = \emptyset$,
     and that the evaluation (P.1) changes values of locations only in $\mathrm{def}(\mathtt{T}_s^i)$.
  3. $B_0[\sigma'_m] \sqsupseteq \sigma'_s$, this follows by Property 4.

Now, we can apply the induction hypothesis on (I.2) (S.2) to obtain a partial bijection B_1 such that

1′. $B_1 \supseteq B_0$,
2′. $\mathrm{dom}(B_1) = \mathrm{dom}(B_0) \cup \mathrm{def}(\mathtt{T}_c^m)$,
3′. $B_1[\sigma''_m] \sqsupseteq \sigma''_s$,
4′. $B_1[\mathtt{T}_c^m] = \mathtt{T}_c$, and
5′. $\chi' = \chi(B_1, \sigma''_i, \sigma''_s)$.

Based on these, we have

1″. $B_1 \supseteq B$.
  This holds because $B_1 \supseteq B_0 \supseteq B$.

2″. $\mathrm{dom}(B_1) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_s^m \ ; \ \mathtt{T}_c^m)$.
  We know that $\mathrm{dom}(B_1) = \mathrm{dom}(B_0) \cup \mathrm{def}(\mathtt{T}_c^m)$ and $\mathrm{dom}(B_0) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_s^m)$. Thus we have $\mathrm{dom}(B_1) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_s^m) \cup \mathrm{def}(\mathtt{T}_c^m) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_s^m \ ; \ \mathtt{T}_c^m)$.

3″. $B_1[\sigma''_m] \sqsupseteq \sigma''_s$.
This follows by Property 3′.

4″. $B_1[\mathtt{T}^m_s \,;\, \mathtt{T}^m_c] = \mathtt{T}^s_s \,;\, \mathtt{T}^s_c$.
This holds if and only if $B_1[\mathtt{T}^m_s] = \mathtt{T}^s_s$ and $B_1[\mathtt{T}^m_c] = \mathtt{T}^s_c$.
We know that $B_0[\mathtt{T}^m_s] = \mathtt{T}^s_s$ and since $\mathrm{dom}(B_1) = \mathrm{dom}(B_0) \cup \mathrm{def}(\mathtt{T}^m_c)$ and $\mathrm{def}(\mathtt{T}^m_s) \cap \mathrm{def}(\mathtt{T}^m_c) = \emptyset$, we have $B_1[\mathtt{T}^m_s] = \mathtt{T}^s_s$. We also know that $B_1[\mathtt{T}^m_c] = \mathtt{T}^s_c$ by Property 4′.

5″. $\chi' = \chi(B_1, \sigma''_i, \sigma''_s)$,
This follows by Property 5′.

Thus we pick $B' = B_1$.

- **Read:** Assume that we have:

$$\frac{\text{(I.1)} \qquad \sigma_i, l' \leftarrow [\sigma_i(l)/x]e \;\Downarrow^{\mathsf{C}}\; \sigma'_i, \mathtt{T}^i_c}{\text{(I.2)} \qquad \sigma_i, l' \leftarrow \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end} \;\Downarrow^{\mathsf{C}}\; \sigma'_i, R^{x.e}_l(\mathtt{T}^i_c)}$$

$$\frac{\text{(S.1)} \qquad \sigma_s, B[l'] \leftarrow [\sigma_s(B[l])/x]B[e] \;\Downarrow^{\mathsf{C}}\; \sigma'_s, \mathtt{T}^s_c}{\text{(S.2)} \quad \sigma_s, B[l' \leftarrow \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end}] \;\Downarrow^{\mathsf{C}}\; \sigma'_s, R^{x.B[e]}_{B[l]}(\mathtt{T}^s_c)}$$

Consider a store σ_m that satisfies the modified-store properties. Then we have two cases for the corresponding change-propagation evaluation. In the first case $l \notin \chi$ and we have:

$$\frac{\text{(P.1)} \qquad \sigma_m, l' \leftarrow \mathtt{T}^i_c, \chi(B, \sigma_i, \sigma_s) \;\Downarrow^{\mathsf{P}}_{\mathsf{C}}\; \sigma'_m, \mathtt{T}^m_c, \chi}{\text{(P.2)} \quad \sigma_m, l' \leftarrow R^{x.e}_l(\mathtt{T}^i_c), \chi(B, \sigma_i, \sigma_s) \;\Downarrow^{\mathsf{P}}_{\mathsf{C}}\; \sigma'_m, R^{x.e}_l(\mathtt{T}^m_c), \chi} \; (l \notin \chi)$$

In this case, we apply the induction hypothesis on (I.1), (S.1), and (P.1) with the partial bijection B to obtain a partial bijection B_0 such that

1. $B_0 \supseteq B$,
2. $\mathrm{dom}(B_0) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}^m_c)$,
3. $B_0[\sigma'_m] \sqsupseteq \sigma'_s$,
4. $B_0[\mathtt{T}^m_c] = \mathtt{T}^s_c$, and
5. $\chi = \chi(B_0, \sigma'_i, \sigma'_s)$.

Furthermore, the following hold for B_0,

1. $\mathrm{dom}(B_0) = \mathrm{dom}(B) \cup \mathrm{def}(R^{x.e}_l(\mathtt{T}^m_c))$.
This follows by Property 2 and because $\mathrm{def}(R^{x.e}_l(\mathtt{T}^m_c)) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}^m_c)$,

2. $B_0[R^{x.e}_l(\mathtt{T}^m_c)] = R^{x.B[e]}_{B[l]}(\mathtt{T}^s_c)$.
We have $B_0[R^{x.e}_l(\mathtt{T}^m_c)] = R^{x.B_0[e]}_{B_0[l]}(B_0[\mathtt{T}^m_c]) = R^{x.B_0[e]}_{B_0[l]}(\mathtt{T}^s_c)$, because of (c). Thus we need to show that $B_0[l] = B[l]$ and $B_0[e] = B[e]$. This is true because,
(a) $l \notin \mathrm{def}(\mathtt{T}^m_c)$ and thus $B[l] = B_0[l]$, and

(b) $\forall l, l \in \text{locs}(e)$ we have $l \in \text{dom}(\sigma_m)$ and thus $l \notin \text{def}(\mathtt{T}_c^m)$, which implies that $B[l] = B_0[l]$, and $B[e] = B_0[e]$.

Thus we pick $B' = B_0$.

In the second case, we have $l \in \chi$ and the read $R_l^{x.e}$ is re-evaluated.

$$\frac{\text{(P.4)} \qquad \sigma_m, l' \leftarrow [\sigma_m(l)/x]e \;\Downarrow^{\mathtt{C}}\; \sigma'_m, \mathtt{T}_c^m}{\text{(P.5)} \qquad \sigma_m, l' \leftarrow R_l^{x.e}(\mathtt{T}_c), \chi(B, \sigma_i, \sigma_s) \;\Downarrow_{\mathtt{C}}^{\mathtt{P}}\; \sigma'_m, R_l^{x.e}(\mathtt{T}_c^m), \chi(B, \sigma_i, \sigma_s) \cup \{l'\}} \; (l \in \chi)$$

Since $B[\sigma_m] \sqsupseteq \sigma_s$, the evaluation in (P.4) is identical to the evaluation in (S.1) and thus, there is a bijection $B_1 \supseteq B$ such that $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$ and $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$. Thus we have

1. $B_1 \supseteq B$,
2. $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$,
3. $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$,
4. $\chi(B, \sigma_i, \sigma_s) \cup \{l\} = \chi(B_1, \sigma'_i, \sigma'_s)$
   To show this, observe that
   (a) $\text{dom}(\sigma'_i) \cap \text{def}(\mathtt{T}_c^m) = \emptyset$, because $\text{dom}(\sigma_m) \supseteq \text{dom}(\sigma'_i)$.
   (b) $\chi(B_1, \sigma'_i, \sigma'_s) = \chi(B, \sigma'_i, \sigma'_s)$, because $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$.
   (c) $\chi(B, \sigma'_i, \sigma'_s) = \chi(B, \sigma_i, \sigma_s) \cup \{l'\}$, because $\text{dom}(B) \subseteq \text{dom}(\sigma_i)$. (Assuming, without loss of generality, that the value of l' changes because of the re-evaluation).
5. $B_1[\sigma'_m] \sqsupseteq \sigma'_s$
   We know that $B_1[\sigma_m] \sqsupseteq \sigma_s$. Furthermore $B_1[\sigma'_m - \sigma_m] = \sigma'_s - \sigma_s$ and thus, $B_1[\sigma'_m] \sqsupseteq \sigma'_s$.

Thus pick $B' = B_1$.

- **Value:** Suppose that

$$\sigma_i, v \;\Downarrow^{\mathtt{S}}\; v, \sigma_i, \varepsilon$$
$$\sigma_s, B[v] \;\Downarrow^{\mathtt{S}}\; B[v], \sigma_s, \varepsilon.$$

Let σ_m be any store that satisfies the modified-store properties. We have

$$\sigma_m, \varepsilon, \chi(B, \sigma_i, \sigma_s) \;\Downarrow_{\mathtt{S}}^{\mathtt{P}}\; \sigma_m, \varepsilon, \chi(B, \sigma_i, \sigma_s)$$

where

1. $B \supseteq B$.
2. $\text{dom}(B) = \text{dom}(B) \cup \text{def}(\varepsilon)$.
3. $B[\sigma_m] \sqsupseteq \sigma_s$, by Modified-Store Property 3.
4. $B[\varepsilon] = \varepsilon$.
5. $\chi(B, \sigma_i, \sigma_s) = \chi(B, \sigma_i, \sigma_s)$.

Thus pick $B' = B$.

- **Apply (Stable):** This is similar to the apply in the changeable mode.
- **Mod:** Suppose that

$$\frac{\begin{array}{ll}(\text{I.1}) & \sigma_i[l_i \to \square], l_i \leftarrow e \;\Downarrow^{\mathtt{C}}\; \sigma_i', \mathtt{T}_c^i \\ (\text{I.2}) & \sigma_i, \mathtt{mod}_\tau\ e \;\Downarrow^{\mathtt{S}}\; l_i, \sigma_i', \langle \mathtt{T}_c^i \rangle_{l_i:\tau}\end{array}}{} \; l_i \notin \mathrm{dom}(\sigma_i)$$

$$\frac{\begin{array}{ll}(\text{S.1}) & \sigma_s[l_s \to \square], l_s \leftarrow B[e] \;\Downarrow^{\mathtt{C}}\; \sigma_s', \mathtt{T}_c^s \\ (\text{S.2}) & \sigma_s, B[\mathtt{mod}_\tau\ e] \;\Downarrow^{\mathtt{S}}\; l_s, \sigma_s', \langle \mathtt{T}_c^s \rangle_{l_s:\tau}\end{array}}{} \; l_s \notin \mathrm{dom}(\sigma_s).$$

  Let σ_m be a store that satisfies the modified-store properties. Then we have

$$\frac{\begin{array}{ll}(\text{P.1}) & \sigma_m, l_i \leftarrow \mathtt{T}_c^i, \chi(B, \sigma_i, \sigma_s) \;\Downarrow^{\mathtt{p}}_{\mathtt{C}}\; \sigma_m', \mathtt{T}_c^m, \chi \\ (\text{P.2}) & \sigma_m, \langle \mathtt{T}_c^i \rangle_{l_i:\tau}, \chi(B, \sigma_i, \sigma_s) \;\Downarrow^{\mathtt{p}}_{\mathtt{S}}\; \sigma_m', \langle \mathtt{T}_c^m \rangle_{l_i:\tau}, \chi\end{array}}{}$$

  Consider the partial bijection $B_0 = B[l_i \mapsto l_s]$. It satisfies the following:

  - $B_0[l_i \leftarrow e] = l_s \leftarrow B[e]$.
    Because $B_0(l_i) = l_s$ and $l_i \notin \mathrm{locs}(e)$.
  - $B_0[\sigma_m] \sqsupseteq \sigma_s[l_s \mapsto \square]$.
    We know that $B[\sigma_m] \sqsupseteq \sigma_s$ by Modified-Store Property 3. Since $l_s \notin \mathrm{dom}(\sigma_s)$, we have $B_0[\sigma_m] \sqsupseteq \sigma_s$.
    Furthermore $l_s = B_0(l_i)$ and $l_i \in \mathrm{dom}(\sigma_m)$ because $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma_i')$.
    Thus $B_0[\sigma_m] \sqsupseteq \sigma_s[l_s \mapsto \square]]$.
  - $\forall l, l \in (\mathrm{def}(\sigma_i') - \mathrm{def}(\sigma_i[l_i \to \square])), \sigma_m(l) = \sigma_i'(l)$.
    Because $\forall l, l \in (\mathrm{def}(\sigma_i') - \mathrm{def}(\sigma_i)), \sigma_m(l) = \sigma_i'(l)$ by Modified-Store Property 2.

  Thus, we can apply the induction hypothesis on (I.1), (S.1) with the partial bijection $B_0 = B[l_i \mapsto l_s]$ to obtain a partial bijection B_1 such that the following hold.

  1. $B_1 \supseteq B_0$,
  2. $\mathrm{dom}(B_1) = \mathrm{dom}(B_0) \cup \mathrm{def}(\mathtt{T}_c^i)$,
  3. $B_1[\sigma_m'] \sqsupseteq \sigma_s'$,
  4. $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$, and
  5. $\chi = \chi(B_1, \sigma_s', \sigma_i')$.

  Furthermore, B_1 satisfies

  1. $\mathrm{dom}(B_1) = \mathrm{dom}(B) \cup \mathrm{def}(\langle \mathtt{T}_c^i \rangle_{l_i:\tau})$.
     By Property 2 and because $\mathrm{def}(\mathtt{T}_c^i) = \mathrm{def}(\langle \mathtt{T}_c^i \rangle_{l_i:\tau})$.
  2. $B_1[\langle \mathtt{T}_c^m \rangle_{l_i:\tau}] = \langle \mathtt{T}_c^s \rangle_{l_s:\tau}$.
     Because $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$ by Property 4 and $B_1(l_i) = l_s$.

  Thus we can pick $B' = B_1$.

- **Let (Stable):** This is similar to the let rule in changeable mode.

■

8 Discussion

Variants. In the process of developing the mechanisms presented in this paper we considered several variants. One variant is to replace the explicit write operation with an implicit one. In the ML library this requires making the target destination an argument to the `read` operation. In AFL it requires adding some implicit type subsumption rules. We decided to include the explicit `write` since we believe it is cleaner.

Side Effects. We require that the underlying language be purely functional. The main reason for this is that each edge (read) stores a closure (code and environment) which might be re-evaluated. It is critical that this closure does not change. The key requirement, therefore, is not that there are no side-effects, but rather that all data is persistent (i.e., the closure's environment cannot be modified). It is therefore likely that the adaptive mechanism could be made to work in an imperative setting as long as relevant data structures are persistent. There has been significant research on persistent data-structures under an imperative setting [13, 12, 14].

We further note that certain "benign" side effects are not harmful. For example, side effects to objects that are not examined by the adaptive code itself are harmless. This includes print statements and changes to "meta" data structures that are somehow recording the progress of the adaptive computation itself. For example, one way to determine which parts of the code are being re-evaluated is to sprinkle the code with print statements and see which ones print during the change propagation. Similarly, re-evaluations of a function can be counted by defining a global counter and incrementing (side effecting) the counter every time the function is called. Also, the memoization of the kind done by lazy languages will not affect the correctness of change-propagation, because the value remains the same whether it has been calculated or not. We therefore expect that our approach can be applied to lazy languages, but we have not explored this direction.

Applications. The work in this paper was motivated by the desire to make it easier to define kinetic data structures for problems in computational geometry [7]. Consider the problem of maintaining some property of a set of objects in space as they move, such as the nearest neighbors or convex hull of a set of points. Kinetic data structures are designed to maintain such properties by re-evaluating parts of the code when certain conditions become violated (e.g., a point moves from one side of a line to the other). Currently, however, every problem requires the design of its own kinetic data structure. We believe that it is possible, instead, to use adaptive versions of non-kinetic algorithms.

Full Adaptivity. It is not difficult to modify the AFL semantics to interpret standard functional code (e.g. the call-by-value lambda-calculus) in a fully adaptive way (i.e., all values are stored in modifiables, and all expressions are changeable). It is also not hard to describe a translator for converting functional code into AFL, such that the result is fully adaptive. The only slightly tricky aspect is translating recursive functions. We in fact had originally considered defining a fully adaptive version of AFL but decided against it since we felt it would be more useful to selectively choose what code is adaptive.

Meta Language. We have not included a "meta" language for AFL that would allow a program to change input and run change-propagation. There are some subtle issues in defining such a language such as how to restrict changes to inputs, and how to identify the "safe" parts of the code in which the program can make changes. We worked on a system that includes an additional type mode, which we called meta-stable. Changes and change-propagation could be performed only in

this mode, and there was no way to get into this mode other than from top-level. We felt, however, that this system did not add much to the main concepts covered in this paper.

9 Conclusion

We have presented a mechanism for adaptive computation based on the idea of a modifiable reference. We expect that this mechanism can be incorporated into any purely functional call-by-value language. A key aspect of our mechanism is that it can dynamically create new computations and delete old computations. The main contributions of the paper are the particular set of primitives we suggest, the change-propagation algorithm, and the semantics along with the proofs that it is sound. The simplicity of the primitives is achieved by using a destination passing style. The efficiency of the change-propagation is achieved by using an optimal order-maintenance algorithm. The soundness of the semantics is aided by a modal type system.

Acknowledgements We are grateful to Frank Pfenning for his advice on modal type systems.

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. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[3] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. In *ACM SIGPLAN Workshop on ML*, 2005.

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

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

[6] Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*, 2005.

[7] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.

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

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

[10] 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.

[11] 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.

[12] Paul F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer-Verlag, August 1989.

[13] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[14] James R. Driscoll, Daniel D. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, 1994.

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

[16] John Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems*. PhD thesis, Department of Computer Science, Cornell University, November 1991.

[17] A. Heydon, R. Levin, T. Mann, and Y. Yu. The vesta approach to software configuration management, 1999.

[18] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on PLDI*, pages 311–320, May 2000.

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

[20] 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.

[21] Yanhong Annie Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach*. PhD thesis, Department of Computer Science, Cornell University, January 1996.

[22] 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.

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

[24] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 487–489, 1985.

[25] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

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

[27] 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.

[28] 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.

[29] Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on POPL*, pages 169–176, January 1982.

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

[31] R. S. Sundaresh and Paul Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on POPL*, pages 1–13, January 1991.

[32] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991.