Adaptive Functional Programming*

Umut A. Acar Guy E. Blelloch Robert Harper

16 January 2002
CMU-CS-01-161

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

An adaptive computation maintains the relationship between its input and output as the input changes. Although various techniques for adaptive computing have been proposed, they remain limited in their scope of applicability. We propose a general mechanism for adaptive computing that enables one to make any purely-functional program adaptive.

We show that the mechanism is 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.

*This research was supported in part by NSF grants CCR-9706572, CCR-0085982, and CCR-0122581. This report is a complete version of [1].

Keywords: Incremental Computing, Functional Languages, Dynamic Algorithms

1 Introduction

An adaptive program responds to input changes by updating its output while only re-evaluating those portions of the program affected by the change. Adaptive programming is useful in situations where 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 obtain the updated output more quickly than a complete re-evaluation. For example, as we shall see below, an adaptive version of Quicksort takes expected logarithmic time to adapt its output when its input list is extended by one key. This is an improvement by a linear factor over simply re-evaluating the sort for the changed inputs.

In this paper we propose a general mechanism for adaptive programming. 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.
- **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.
- **Efficiency:** The mechanism admits a simple implementation and yields efficient adaptivity. For example, the adaptive version of Quicksort updates the output in expected $O(\log n)$ time upon extension to the input.

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 allows us to record 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 dependency 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 may be maintained as a graph in which 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. 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 dependency graph connecting readers to writers. It is also essential to maintain an ordering on the edges and keep track of which edges (reads) 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. The 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 [4].

2 Related Work

Several researchers have studied approaches that are similar to what we call adaptive programming. The idea of using dependency graphs for incremental updates was introduced by Demers, Reps and Teitelbaum [3] in the context of attribute grammars. Reps then showed an algorithm to propagate a change optimally [16], and Hoover generalized the approach outside the domain of attribute grammars [9]. 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 [14]. 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 [18], 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.

Another approach to incremental/adaptive computations is function caching [14, 13]. In function caching, a computation reuses cached results from earlier evaluations whenever appropriate. Thus, one must run the computation from scratch to identify the part of the computation that does not change. In contrast, in our approach, an input change pinpoints the parts of the computation that need to be re-evaluated. Function caching therefore is bad at handling "deep" modifications. We conjecture, for example, that with function caching no algorithm can update a sorted linked-list in less than linear expected time. This is because the inserted element is expected to end up half way down the list, and function caching will always recreate the part of the list ahead of the inserted element. There are two other problems with function caching. First it can be hard to effectively check for equality of arguments for the purpose of matching elements in the cache. This is particularly true if the inputs are functions themselves, possibly with captured environments. Second, for efficiency it is critical to evict elements from the cache. The suggested methods we have seen to decide when and what to evict seem ad-hoc, although Liu and Teitelbaum have made some progress using automatic program transformation techniques to decide what to cache [11, 10]. In spite of these problems, function caching might have some advantages over our method for "shallow" modifications. We expect that these techniques can be integrated to further improve performance in certain situations.

Other approaches are based on various forms of partial evaluation [8, 17]. These approaches are arguably cleaner than the function caching approach (they don't have the issues with equality of inputs or deciding when to evict from the cache), but are even more limited in the type of adaptivity they allow. Ramalingam and Reps wrote an excellent bibliography summarizing other work on incremental computation [15].

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 Standard ML based on the interface of a module implementing the basic adaptivity mechanisms. We then describe the change-propagation algorithm that lies at the heart of the mechanism and establish an upper bound for its running time. Using this bound, we then prove the expected $O(\log n)$ time bound for adaptive Quicksort to accommodate 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.

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-executed. The trace also records the containment structure of the computation, which is updated during change propagation. Using this presentation we give a proof of correctness of the change propagation algorithm stating that change propagation yields essentially the same result as a complete re-execution on the changed inputs.

We note that we had originally thought that incorporating an adaptivity mechanism in ML would require the involvement of a compiler. Working out the semantics of AFL led to the particular mechanism we describe and its simple implementation as an ML library.

```
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 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.

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

[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 in which run-time checks are needed to check correctness. The language described in Section 5 enforces all these restrictions statically using a modal type system.

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 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.

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. 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` is defined to take an initializer and pass it to the `mod` 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.

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

fun fromList(nil) =
      let val m = newElt(NIL)
      in (m,m)
      end
| fromList(h::r) =
      let val (l,last) = fromList(r)
      in (newElt(CONS(h,l)),last)
      end

fun test(lst,v) =
let
   val _ = init()
   val (l,last) = fromList(lst)
   val r = qsort'(l)
in
   (change(last,CONS(v,newElt(NIL)));
    propagate();
    r)
end
```

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

4.3 Adaptivity

An adaptive computation allows the programmer to change input values and update the result. 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 `fromList` function 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 Augmented Dependency 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. It is helpful to visualize this record as a dependency graph augmented with additional information regarding the containment hierarchy and the evaluation order of reads. In such a dependency 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. We also tag the edges with the reader function.

To operate correctly, the change-propagation algorithm needs to know the containment hierarchy of reads. To maintain this information, we tag each edge and node with a *time stamp*, which are generated by the `mod` and `read`. All expressions are evaluated in a time range (t_s, t_e) and time-stamps generated by the expression are allocated sequentially within

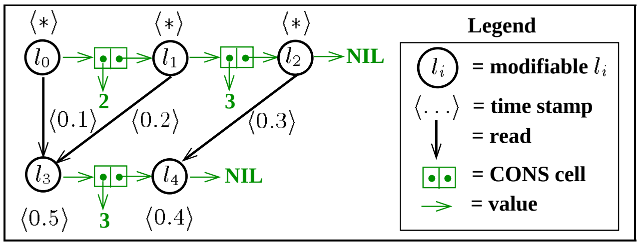

Figure 4: The ADG for an application of `filter'` to the function `fn x => x>2` and the input modifiable list `2::3::nil`. The output is the modifiable list `3::nil`.

that range, *i.e.*, each generated time stamp is greater than the previous one, but less than the end of the time range. 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 stamp of a node corresponds to the time it was initialized). 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. We note that the time span can be used to identify the containment relationship of reads. In particular, a read R_a is contained in a read R_b if and only if the start time 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, and assume that top-level expressions are evaluated in the range $(0.0, 1.0)$. Subsequently, we will show how the Dietz-Sleator Order-Maintenance Algorithm can be used to maintain time stamps efficiently [4].

We define an *augmented dependency graph* (ADG) as a DAG in which each edge has an associated reader and time stamp, and each node has an associated value and time stamp.[2] We say that a node (and corresponding modifiable) is an *input* if it has no incoming edges.

An example should help make the ideas clear. Consider the adaptive filter function `filter'` shown in Figure 2. The function takes another function `f` and a modifiable list `l` as parameters and outputs a modifiable list that contains the items of `l` satisfying `f`. Figure 4 shows the dependency 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 an asterisk in Figure 4.

4.5 Change Propagation.

Given an augmented dependency graph and a set of changed input modifiables, the *change-propagation algorithm* updates the ADG and the output by propagating changes in the ADG. We say that an edge, or corresponding read, is *invalidated* if the source of the edge changes value. We say that an edge is *obsolete* if it is contained within an invalidated edge.

[2]We do not formalize ADGs more precisely here since we view them as an implementation of a cleaner notion of traces, which we formalize in Section 5.

Propagate Changes

I is the set of changed inputs

$(V, E) = G$ is an ADG

1 $Q = \bigcup_{v \in I}$ outEdges(v)
2 while Q is not empty
3 $\quad e =$ deleteMin(Q)
4 $\quad (T_s, T_e) =$ timeSpan(e)
5 $\quad V = V - \{v \in V | T_s < T(v) < T_e\}$
6 $\quad E' = \{e' \in E | T_s < T(e') < T_e\}$
7 $\quad E = E - E'$
8 $\quad Q = Q - E'$
9 $\quad v' =$ apply(reader(e), val(src(e))) in time (T_s, T_e)
10 $\quad$ if $v' \neq$ val(target(e)) then
11 $\quad\quad$ val(target(e)) $= v'$
12 $\quad\quad Q = Q +$ outEdges(target(e))

Figure 5: The change-propagation algorithm.

Figure 5 defines the change-propagation algorithm. The algorithm maintains a Priority Queue of invalidated 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 of the while loop processes one invalidated edge, and we call the iteration an *edge update*. The update re-evaluates the associated reader. This makes any code that was within the reader's dynamic scope obsolete. A key aspect of the algorithm is that when an edge is updated, all nodes and edges that are contained within that edge are deleted from both the graph and queue. This prevents the reader of an obsolete edge from being re-evaluated. Evaluating such a reader on a changed input may incorrectly update a modifiable, incorrectly raise an exception, or even not terminate. After the reader is re-evaluated we check if the value of the target has changed (line 10) by using the conservative comparison function passed to `mod`. If it has changed, we add the out-edges of the target to the queue to propagate that change.

As an example, consider an initial evaluation of `filter` whose dependency 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. The top left frame in Figure 6 shows the input change. Now, we run the change-propagation algorithm to update the output. First, we insert the sole outgoing edge of l_1, namely (l_1,l_3), into the queue. Since this is the only (hence, the earliest) edge in the queue, we remove it from the queue and establish the current time-span as $\langle 0.2\rangle$-$\langle 0.5\rangle$. Next, we delete all the nodes and edges contained in this edge from the ADG and from the queue (which is empty) as shown by the top right frame in Figure 6. Then we redo the read by re-evaluating the reader `(fn l' => case l' of ...)` (8-15 in Figure 2) in the current time span $\langle 0.2\rangle$-$\langle 0.5\rangle$. 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, as shown in the bottom frame in Figure 6. This creates two new edges, which are given the time stamps, $\langle 0.3\rangle$, and $\langle 0.4\rangle$. The targets of these edges, l_7 and l_8, are assigned the time stamps, $\langle 0.475\rangle$, and $\langle 0.45\rangle$, matching the order that they were initialized (these time stamps are otherwise chosen arbitrarily to fit in the range $\langle 0.4\rangle$-$\langle 0.5\rangle$).

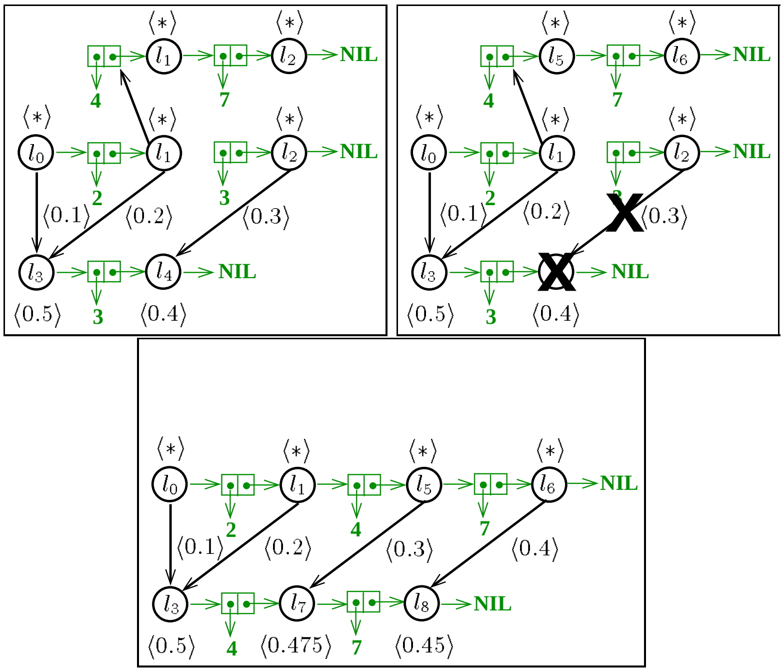

Figure 6: Snapshots of the ADG during change propagation.

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 ADG needs to support deleting an edge, a node, and finding the outgoing edges of a node. An adjacency list representation in which the edges of a node are maintained in a doubly-linked list implements these operations in constant time. The algorithm also needs to identify all the edges between two time stamps so they can be deleted. This can be implemented with a time-ordered, doubly-linked list of all edges. Inserting, deleting, and find-next all take constant time per edge. The priority queue should support addition, deletion, and delete-minimum operations efficiently. Standard balanced-tree based priority-queue algorithms perform these operations in logarithmic time. This is sufficient for our purposes and any of these algorithms can be used to implement priority queues.

A more interesting question is how to implement time stamps efficiently. To do this, we require efficient support for four operations: compare two time stamps, insert a new time stamp after a given time stamp, delete a time stamp, and retrieve the next time stamp (used in deleting the time-span of an edge). Using real numbers is not an efficient solution, because, in change propagation, an arbitrary number of new time stamps could be inserted between two fixed time stamps. This requires arbitrary precision real numbers, which are costly. A simple alternative to real numbers is to have all the time stamps ordered in a list. To insert or delete a time stamp, we simply insert it into the list or delete it from the list. To compare two time stamps, we compare their positions in the list—the time stamp closer to the beginning of the list is smaller. This comparison operation, however, can take linear time in the length of the list. A more efficient approach is to assign an integer rank to each time stamp in the list 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 then may have to do some re-ranking to find space to insert an integer between two adjacent integers. Dietz and Sleator give two efficient algorithms for this problem, which is known as the Order-Maintenance Problem [4]. The first algorithm is a simple algorithm that performs all operations in amortized constant time, the second more sophisticated algorithm 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 augmented dependency 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, and find-next operations in constant time.

We define several performance measures for change propagation. Consider running the change-propagation algorithm, and let I denote the set of all invalidated edges. Of these edges, some of them participate in an edge update, whereas some become obsolete and are deleted before participating. We refer to the set of updated edges as I_u. For an updated edge $e \in I_u$, 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

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

Proof: The time for propagate can be partitioned into 4 items: (1) re-evaluation of readers, (2) creation of time stamps, (3) deletion of time stamps and contained edges, and (4) insertion to and deletions from the priority queue. Re-evaluation of the readers takes $\sum_{e \in I_u} |e|$ 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, creating time stamps takes $O(\sum_{e \in I_u} |e|)$ time. Determining each time stamp to delete, deleting the time stamp and the corresponding node or edge from the ADG and the time-ordered doubly-linked edge list takes constant time. Thus total time for these deletions is $(\sum_{e \in I_u} ||e||)$.

Finally, each edge is added to the priority queue once and deleted from the queue once, thus the time for maintaining the priority queue is $O(|I| \log q)$. The total time is the sum of these terms. ■

4.8 Performance of Adaptive Quicksort

We now analyze the propagate 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.

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 invalidated reads is at most $2h$ ($|I| \leq 2h$ and $I_u = I$), each reader takes constant time to re-evaluate ($\forall e \in I, |e| = O(1)$), the time span of a reader contains no other time stamps ($\forall e \in I, ||e|| = 0$), 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) are invalidated, will be added to the queue, and then updated. Any other edges that these updates add to the queue will have start times after the start times of these edges. Re-evaluating the reader of one of the two edges will rewrite NIL and therefore not change its target. Re-evaluating the other will change its target from NIL to the value `CONS(`v`,`l_{n+1}`)`, and therefore extend the corresponding list. Re-evaluating both readers takes constant time and the update deletes no time stamps. Only one of the two recursive calls to `qs` has any changed data, and that one has its input extended with one element. Since the call tree of the `qs` has depth at most $d - 1$, the induction hypothesis applies. Thus, $|e| = O(1)$ and $||e|| = 0$ for all invalidated edges. Furthermore, the total number of invalidated edges is $|I| \leq 2(d-1)+2 = 2d$ and all edges are re-evaluated ($I_u = I$). 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 invalidated 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: $\mathrm{E}[|I|] = O(\log n)$, $I = I_u$, $q = 4$, and $\forall e \in I, |e| = O(1), ||e|| = 0$. 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. ∎

4.9 The ML Implementation

We present an implementation of our adaptive mechanism in ML. It uses 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 7), `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 8 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 ADG. The reader and time-span are represented explicitly in the `edge` type, but the source and destination are implicit in

```
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 7: The signature of an ordered list.

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 37 and for the node on line 29 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 20–25 in the ML code. Another important difference is that the deletion of contained edges is done lazily. Instead of deleting edges from the Queue and from the graph immediately, the time stamp of the edge is marked as invalid (by being removed from the ordered-list data structure), and is deleted when it is next encountered. This can be seen in line 55.

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.

```
structure Adaptive :> ADAPTIVE = struct
  type changeable = unit
  exception unsetMod

  type edge = {reader:  (unit -> unit),
               timeSpan:  (Time.t * Time.t)}

  type 'a node = {value :  (unit -> 'a) red,
                  wrt :  ('a -> unit) ref,
                  outEdges :  edge list ref}
  type 'a mod = 'a node
  type 'a dest = 'a node

  val currentTime = ref(Time.init())
  val PQ = ref(Q.empty)              (* Priority queue *)

  fun init() = (currentTime := Time.init(); PQ := Q.empty)

  fun mod cmp f = let
    val value = ref(fn() => raise unsetMod)
    val wrt = ref(fn(v) => raise unsetMod)
    val outEdges = ref(nil)
    val m = {value=value, wrt=wrt, outEdges=outEdges}
    fun change t v =
      (if cmp(v,(!value)()) then ()
       else
         (value := (fn() => v);
          List.app (fn x => PQ := Q.insert(x,!PQ))
                   (!outEdges);
          outEdges := nil);
       currentTime := t)
    fun write(v) =
      (value := (fn() => v);
       Time.insert(currentTime);
       wrt:= change(!currentTime))
    val _ = wrt := write
  in
    f(m); m
  end

  fun write({wrt, ...} :  'a dest, v) = (!wrt)(v)

  fun read({value, outEdges, ...} :  'a mod, f) = let
    val start = Time.insert(currentTime)
    fun run() =
      (f((!value)());
       outEdges := {reader=run,
                    timeSpan=(start,(!currentTime))}
                   ::(!outEdges))
  in
    run()
  end

  fun change(l:  'a mod, v) = write(l, v)

  fun propagate'() =
    if (Q.isEmpty(!PQ)) then
      ()
    else let
      val (edge, pq) = Q.deleteMin(!PQ)
      val _ = PQ := pq
      val {reader=f,timeSpan=(start,stop)} = edge
    in
      if (Time.isSplicedOut start) then
        propagate'()        (* Obsolete read, discard.*)
      else
        (Time.spliceOut(start,stop);    (* Splice out *)
         currentTime := start;
         f();                           (* Rerun the read *)
         propagate'())
    end

  fun propagate() = let
    val ctime = !currentTime
  in
    (propagate'();
     currentTime := ctime)
  end
end
```

Figure 8: 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 informal 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 with primitives for adaptive computation, called AFL. 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. Because of this, AFL also includes two forms of function type, one for functions whose body is stable, and one for functions whose body is changeable. The former corresponds to the standard function type found in any functional language. The latter is included to improve efficiency by allowing such functions to share their (implicit) target with the caller. This avoids the need to allocate a modifiable for the result of a procedure call, and is crucial to supporting the tail recursion optimization in changeable mode.

AFL does not include analogues of the meta-operations for making and propagating changes found 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. Just as with the ML implementation, the dynamic semantics must keep a record of the adaptive aspects of the computation. However, rather than use ADG's, the semantics maintains this information in the form of a trace, which guides the change propagation algorithm. By doing so we are able to give a relatively straightforward proof of correctness of the change propagation algorithm in Section 7.

5.1 Abstract Syntax.

The abstract syntax of AFL is given in Figure 9. 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 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, $\tau_1 \xrightarrow{\mathtt{s}} \tau_2$; the changeable function type, $\tau_1 \xrightarrow{\mathtt{c}} \tau_2$; 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 $\mathtt{mod}_\tau\ e_c$ 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 `write`(v) writes the value v into the target. The changeable ex-

$$
\begin{array}{llcl}
\textit{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau\ \texttt{mod} \mid \tau_1 \xrightarrow{\mathsf{S}} \tau_2 \mid \tau_1 \xrightarrow{\mathsf{C}} \tau_2 \\[1ex]
\textit{Values} & v & ::= & c \mid x \mid l \mid \texttt{fun}_{\mathsf{S}}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e_s\ \texttt{end} \mid \\
 & & & \texttt{fun}_{\mathsf{C}}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e_c\ \texttt{end} \\[1ex]
\textit{Op's} & o & ::= & \texttt{not} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots \\[1ex]
\textit{Const's} & c & ::= & n \mid \texttt{true} \mid \texttt{false} \\[1ex]
\textit{Exp's} & e & ::= & e_s \mid e_c \\[1ex]
\textit{St Exp's} & e_s & ::= & v \mid o(v_1,\ldots,v_n) \mid \texttt{apply}_{\mathsf{S}}(v_1,v_2) \mid \\
 & & & \texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_s'\ \texttt{end} \mid \texttt{mod}_\tau\ e_c \mid \\
 & & & \texttt{if}\ v\ \texttt{then}\ e_s\ \texttt{else}\ e_s' \\[1ex]
\textit{Ch Exp's} & e_c & ::= & \texttt{write}(v) \mid \texttt{apply}_{\mathsf{C}}(v_1,v_2) \mid \\
 & & & \texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_c\ \texttt{end} \\
 & & & \texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c\ \texttt{end} \mid \\
 & & & \texttt{if}\ v\ \texttt{then}\ e_c\ \texttt{else}\ e_c'
\end{array}
$$

Figure 9: The abstract syntax of AFL.

pression `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.

We defined the set of location of an expression as follows.

Definition 3 (Locations of an expression)
The locations of an expression e, denoted locs(e), is defined as follows.

- **Values**

$$
\begin{array}{rcl}
\mathrm{locs}(c) & = & \emptyset \\
\mathrm{locs}(x) & = & \emptyset \\
\mathrm{locs}(l) & = & \{l\} \\
\mathrm{locs}(\texttt{fun}_{\mathsf{S}}\ f(x:\tau):\tau'\ \texttt{is}\ e_s\ \texttt{end}) & = & \mathrm{locs}(e_s) \\
\mathrm{locs}(\texttt{fun}_{\mathsf{C}}\ f(x:\tau):\tau'\ \texttt{is}\ e_c\ \texttt{end}) & = & \mathrm{locs}(e_c)
\end{array}
$$

- **Stable Expressions**

$$
\begin{array}{rcl}
\mathrm{locs}(o(v_1,\ldots,v_n)) & = & \mathrm{locs}(v_1) \cup \ldots \cup \mathrm{locs}(v_n) \\
\mathrm{locs}(\texttt{apply}_{\mathsf{S}}(v_1,v_2)) & = & \mathrm{locs}(v_1) \cup \mathrm{locs}(v_2) \\
\mathrm{locs}(\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_s'\ \texttt{end}) & = & \mathrm{locs}(e_s) \cup \mathrm{locs}(e_s') \\
\mathrm{locs}(\texttt{if}\ v\ \texttt{then}\ e_s\ \texttt{else}\ e_s') & = & \mathrm{locs}(v) \cup \mathrm{locs}(e_s) \cup \mathrm{locs}(e_s') \\
\mathrm{locs}(\texttt{mod}_\tau\ e_c) & = & \mathrm{locs}(e_c)
\end{array}
$$

- **Changeable Expressions**

$$
\begin{array}{rcl}
\mathrm{locs}(\mathtt{apply}_{\mathtt{C}}(v_1, v_2)) & = & \mathrm{locs}(v_1) \cup \mathrm{locs}(v_2) \\
\mathrm{locs}(\mathtt{let}\ x\ \mathtt{be}\ e_s\ \mathtt{in}\ e_c\ \mathtt{end}) & = & \mathrm{locs}(e_s) \cup \mathrm{locs}(e_c) \\
\mathrm{locs}(\mathtt{if}\ v\ \mathtt{then}\ e_s\ \mathtt{else}\ e_c) & = & \mathrm{locs}(v) \cup \mathrm{locs}(e_s) \cup \mathrm{locs}(e_c) \\
\mathrm{locs}(\mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ e_c\ \mathtt{end}) & = & \mathrm{locs}(v) \cup \mathrm{locs}(e_c) \\
\mathrm{locs}(\mathtt{write}(v)) & = & \mathrm{locs}(v)
\end{array}
$$

Constants

$$\frac{}{\Lambda; \Gamma \vdash_{\mathtt{S}} n : \mathtt{int}}$$

$$\frac{}{\Lambda; \Gamma \vdash_{\mathtt{S}} \mathtt{true} : \mathtt{bool}} \quad \frac{}{\Lambda; \Gamma \vdash_{\mathtt{S}} \mathtt{false} : \mathtt{bool}}$$

Locs, Vars

$$\frac{(\Lambda(l) = \tau)}{\Lambda; \Gamma \vdash_{\mathtt{S}} l : \tau\ \mathtt{mod}} \quad \frac{(\Gamma(x) = \tau)}{\Lambda; \Gamma \vdash_{\mathtt{S}} x : \tau}$$

Fun

$$\frac{\Lambda; \Gamma, f : \tau_1 \xrightarrow{\mathtt{S}} \tau_2, x : \tau_1 \vdash_{\mathtt{S}} e : \tau_2}{\Lambda; \Gamma \vdash_{\mathtt{S}} \mathtt{fun}_{\mathtt{S}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end} : (\tau_1 \xrightarrow{\mathtt{S}} \tau_2)}$$

$$\frac{\Lambda; \Gamma, f : \tau_1 \xrightarrow{\mathtt{C}} \tau_2, x : \tau_1 \vdash_{\mathtt{C}} e : \tau_2}{\Lambda; \Gamma \vdash_{\mathtt{S}} \mathtt{fun}_{\mathtt{C}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end} : (\tau_1 \xrightarrow{\mathtt{C}} \tau_2)}$$

Prim

$$\frac{\Lambda; \Gamma \vdash_{\mathtt{S}} v_i : \tau_i \quad (1 \le i \le n) \quad \vdash_o o : (\tau_1, \ldots, \tau_n)\ \tau}{\Lambda; \Gamma \vdash_{\mathtt{S}} o(v_1, \ldots, v_n) : \tau}$$

If

$$\frac{\Lambda; \Gamma \vdash_{\mathtt{S}} x : \mathtt{bool} \quad \Lambda; \Gamma \vdash_{\mathtt{S}} e_1 : \tau \quad \Lambda; \Gamma \vdash_{\mathtt{S}} e_2 : \tau}{\Lambda; \Gamma \vdash_{\mathtt{S}} \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

Apply

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

Let

$$\frac{\Lambda; \Gamma \vdash_{\mathtt{S}} e_1 : \tau_1 \quad \Lambda; \Gamma, x : \tau_1 \vdash_{\mathtt{S}} e_2 : \tau_2}{\Lambda; \Gamma \vdash_{\mathtt{S}} \mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} : \tau_2}$$

Mod

$$\frac{\Lambda; \Gamma \vdash_{\mathtt{C}} e : \tau}{\Lambda; \Gamma \vdash_{\mathtt{S}} \mathtt{mod}_{\tau}\ e : \tau\ \mathtt{mod}}$$

Figure 10: Typing of stable expressions.

5.2 Static Semantics

The AFL type system is inspired by the type theory of modal logic given by Pfenning and Davies' [12]. 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.

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

$$\text{If}\quad \frac{\Lambda;\Gamma \vdash_{\mathsf{S}} x : \mathtt{bool} \quad \Lambda;\Gamma \vdash_{\mathsf{C}} e_1 : \tau \quad \Lambda;\Gamma \vdash_{\mathsf{C}} e_2 : \tau}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{if}\, x\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 : \tau}$$

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

$$\text{Let}\quad \frac{\Lambda;\Gamma \vdash_{\mathsf{S}} e_1 : \tau_1 \quad \Lambda;\Gamma, x : \tau_1 \vdash_{\mathsf{C}} e_2 : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} : \tau_2}$$

$$\text{Read}\quad \frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_1 : \tau_1\ \mathtt{mod} \quad \Lambda;\Gamma, x : \tau_1 \vdash_{\mathsf{C}} e : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{read}\ v_1\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end} : \tau_2}$$

Figure 11: Typing of changeable expressions.

The judgement $\Lambda;\Gamma \vdash_{\mathsf{S}} e : \tau$ states that e is a well-formed stable expression of type τ, relative to Λ and Γ. The judgement $\Lambda;\Gamma \vdash_{\mathsf{C}} e : \tau$ 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 rules for deriving these judgements are given in Figures 10 and 11.

5.3 Dynamic Semantics

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

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

$$\text{Value} \qquad \sigma, v \Downarrow^{\mathsf{S}} v, \sigma, \varepsilon$$

$$\text{Op's} \qquad \frac{(v' = \mathtt{app}(o, (v_1, \ldots, v_n)))}{\sigma, o(v_1, \ldots, v_n) \Downarrow^{\mathsf{S}} v', \sigma, \varepsilon}$$

$$\text{If} \qquad \frac{\sigma, e_1 \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}{\sigma, \mathtt{if}\,\mathtt{true}\,\mathtt{then}\, e_1\, \mathtt{else}\, e_2 \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}$$

$$\frac{\sigma, e_2 \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}{\sigma, \mathtt{if}\,\mathtt{false}\,\mathtt{then}\, e_1\, \mathtt{else}\, e_2 \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}$$

$$\text{Apply} \qquad \frac{\begin{array}{c}(v_1 = \mathtt{fun}_{\mathsf{S}}\ f(x : \tau_2) : \tau\ \mathtt{is}\ e\ \mathtt{end}) \\ \sigma, [v_1/f, v_2/x]\ e \Downarrow^{\mathsf{S}} v', \sigma', \mathtt{T}_s\end{array}}{\sigma, \mathtt{apply}_{\mathsf{S}}(v_1, v_2) \Downarrow^{\mathsf{S}} v', \sigma', \mathtt{T}_s}$$

$$\text{Let} \qquad \frac{\begin{array}{lcl}\sigma, e_1 & \Downarrow^{\mathsf{S}} & v_1, \sigma', \mathtt{T}_s \\ \sigma', [v_1/x]e_2 & \Downarrow^{\mathsf{S}} & v_1, \sigma'', \mathtt{T}'_s\end{array}}{\sigma, \mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \Downarrow^{\mathsf{S}} v_2, \sigma'', (\mathtt{T}_s\ ;\ \mathtt{T}'_s)}$$

$$\text{Mod} \qquad \frac{\sigma[l \to \square], l \leftarrow e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}_c \quad (l \notin \mathrm{dom}(\sigma))}{\sigma, \mathtt{mod}_\tau\ e \Downarrow^{\mathsf{S}} l, \sigma', \langle \mathtt{T}_c \rangle_{l:\tau}}$$

Figure 12: Evaluation of stable expressions.

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.

The augmented 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 ADG 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 12. Most of the rules are standard for a store-passing semantics. For example, the `let` rule sequences

$$\text{Write} \qquad \sigma, l \leftarrow \mathtt{write}(v) \;\Downarrow^{\mathsf{C}}\; \sigma[l \leftarrow v], \mathtt{W}_\tau$$

$$\text{If} \qquad \frac{\sigma, l \leftarrow e_1 \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c}{\sigma, l \leftarrow \mathtt{if\,true\,then}\, e_1 \,\mathtt{else}\, e_2 \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c}$$

$$\frac{\sigma, l \leftarrow e_2 \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c}{\sigma, l \leftarrow \mathtt{if\,false\,then}\, e_1 \,\mathtt{else}\, e_2 \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c}$$

$$\text{Apply} \qquad \frac{\begin{array}{c}(v_1 = \mathtt{func}_{\mathsf{C}}\; f(x:\tau_1):\tau_2 \;\mathtt{is}\; e \;\mathtt{end}) \\ \sigma, l \leftarrow [v_1/f, v_2/x]\; e \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c\end{array}}{\sigma, l \leftarrow \mathtt{apply}_{\mathsf{C}}(v_1, v_2) \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c}$$

$$\text{Let} \qquad \frac{\begin{array}{lll}\sigma, e_1 & \Downarrow^{\mathsf{S}} & v_1, \sigma', \mathtt{T}_s \\ \sigma', l \leftarrow [v_1/x]e_2 & \Downarrow^{\mathsf{C}} & \sigma'', \mathtt{T}_c\end{array}}{\sigma, l \leftarrow \mathtt{let}\; x \;\mathtt{be}\; e_1 \;\mathtt{in}\; e_2 \;\mathtt{end} \;\Downarrow^{\mathsf{C}}\; \sigma'', (\mathtt{T}_s \;;\; \mathtt{T}_c)}$$

$$\text{Read} \qquad \frac{\sigma, l' \leftarrow [\sigma(l)/x]\; e \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c}{\sigma, l' \leftarrow \mathtt{read}\; l \;\mathtt{as}\; x \;\mathtt{in}\; e \;\mathtt{end} \;\Downarrow^{\mathsf{C}}\; \sigma', R_l^{x.e}(\mathtt{T}_c)}$$

Figure 13: Evaluation of changeable expressions.

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.

Changeable Evaluation. The evaluation rules for changeable expressions are given in Figure 13. The `let` rule is similar to the corresponding rule in stable mode, except that the bound expression, e_1, is evaluated in stable mode, whereas the body, e_2, 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.

6 Type Safety of AFL

The static semantics of AFL ensures these four 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.

The proof of type safety for AFL hinges on a type preservation theorem for the dynamic semantics. As may be expected, the preservation theorem ensures that the value of a well-typed stable expression is also well-typed (indeed, has the same type). In addition preservation ensures that evaluation of a changeable expression preserves the type of the store. The typing relation for stores ensures not only that the contents of locations are consistent with their type, but also that there are no cyclic dependencies among them. Thus preservation for AFL ensures that no cycles can arise during evaluation, which is consistent with pure functional programming.

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

Location typings may be partially ordered 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. $\leq_{\Lambda'} \restriction \mathrm{dom}(\Lambda) = \leq_\Lambda$.

It is easy to check that this is a partial ordering.

The *ordered extension*, $\Lambda[l'{:}\tau'{<}l]$, of a location typing Λ by assigning the type τ' to the location $l' \notin \mathrm{dom}(\Lambda)$ immediately before $l \in \mathrm{dom}(\Lambda)$ is the location typing Λ' 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 $l'' \leq_{\Lambda'} l'''$.

If $l \in \mathrm{dom}(\Lambda)$ and $l' \notin \mathrm{dom}(\Lambda)$, then $\Lambda \sqsubseteq \Lambda[l'{:}\tau'{<}l]$.

The *restriction*, $\Lambda \upharpoonright l$, of a location typing Λ to a location $l \in \mathrm{dom}(\Lambda)$, is 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. $\leq_{\Lambda'} = \leq_\Lambda \upharpoonright \mathrm{dom}(\Lambda')$.

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

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

1. $\mathrm{dom}(\sigma) = \mathrm{dom}(\Lambda)$.

2. for each $l \in \mathrm{def}(\sigma)$, $\Lambda \upharpoonright l \vdash_{\mathsf{S}} \sigma(l) : \Lambda(l)$.

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. In particular this ensures that the dependency relation among locations in the store is acyclic.

6.2 Trace Typing

The formulation of the type safety theorem requires a notion of typing for traces. The judgement $\Lambda, l_0 \vdash_{\mathsf{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$ 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 $\Lambda \vdash_{\mathsf{S}} \mathtt{T}_s\,\mathsf{ok}$ to mean that $\Lambda \vdash_{\mathsf{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$ for some Λ'.

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

The rules for deriving these judgements are given in Figure 14. The input location typing specifies the active locations, of which only those prior to the cursor are eligible as subjects of a read. 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 ouput 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.

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

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

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

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

Figure 14: Typing of Traces.

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

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 $\Lambda; \Gamma \vdash_{\mathsf{S}} v : \tau$.

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

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 new locations are allocated prior to l_0 in the ordering. 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) $\sigma, e \Downarrow^{\mathsf{S}} v, \sigma', T_s$,

   (b) $\sigma : \Lambda$,

   (c) $l_0 \in dom(\Lambda)$,

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

   (e) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} e : \tau$,

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

   (f) $\Lambda' \upharpoonright l_0 \vdash_{\mathsf{S}} v : \tau$,

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

   (h) $\Lambda, l_0 \vdash_{\mathsf{S}} T_s \rightsquigarrow \Lambda'$.

2. *If*

   (a) $\sigma, l_0 \leftarrow e \Downarrow^{\mathsf{C}} \sigma', T_c$,

   (b) $\sigma : \Lambda$,

   (c) $\Lambda(l_0) = \tau_0$,

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

   (e) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{C}} e : \tau_0$,

   then

   (f) $l_0 \in def(\sigma')$,

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

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

   (h) $\Lambda, l_0 \vdash_{\mathsf{C}} T_c : \tau_0 \rightsquigarrow \Lambda'$.

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

- Suppose that

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

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

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

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

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

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

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

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

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

Then we have

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

(2b′) $\sigma' : \Lambda'$. Since $\sigma : \Lambda$ by (1b), we have $\Lambda \upharpoonright l' \vdash_{\mathsf{S}} \sigma(l') : \Lambda(l')$ 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 $\Lambda' \upharpoonright l' \vdash_{\mathsf{S}} \sigma'(l') : \Lambda'(l')$, 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′) $\Lambda' \upharpoonright l \vdash_{\mathsf{C}} e : \tau$ since $\Lambda' \upharpoonright l = \Lambda \upharpoonright l_0$ and (1e(i)).

Therefore, by induction,

(2f′) $l \in \mathrm{def}(\sigma'')$;

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

(2g′) $\sigma'' : \Lambda''$;

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

Hence we have

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

(1g) $\sigma'' : \Lambda''$ by (2g′);

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

- Suppose that

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

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

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

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

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

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

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

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), $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} v : \tau$ by (2e(i)), and $\Lambda(l_0) = \tau$ by (2c).

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

- Suppose that

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

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

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

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

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

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

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

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

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

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

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 (2d).

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

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

It follows by induction that

(2f′) $l_0 \in \mathrm{def}(\sigma')$

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

(2g′) $\sigma' : \Lambda'$;

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

Therefore we have

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

(2g) $\sigma' : \Lambda'$ by (2g′);

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

- (a) $\Lambda \upharpoonright l_0, x{:}\Lambda(l) \vdash_{\mathsf{C}} e : \tau_0$ by (2e(ii));
- (b) $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau_0 \rightsquigarrow \Lambda'$ 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 $\Lambda \vdash_{\mathsf{S}} v : \tau$. Then

- *If $\tau = \texttt{int}$, then v is a numeric constant.*
- *If $\tau = \texttt{bool}$, then v is either $\texttt{true}$ or $\texttt{false}$.*
- *If $\tau = \tau_1 \xrightarrow{\mathsf{C}} \tau_2$, then $v = \texttt{fun}_{\mathsf{C}}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e\ \texttt{end}$ with $\Lambda; f{:}\tau_1 \xrightarrow{\mathsf{C}} \tau_2, x{:}\tau_1 \vdash_{\mathsf{C}} e : \tau_2$.*
- *If $\tau = \tau_1 \xrightarrow{\mathsf{S}} \tau_2$, then $v = \texttt{fun}_{\mathsf{S}}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e\ \texttt{end}$ with $\Lambda; f{:}\tau_1 \xrightarrow{\mathsf{S}} \tau_2, x{:}\tau_1 \vdash_{\mathsf{S}} e : \tau_2$.*
- *If $\tau = \tau'\ \texttt{mod}$, 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 is Sound

We formalize the notion of an input change and present a formal version of the change-propagation algorithm. Using this formal framework, we prove the type safety and the correctness of the change-propagation algorithm.

Changing the Input. We represent an input change with a *difference store*. A difference store is a finite mapping assigning values to locations. Unlike a store, a difference store may contain "dangling" locations that are not defined within the store. The process of modifying a store with a difference store is defined as follows.

Definition 9 (Store Modification)
Let $\sigma : \Lambda$ be a store and let δ be a difference store. The modification of σ by δ, denoted by $\sigma \oplus \delta$, is a well-typed store $\sigma' : \Lambda'$, $\sigma' = \sigma \oplus \delta$, where

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

Note that the definition requires the result store to be well typed. The store modification yields an input change when it is applied to an input store.

$$\sigma, \varepsilon, \mathtt{C} \ \Downarrow^{\mathsf{p}}_{\mathsf{s}} \ \sigma, \varepsilon, \mathtt{C}$$

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

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

Write
$$\sigma, l \leftarrow \mathtt{W}_\tau, \mathtt{C} \ \Downarrow^{\mathsf{p}}_{\mathsf{s}} \ \sigma, \mathtt{W}_\tau, \mathtt{C}$$

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

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

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

Figure 15: Change propagation rules (stable and changeable).

Change Propagation Algorithm. We present a formal version of the change-propagation algorithm, which is informally described in Section 4. In the rest of this section, we will use the term change-propagation algorithm to refer to this formal algorithm.

The change-propagation algorithm takes a modified store, a trace obtained by evaluating an AFL program with respect to the original store, and a set of input locations that are changed by the store modification, called the *changed set*. The algorithm scans the trace as it seeks for reads of changed locations. When such a read is found, the associated expression is re-evaluated with the new value to obtain a revised trace and store. Furthermore, the target of a re-evaluated read is added to the changed set, because re-evaluation may change its value. Thus, the order in which the reads are re-evaluated is important. The change-propagation algorithm scans the trace in the order that it was originally generated. This ensures that the trace is scanned only once and is done by establishing a correspondence between the change-propagation rule that handles a trace and the AFL rule that generates that trace.

Formally, the change propagation algorithm is given by these two judgements:

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

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

The rules defining the change-propagation judgements are given in Figure 15. Given a trace, change propagation mimics the evaluation rule of AFL that originally generated that trace. To stress this correspondence, each change-propagation 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 two different actions depending on whether this location is in the changed set or not. If the location has changed (is in the changed set), then the expression is re-evaluated with the new value of location. This re-evaluation yields a revised store and a new trace. The new trace "repairs" the original trace by replacing the trace of the read. Also, the target location is added to the changed set because it may now have a different value. Finally, the "repaired" trace, the revised store, and the revised changed set is yielded. If the read location has not been changed (is not in the changed set), then there is no need to re-evaluate this read and change-propagation continues by scanning the rest of the trace. This is because a re-evaluation would generate the same effects to the store and to the trace as done by the initial evaluation. Since these effects are already present in the store and the trace, this read could safely be skipped.

Note that the purely functional change-propagation algorithm presented here scans the whole trace. Therefore, a direct implementation of this algorithm will run in time linear in the size of the trace. On the other hand, the change-propagation algorithm revises the trace by only replacing the changeable trace 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 all the rest of the trace. This is indeed how the ML implementation performs change propagation using an augmented dependency graph as described in Section 4.

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 $\mathit{def}(\sigma) = \mathit{dom}(\sigma)$.

1. *If*

    (a) $\sigma, \mathtt{T}_s, \mathtt{C} \Downarrow^{\mathsf{p}}_{\mathsf{s}} \sigma', \mathtt{T}'_s, \mathtt{C}'$,

    (b) $\sigma : \Lambda$,

    (c) $l_0 \in \mathit{dom}(\Lambda)$,

    (d) $\Lambda, l_0 \vdash_{\mathsf{s}} \mathtt{T}_s\,\mathsf{ok}$, *and*

    (e) $\mathtt{C} \subseteq \mathit{dom}(\Lambda)$,

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

(f) $\sigma' : \Lambda'$,

(g) $\Lambda, l_0 \vdash_{\mathsf{S}} T'_s \rightsquigarrow \Lambda'$,

(h) $\mathtt{C}' \subseteq dom(\Lambda)$.

2. *If*

(a) $\sigma, l_0 \leftarrow T_c, \mathtt{C} \Downarrow^{\mathsf{P}}_{\mathsf{C}} \sigma', T'_c, \mathtt{C}'$,

(b) $\sigma : \Lambda$,

(c) $\Lambda(l_0) = \tau_0$,

(d) $\Lambda, l_0 \vdash_{\mathsf{C}} T_c : \tau_0$, *and*

(e) $\mathtt{C} \subseteq dom(\Lambda)$,

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

(f) $\sigma' : \Lambda'$,

(g) $\Lambda, l_0 \vdash_{\mathsf{C}} T'_c : \tau_0 \rightsquigarrow \Lambda'$, *and*

(h) $\mathtt{C}' \subseteq dom(\Lambda)$.

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 $l \in \mathtt{C}$ and

(2a) $\sigma, l_0 \leftarrow R^{x.e}_l(\mathtt{T}_c), \mathtt{C} \Downarrow^{\mathsf{P}}_{\mathsf{C}} \sigma', R^{x.e}_l(\mathtt{T}'_c), \mathtt{C} \cup \{l_0\}$;

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

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

(2d) $\Lambda, l_0 \vdash_{\mathsf{C}} R^{x.e}_l(\mathtt{T}_c) : \tau_0$;

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

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

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

(2b(i) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} \sigma(l) : \Lambda(l)$, by (2b);

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

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

(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);

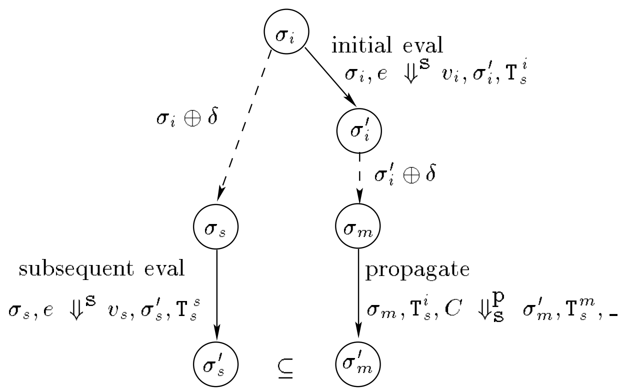

Figure 16: Change propagation simulates a complete re-evaluation.

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

(2e$'$) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{C}} [\sigma(x)/l]e : \tau_0$ 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) $\Lambda, l_0 \vdash_{\mathsf{C}} R^{x.e}_l(\mathtt{T}'_c) : \tau_0$ 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 only re-evaluating the affected sub-expressions of an AFL program. Here we show that change propagation yields the same output and the trace as a complete re-evaluation and thus is correct.

Figure 16 illustrates this simulation process. First, we evaluate a program e with respect to an initial store σ_i; this is called the *initial evaluation*. We assume that a program is a stable expression and the initial store contains no reserved locations. The initial evaluation yields a value v_i, an extended store σ'_i, and a trace $\mathtt{T}^i_s$. Then, we 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, we first apply the modifications δ to σ_i', to obtain a new store σ_m as $\sigma_m = \sigma_i' \oplus \delta$. We then perform change propagation with respect to σ_m, using the trace of the initial evaluation, and the set of changed locations $C = \mathrm{dom}(\sigma_i') \cap \mathrm{dom}(\delta)$. As a result, we obtain a revised trace and store σ_m' and a revised trace $\mathtt{T}_s^m$. For the change-propagation to work properly, we require that δ changes 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 $\mathtt{T}_s^s$ respectively, to those obtained by the change propagation, σ_m' and $\mathtt{T}_s^m$ respectively. Since these two evaluations are independent, we do not expect the locations generated in them to match. In fact, the two traces and the stores can contain different locations. On the other hand, this is not a problem because locations themselves are not visible to the user. To capture this, we introduce an equivalence relation for stores and traces that disregards locations (names) via a partial bijection between locations. 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 application of a partial bijection, B, to a location, l, is denoted as $B(l)$ and defined as

$$B(l) = \begin{cases} l' & if\ (l,l') \in B \\ l & otherwise \end{cases}$$

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 each location l with $B(l)$. The formal definitions for these are given below.

Definition 12 (Application of a partial bijection to an expression)
The application of a partial bijection B to an expression e yields an expression e' which is obtained from e by substituting each location l with $B(l)$ as shown in Figure 17.

Definition 13 (Application of partial bijection to a store)
The application of a partial bijection B to a store σ, denoted $B[\sigma]$, yields another store σ' defined as $\sigma' = B[\sigma] = \{\, (B(l), B[\sigma[l]]) \mid l \in dom(\sigma) \,\}$.

Definition 14 (Application of a partial bijection to a trace)
The application of a partial bijection to a trace is defined as follows.

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

- **Values**

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

- **Stable Expressions**

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

- **Changeable Expressions**

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

- **Expression with a destination**

$$B[l \leftarrow e] = B(l) \leftarrow B[e]$$

Figure 17: Application of a partial bijection B to an expression.

Before we show the lemma that the correctness theorem relies on, we give a few definitions.

Given a partial bijection B and two stores, we define the set of changed locations in σ with respect to the store σ' and B as follows.

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

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

Definition 16 (Store Containment)
We say that a store,σ, is contained *in another σ' (denoted $\sigma \sqsubseteq \sigma'$), if*

1. $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$, and

2. $\forall l, l \in \mathrm{def}(\sigma), \sigma[l] = \sigma'[l]$.

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

Figure 18: The defined domain and the domain of a trace.

Note that, the definition of store containment requires only the values of the defined locations to be preserved. The reserved locations of the contained store, however, can be assigned to any value by the containing store.

We define the *domain* and the *defined domain* of a trace $\mathtt{T}$, written $\text{dom}(\mathtt{T})$ and $\text{def}(\mathtt{T})$, respectively, as shown in Figure 18.

Our goal is to prove that an initial evaluation of an expression followed by change propagation gives the same result (up to a partial bijection) as the corresponding subsequent evaluation of the same expression. We do this by induction on evaluation. To make the induction work, we first prove a more generalized lemma. This lemma deals with expression that are equal up to a partial bijection. The correctness proof, which is only concerned with *equal* expressions is a special case of the lemma.

In the proof of the change propagation lemma, we assume for brevity that a changeable expression evaluates to a different value when re-evaluated, *i.e.*, the value of the destination location changes. This assumption causes no loss of generality, and can be eliminated by additional machinery to enable comparison of the old and the new values of the destination location.

Lemma 17 (Change Propagation)
Let e_s and e_c be stable and changeable expressions respectively, σ_i and σ_s be two stores, and B be a partial bijection from $dom(\sigma_i)$ to $dom(\sigma_s)$. The following hold:

- *If*

$$\sigma_i, l \leftarrow e_c \ \Downarrow^{\mathsf{C}} \ \sigma'_i, \mathtt{T}_c^i, \text{and}$$
$$\sigma_s, B[l \leftarrow e_c] \ \Downarrow^{\mathsf{C}} \ \sigma'_s, \mathtt{T}_c^s,$$

  then for any store σ_m satisfying the following

  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

$$\sigma_m, l \leftarrow \mathtt{T}_c^i, \mathtt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow^{\mathsf{P}}_{\mathsf{C}} \ \sigma'_m, \mathtt{T}_c^m, \mathsf{C}, \text{where}$$

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. $\mathtt{C} = \mathtt{changed}(B', \sigma'_i, \sigma'_s)$.

- *If*

$$\sigma_i, e_s \Downarrow^{\mathsf{S}} v_i, \sigma'_i, \mathtt{T}_s^i, \text{and}$$
$$\sigma_s, B[e_s] \Downarrow^{\mathsf{S}} v'_i, \sigma'_s, \mathtt{T}_s^s,$$

then for any store σ_m satisfying the following

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

$$\sigma_m, \mathtt{T}_s^i, \mathtt{changed}(B, \sigma_i, \sigma_s) \Downarrow^{\mathsf{p}}_{\mathsf{S}} \sigma'_m, \mathtt{T}_s^m, \mathtt{C}, where$$

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

Proof: The proof is by simultaneous induction on the 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`.

For ease of reference, we refer to the three conditions that are satisfied by the modified store σ_m as the *modified-store properties.* These properties are:

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

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

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

$$\sigma_m, l \leftarrow \mathtt{W}_\tau, \mathtt{changed}(B, \sigma_i, \sigma_s) \Downarrow^{\mathsf{p}}_{\mathsf{C}} \sigma_m, \mathtt{W}_\tau, \mathtt{changed}(B, \sigma_i, \sigma_s).$$

Now we show that the partial bijection B satisfies the following properties.

1. $B \sqsupseteq B$
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. $\mathtt{changed}(B, \sigma_i, \sigma_s) = \mathtt{changed}(B, \sigma_i[l \to v], \sigma_s[B(l) \to B[v]])$, by definition.

Thus we can pick $B' = B$.

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

$$\frac{(1.1)\quad \sigma_i, l \leftarrow [v/x, \mathtt{fun}_\mathtt{C}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end}/f]\ e \ \Downarrow^{\mathtt{C}}\ \sigma'_i, \mathtt{T}^i_c}{(1.2)\quad \sigma_i, l \leftarrow \mathtt{apply}_\mathtt{C}(\mathtt{fun}_\mathtt{C}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end}, v) \ \Downarrow^{\mathtt{C}}\ \sigma'_i, \mathtt{T}^i_c}$$

$$\frac{(2.1)\quad \sigma_s, B(l) \leftarrow [B[v]/x, B[\mathtt{fun}_\mathtt{C}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end}]/f]\ e] \ \Downarrow^{\mathtt{C}}\ \sigma'_s, \mathtt{T}^s_c}{(2.2)\quad \sigma_s, B[l \leftarrow \mathtt{apply}_\mathtt{C}(\mathtt{fun}_\mathtt{C}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end}, v)] \ \Downarrow^{\mathtt{C}}\ \sigma'_s, \mathtt{T}^s_c}$$

Consider evaluations (1.1) and (2.1) and a store σ_m that satisfies the modified-store properties. By induction we have a partial bijection B_0 and

$$\sigma_m, l \leftarrow \mathtt{T}^i_c, \mathtt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow^{\mathtt{p}}_{\mathtt{C}}\ \sigma_m, \mathtt{T}^m_c, \mathtt{C},$$

where

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

Since, (1.2) and (2.2) return the same trace and store as (1.1) and (2.1) respectively, we pick $B' = B_0$.

- **Let:**

$$\frac{\begin{array}{lll}(1.1) & \sigma_i, e & \Downarrow^{\mathtt{S}}\ v_i, \sigma'_i, \mathtt{T}^i_s \\ (1.2) & \sigma'_i, l \leftarrow [v_i/x]e' & \Downarrow^{\mathtt{C}}\ \sigma''_i, \mathtt{T}^i_c\end{array}}{(1.3)\quad \sigma_i, l \leftarrow \mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ e'\ \mathtt{end} \ \Downarrow^{\mathtt{C}}\ \sigma''_i, (\mathtt{T}^i_s\ ;\ \mathtt{T}^i_c)}$$

$$\frac{\begin{array}{lll}(2.1) & \sigma_s, B[e] & \Downarrow^{\mathtt{S}}\ v_s, \sigma'_s, \mathtt{T}^s_s \\ (2.2) & \sigma'_s, B(l) \leftarrow [v_s/x]B[e'] & \Downarrow^{\mathtt{C}}\ \sigma''_s, \mathtt{T}^s_c\end{array}}{(2.3)\quad \sigma_s, B[l \leftarrow \mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ e'\ \mathtt{end}] \ \Downarrow^{\mathtt{C}}\ \sigma''_s, (\mathtt{T}^s_s\ ;\ \mathtt{T}^s_c)}$$

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 $\mathtt{T}_s^i \; ; \; \mathtt{T}_c^i$.

$$\frac{\begin{array}{lll} (3.1) & \sigma_m, \mathtt{T}_s^i, \mathtt{changed}(B, \sigma_i, \sigma_s) & \Downarrow^{\mathsf{p}}_{\mathsf{s}} \; \sigma'_m, \mathtt{T}_s^m, \mathtt{C} \\ (3.2) & \sigma'_m, l \leftarrow \mathtt{T}_c^i, \mathtt{C} & \Downarrow^{\mathsf{p}}_{\mathsf{c}} \; \sigma''_m, \mathtt{T}_c^m, \mathtt{C}' \end{array}}{(3.3) \quad \sigma_m, l \leftarrow (\mathtt{T}_s^i \; ; \; \mathtt{T}_c^i), \mathtt{changed}(B, \sigma_i, \sigma_s) \Downarrow^{\mathsf{p}}_{\mathsf{c}} \sigma''_m, (\mathtt{T}_s^m \; ; \; \mathtt{T}_c^m), \mathtt{C}'}$$

We apply the induction hypothesis on (1.1) (2.1) and (3.1) 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. $\mathtt{C} = \mathtt{changed}(B_0, \sigma'_i, \sigma'_s)$.

Using these properties, we now show that we can apply the induction hypothesis on (1.2) and (2.2) 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 $B[e'] = B_0[e']$, we observe that $\mathrm{dom}(B_0) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_s^m)$ and $\mathrm{def}(\mathtt{T}_s^m) \cap \mathrm{locs}(e) = \emptyset$, because $\mathrm{locs}(e) \subseteq \mathrm{dom}(\sigma_i)$ and $\mathrm{locs}(e) \subseteq \mathrm{dom}(\sigma_m)$.
- $\mathtt{C} = \mathtt{changed}(B_0, \sigma'_i, \sigma''_i)$. 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 (3.1) 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 (1.2) (2.2) 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′. $\mathtt{C}' = \mathtt{changed}(B_1, \sigma_i'', \sigma_s'')$.

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

5″. $\mathtt{C}' = \mathtt{changed}(B_1, \sigma_i'', \sigma_s'')$,
This follows by Property 5′.

Thus we pick $B' = B_1$.

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

$$\frac{1.1 \qquad \sigma_i, l' \leftarrow [\sigma_i[l]/x]e \ \Downarrow^{\mathtt{C}} \ \sigma_i', \mathtt{T}_c^i}{1.2 \sigma_i, l' \leftarrow \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end} \ \Downarrow^{\mathtt{C}} \ \sigma_i', R_l^{x.e}(\mathtt{T}_c^i)}$$

$$\frac{2.1 \qquad \sigma_s, B[l'] \leftarrow [\sigma_s[B(l)]/x]B[e] \ \Downarrow^{\mathtt{C}} \ \sigma_s', \mathtt{T}_c^s}{2.2 \sigma_s, B[l' \leftarrow \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ e\ \mathtt{end}] \ \Downarrow^{\mathtt{C}} \ \sigma_s', R_{B(l)}^{x.B[e]}(\mathtt{T}_c^s)}$$

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 $l \notin \mathtt{C}$ and we have:

$$\frac{3.1 \qquad \sigma_m, l' \leftarrow \mathtt{T}_c^i, \mathtt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow_{\mathtt{C}}^{\mathtt{P}} \ \sigma_m', \mathtt{T}_c^m, \mathtt{C}}{3.2 \sigma_m, l' \leftarrow R_l^{x.e}(\mathtt{T}_c^i), \mathtt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow_{\mathtt{C}}^{\mathtt{P}} \ \sigma_m', R_l^{x.e}(\mathtt{T}_c^m), \mathtt{C}} \ (l \notin \mathtt{C})$$

In this case, we apply the induction hypothesis on (1.1) (2.1) and (3.1) with the partial bijection B. By induction, we obtain a partial bijection B_0 and

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

5. $\mathtt{C} = \mathtt{changed}(B_0, \sigma_i', \sigma_s')$.

Furthermore, the following hold for B_0,

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

2. $B_0[R_l^{x.e}(\mathtt{T}_c^m)] = R_{B(l)}^{x.B[e]}(\mathtt{T}_c^s)$.
   We have $B_0[R_l^{x.e}(\mathtt{T}_c^m)] = R_{B_0(l)}^{x.B_0[e]}(B_0[\mathtt{T}_c^m]) = R_{B_0(l)}^{x.B_0[e]}(\mathtt{T}_c^s)$, 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) $l \notin \mathrm{def}(\mathtt{T}_c^m)$ and thus $B(l) = B_0(l)$, and
   (b) $\forall l, l \in \mathrm{locs}(e)$ we have $l \in \mathrm{def}(\sigma_m)$ and thus $l \notin \mathrm{def}(\mathtt{T}_c^m)$, 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 $l \in \mathtt{C}$ and the read $R_l^{x.e}$ is re-evaluated.

$$\frac{(4.1)\qquad \sigma_m, l' \leftarrow [\sigma_m[l]/x]e \ \Downarrow^{\mathtt{C}}\ \sigma_m', \mathtt{T}_c^m}{(4.2)\qquad \sigma_m, l' \leftarrow R_l^{x.e}(\mathtt{T}_c), \mathtt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow_{\mathtt{C}}^{\mathtt{P}}\ \sigma_m', R_l^{x.e}(\mathtt{T}_c^m), \mathtt{changed}(B, \sigma_i, \sigma_s) \cup \{l'\}} \ (l \in \mathtt{C})$$

Since $B[\sigma_m] \sqsupseteq \sigma_s$, the evaluation in (4.1) is identical to the evaluation in (2.1) and thus, there is a bijection $B_1 \supseteq B$ such that $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$ and $\mathrm{dom}(B_1) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_c^m)$. Thus we have

1. $B_1 \supseteq B$,
2. $\mathrm{dom}(B_1) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_c^m)$,
3. $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$,
4. $\mathtt{changed}(B, \sigma_i, \sigma_s) \cup \{l\} = \mathtt{changed}(B_1, \sigma_i', \sigma_s')$
   To show this, observe that
   (a) $\mathrm{dom}(\sigma_i') \cap \mathrm{def}(\mathtt{T}_c^m) = \emptyset$, because $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma_i')$.
   (b) $\mathtt{changed}(B_1, \sigma_i', \sigma_s') = \mathtt{changed}(B, \sigma_i', \sigma_s')$, because $\mathrm{dom}(B_1) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_c^m)$.
   (c) $\mathtt{changed}(B, \sigma_i', \sigma_s') = \mathtt{changed}(B, \sigma_i, \sigma_s) \cup \{l'\}$, because $\mathrm{dom}(B) \subseteq \mathrm{dom}(\sigma_i)$. (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

$$\sigma_m, \varepsilon, \texttt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow^{\mathsf{P}}_{\mathsf{S}} \ \sigma_m, \varepsilon, \texttt{changed}(B, \sigma_i, \sigma_s)$$

where

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

Thus pick $B' = B$.

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

$$\frac{(1.1) \qquad \sigma_i[l_i \to \square], l_i \leftarrow e \ \Downarrow^{\mathsf{C}} \ \sigma_i', \mathtt{T}_c^i}{(1.2) \qquad \sigma_i, \mathtt{mod}_\tau \ e \ \Downarrow^{\mathsf{S}} \ l_i, \sigma_i', \langle \mathtt{T}_c^i \rangle_{l_i:\tau}} \ l_i \notin \mathrm{dom}(\sigma_i)$$

$$\frac{(2.1) \qquad \sigma_s[l_s \to \square], l_s \leftarrow B[e] \ \Downarrow^{\mathsf{C}} \ \sigma_s', \mathtt{T}_c^s}{(2.2) \qquad \sigma_s, B[\mathtt{mod}_\tau \ e] \ \Downarrow^{\mathsf{S}} \ l_s, \sigma_s', \langle \mathtt{T}_c^s \rangle_{l_s:\tau}} \ l_s \notin \mathrm{dom}(\sigma_s).$$

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

$$\frac{(3.1) \qquad \sigma_m, l_i \leftarrow \mathtt{T}_c^i, \texttt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow^{\mathsf{P}}_{\mathsf{C}} \ \sigma_m', \mathtt{T}_c^m, \mathtt{C}}{(3.2) \qquad \sigma_m, \langle \mathtt{T}_c^i \rangle_{l_i:\tau}, \texttt{changed}(B, \sigma_i, \sigma_s) \ \Downarrow^{\mathsf{P}}_{\mathsf{S}} \ \sigma_m', \langle \mathtt{T}_c^m \rangle_{l_i:\tau}, \mathtt{C}}$$

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 (1.1), (2.1) 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. $\mathtt{C} = \mathtt{changed}(B_1, \sigma'_s, \sigma'_i)$.

Furthermore, B_1 satisfies that

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.

■

We can now prove the correctness theorem for the change-propagation algorithm. In the theorem, the reason that the store σ'_m is a super set of σ'_s is that σ'_m contains remnant locations from the initial evaluation, whereas σ'_s does not.

Theorem 18 (Correctness)
Let e be a program, σ_i 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 16. If

1. $\sigma_i, e \Downarrow^{\mathsf{S}} v_i, \sigma'_i, T_s^i$, *(initial evaluation)*

2. $\sigma_s, e \Downarrow^{\mathsf{S}} v_s, \sigma'_s, T_s^s$, *(subsequent evaluation)*

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

then the following holds:

1. $\sigma_m, T_s^i, (dom(\sigma'_i) \cap dom(\delta)) \Downarrow^{\mathsf{P}}_{\mathsf{S}} \sigma'_m, T_s^m, \_,$

2. *there is a partial bijection $\mathcal{B}$ such that*

   (a) $\mathcal{B}[v_i] = v_s$,
   (b) $\mathcal{B}[T_s^m] = T_s^s$,
   (c) $\mathcal{B}[\sigma'_m] \supseteq \sigma'_s$.

Proof: The proof is by an application of Lemma 17. To apply the lemma we define the partial bijection B (of the lemma) to be the identity function in the $\mathrm{dom}(\sigma_i)$. Since $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma_i)$, this is valid. We also show the following for σ_m

1. $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma'_i)$.
   This follows by the definition of the store modification.

2. $\forall l, l \in (\mathrm{def}(\sigma_i') - \mathrm{def}(\sigma_i)), \sigma_m[l] = \sigma_i'[l]$.
Since $\mathrm{dom}(\sigma_i') \cap \mathrm{dom}(\delta) \subseteq \mathrm{dom}(\sigma_i)$, the store modification only changes values of the locations in $\mathrm{dom}(\sigma_i)$, which is equal to $\mathrm{def}(\sigma_i)$.

3. $B[\sigma_m] \sqsupseteq \sigma_s$
Since B is the identity, all we have to show is that $\sigma_m \sqsupseteq \sigma_s$. This holds because $\sigma_s = \sigma_i \oplus \delta$, and $\sigma_m = \sigma_i' \oplus \delta$ and $\sigma_i' \supseteq \sigma_i$.

The theorem follows by taking the partial bijection $\mathcal{B}$ required by the theorem equal to the partial bijection B' yielded by Lemma 17.

■

8 Discussion

Variants. In the process of developing the mechanisms presented in this paper we considered several variants. Here we mention a few of them. 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. We also considered a variant of our mechanism in which the `mod`, `read`, and `write` are combined into a single operation. This operation reads a modifiable, evaluates an expression with the value of the modifiable, and writes the result into a new modifiable. In the ML library the operation can be defined as follows.

```
function modrw(x : 'a mod, f : 'a -> 'b) : 'b =
   mod(fn d => read x (fn x' => write(d, f(x'))))
```

This operation, along with another that does two reads, were sufficient to express many of the examples we were working with. The operations, however, are not expressive enough for many other examples, and in particular for Quicksort. In practice it would worthwhile including these two operations in a comprehensive adaptive library since implementing them directly would be more efficient than the composition given above.

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 [6, 5, 7].

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, or any 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. In fact, re-evaluations of a function can be counted by simply inserting a counter at the start of the function. 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.

Function Caching. As mentioned in the related work section, it might be useful to add function caching to our framework. We believe this is a promising extension, but should note that it is not trivial to incorporate this feature. The problem is that function caching and modifiables interact in subtle ways—function caching requires purely functional code, but our framework involves side-effects in its implementation.

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 [2]. 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. We also would like to thank Mihai Budiu, Aleks Nanevski, and the anonymous referees for their comments on the earlier drafts of this paper.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the Twenty-ninth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2002.

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

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

[10] Yanhong A. Liu, Scott Stoller, and Tim Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on POPL*, pages 157–170, January 1996.

[11] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–30, February 1995.

[12] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[13] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual Symposium on POPL*, pages 315–328, January 1989.

[14] William Pugh. *Incremental computation via function caching*. PhD thesis, Department of Computer Science, Cornell University, August 1987.

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

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

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

[18] 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.