Under consideration for publication in J. Functional Programming

A consistent semantics of self-adjusting computation

UMUT A. ACAR
Carnegie Mellon University

MATTHIAS BLUME
Google

JACOB DONHAM
Twitter

Abstract

This paper presents a semantics of self-adjusting computation and proves that the semantics is correct and consistent. The semantics introduces memoizing change propagation, which enhances change propagation with the classic idea of memoization to enable reuse of computations even when memory is mutated via side effects. During evaluation, computation re-use via memoization triggers a change propagation algorithm that adapts the reused computation to the memory mutations (side-effects) that took place since the creation of the computation. Since the semantics includes both memoization and change-propagation, it involves both non-determinism (due to memoization) and mutation (due to change propagation). Our consistency theorem states that the non-determinism is not harmful: any two evaluations of the same program starting at the same state yield the same result. Our correctness theorem states that mutation is not harmful: self-adjusting programs are compatible with purely functional programming. We formalize the semantics and its meta-theory in the LF logical framework and machine check our proofs using Twelf.

1 Introduction

Many applications operate on data that change over time: compilers respond to changes to source code by recompiling as necessary, robots must interact with the physical world as it naturally changes over time, and scientific simulations compute with objects whose properties change over time, e.g., as they interact. Self-adjusting computation offers techniques for designing, analyzing, and implementing programs that respond to changes to their data automatically and efficiently. Previous work on self-adjusting computation developed programming languages for implementing such programs (e.g. [Ley-Wild et al. 2008b; Hammer et al. 2009]), cost semantics and algorithmic analysis techniques for designing and analyzing them (e.g. [Ley-Wild et al. 2008a; Acar 2005]). Applications of self-adjusting computation to a broad range of problems [Acar et al. 2009] have also been considered, including diverse areas such as computational geometry (e.g. [Acar et al. 2010]), machine learning (e.g., [Sümer et al. 2011]), and large-scale data sets [Bhatotia et al. 2011]. The results on applications show that the approach can help achieve asymptotically optimal updates and even help solve open problems.

Research on self-adjusting computation started with the invention of dynamic dependency graphs and a change-propagation algorithm for updating them [Acar et al. 2006]. The idea behind these techniques is to represent evaluation of purely functional programs with a *dynamic dependency graph* or a *DDG* and use a change propagation algorithm to update the graph and the output of the evaluation when the data changes. To perform correct and efficient updates, *change propagation* identifies the nodes of the DDG that depend on the changed data, re-evaluates the computations performed by the nodes—the nodes contains fragments of code—-and restructures the DDG by inserting and deleting dependency edges as needed. Change propagation is an imperative algorithm: it mutates a selected subset of memory cells in order to mimic a re-evaluation of the whole program. While change propagation can indeed update computations efficiently in certain cases, as we discuss in Section 2, in many cases it falls short of the goal re-using computations optimally.

In this paper, we enhance change propagation with the classical idea of memoization for improved computation re-use. The idea behind memoization [Michie 1968] is to remember the results of function calls (by recording a map from the arguments to the results) and reuse them instead of repeating calls. For memoization to be correct, it is critical for memoized functions to be pure, because otherwise the mapping from the arguments of functions to their results can change as a result of mutations to memory. As we describe in Section 2, the form of computation re-use provided by memoization turns out to be essentially dual to change propagation, suggesting that if applied simultaneously, they can lead to a broadly effective computation re-use mechanism. Using both techniques in the same computation, however, is challenging, because memoization can re-use only purely functional computations whereas dynamic dependency graphs and change propagation rely heavily on imperative updates.

We overcome this challenge by introducing *memoizing change propagation*, which integrates change propagation and memoization by offering a mechanism for reusing impure, imperative computations. The key idea behind memoizing change propagation is to reuse computations themselves instead of just results of computations. More specifically, as in classical memoization, we use memoized function calls (or expressions more generally) to trigger computation reuse. Instead of remembering and re-using results of function calls as in classical memoization, however, we remember and reuse their dynamic dependency graphs. This allows us to re-use a function call even when its arguments have changed (due to mutations) by applying recursively the change propagation algorithm on the re-used dynamic dependency graph.

To study the expressiveness and soundness of memoizing change propagation, we extend the *adaptive functional language* AFL [Acar et al. 2006], which supports change propagation, with a construct for memoization. We call this language AML (Section 3). The dynamic semantics of AML is store-based. Mutation to the store between successive evaluations models incremental changes to the input. The evaluation of an AML program also allocates store locations and updates existing locations. We model memoization as a non-deterministic oracle: a memoized expression is evaluated by first consulting the *memo-oracle*, which non-deterministically returns either a *miss* or a *hit*. In evaluation, a hit returns a trace of the evaluation of the memoized expression, which is recursively adapted to mutations by performing a change propagation on the returned trace. By using a non-deterministic oracle for modeling memoization, we ensure that the semantics and thus our results remain applicable irrespective of how memoization is realized, e.g., specifics of which computations are maintained in cache do not matter.

We prove two main theorems stating that the semantics is *consistent* and *correct* (Section 4). The consistency theorem proves that the non-determinism (due to memoization) is harmless by showing that any two evaluations of the same program in the same store yield the same result. Specifically, this implies that an execution in which no computations were reused (nondeterministic oracle always "missing") and any computation where results were reused (nondeterministic oracle sometimes "hitting") are equivalent. The correctness theorem in turn states a correspondence between self-adjusting computation and purely functional programming by showing that evaluation returns (observationally) the same value as a purely functional evaluation. Taken together the two theorems imply that self-adjusting computation is congruous with purely functional programming. Our proofs do not make any assumptions about typing. Our results therefore apply in both typed and untyped settings.

The proofs for the correctness and consistency theorems (Section 4) are made challenging because the semantics consists of a complex set of judgments (where change propagation and ordinary evaluation are mutually recursive), and because the semantics involves mutation and two kinds of non-determinism: non-determinism in memory allocation, and non-determinism due to memoization. Due to mutation, we are required to prove that evaluation preserves certain well-formedness properties (e.g., absence of cycles and dangling pointers). Due to non-deterministic memory allocation, we cannot compare the results from different evaluations directly. Instead, we compare values structurally by comparing the contents of locations. To address non-determinism due to memoization, we allow evaluation to recycle existing memory locations. Based on these techniques, we first prove that memoization is harmless: for any evaluation there exists a memoization-free counterpart that yields the same result without reusing any computations. Based on structural equality, we then show that memoization-free evaluations and fully deterministic evaluations are equivalent. These proof techniques may be of independent interest.

To increase confidence in our results, we encoded the syntax and semantics of AML and its meta-theory in the LF logical framework [Harper et al. 1993] and machine-checked the proofs using Twelf [Pfenning and Schürmann 1999] (Section 6). The Twelf formalization consist of 7800 lines of code. The Twelf code is fully foundational: it encodes all background structures required by the proof and proves all lemmas from first principles. The full Twelf code for the proofs (as a tar archive) can be found at the url `http://www.umut-acar.org/publications/jfp2013-twelf-proof.tar`. The Twelf proof is also reachable via the first author's web page at `http://www.umut-acar.org`.

We note that checking the proofs in Twelf was not a merely an encoding exercise. In fact, our initial attempts at producing a paper-and-pencil proof failed. The process of creating and checking the proof mechanically in Twelf allowed us to come up with the proof, while also helping us simplify the rule systems and generalize the proof to untyped languages. We therefore feel that the use of Twelf was critical to this result.

Since the semantics models memoization as a non-deterministic oracle, and since it does not specify how the memory should be allocated while allowing pre-existing locations to be recycled, the dynamic semantics of AML does not translate to an algorithm directly. The semantics, however, can be implemented by controlling reuse of computations and memory locations carefully. Since the publication of the conference version of this paper, several such implementations have been completed and applied to a relatively broad range

```
datatype 'a list = nil | cons 'a * 'a list
map :: ('a -> 'b) -> ('a list) -> ('b list)
fun map f l =
  case l of
    nil => nil
    cons(h,t) => cons(f h, map f t)
```

Fig. 1: The code for map in an ML-like functional language.

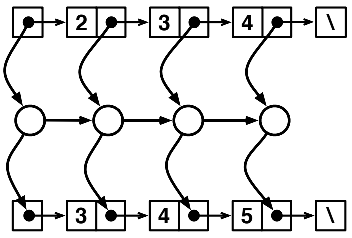

Fig. 2: The call tree and the data dependencies for `map (fn x => x+1) [2,3,4]`.

of problems. In Sections 7 and 8, we briefly overview the follow-up papers that implement and apply the presented semantics.

2 Overview

We briefly describe the main limitation of self-adjusting computation based on dynamic dependency graphs and change propagation [Acar et al. 2006], how it may be improved, the challenges in realizing this improvement, and how we tackle these challenges. We use an ML like language extended with several primitives for self-adjusting computation and use a simple example to guide the discussion. We make the language primitives and their semantics precise in the rest of the paper.

Example: `map`. Figure 1 shows the code for our running example, `map`. The function `map` operates on lists, which are defined either as empty (`nil`) or a cons cell consisting of a head element and a tail, which itself is a list. It applies the specified function `f` to every element of the list to produce an output list of the same length.

Call trees and data dependencies. In this discussion, we will carefully look into the structure of the evaluations of the `map` function. We will represent evaluations with their *computation graphs* consisting of the function calls performed during the execution along with edges that illustrate the function-call operations (caller-callee relationships between function calls) and the dependencies between data and function calls. We refer to the sub-graph consisting of function calls and the edges between them as the *call tree*. Computation graphs are similar to the dynamic dependency graphs that we will discuss soon.

Figure 2 illustrates the computation graph for an execution of `map` when applied to the list `[2,3,4]` with the integer increment function. We draw a list as a sequence of "cons" cells, each of which contain an element and a "tail". We draw the call tree as a sequence of circles, each of which represent a function call. The calls are linked with edges that represent the function calls, or the caller-callee relationships. Additionally, we illustrate

```
datatype 'a modlist' = nil | cons 'a * ('a modlist' mod)
type 'a modlist = 'a modlist'
map :: (a -> b) -> ('a modlist) -> ('a modlist)
fun map f l =
  mod (read l as l' in
    case l' of
      nil => write nil
      cons(h,t) => write (cons(f h, map f t)))
```

Fig. 3: The code for self-adjusting map with modifiables.

the data dependencies between the function calls and the input by using curved edges. For simplicity, we omit the data dependencies between calls and the individual elements (which are implied by the dependencies on the cons cells), and the dependencies to the increment function.

We say that a function call u is *deeper* (*shallower*) than another call v if the distance between the root of the call tree and u is greater (less) than the distance between v and the root. Informally, we say that a function call is deep (shallow) if it has only a constant number of descendants (ancestors), i.e., the number of descendants is independent of the input size. The shallowest node in a call tree is the root and the deepest nodes are the leaves.

Following a similar methodology, we say that a data change is *deep* (*shallow*) if the function calls that depend on the data affected by the change are deep (shallow). For example, in Figure 2, inserting a new element at the end of the list performs a deep change, because the function calls affected are deep in the call tree; inserting a new element at the head of the list performs a shallow change because the function calls affected are shallow.

Dynamic dependency graphs and change propagation. To enable computations respond to incremental changes to their data, previous work on self-adjusting computation propose the notion of *modifiable references* or *modifiables* for writing programs that can respond to changes to the contents of modifiables automatically. The idea behind the automatic updates is to record the evaluation of a program in the form of a *dynamic dependency graph* and use a *change-propagation* algorithm to update the graph whenever the contents of the modifiables change.

Figure 3 illustrates the code for the self-adjusting `map` written with modifiable references. We define a modifiable list, `modlist`, as a list where each "tail" is inside of a modifiable. We use the notation [| |] to denote modifiable lists. Placing the tail inside a modifiable allows the user to change (mutate) its contents and ask the computation to update the output automatically. Having determined the input type for `map`, we modify the body of the function by inserting primitives for creating (`mod`), reading (`read`), and writing (`write`) modifiable references so that the output is also a modifiable list. These primitives are analogous to the primitives for creating, dereferencing, and writing ordinary references but are actually quite different. For example, the `read` primitive requires the modifiable to be read, and specifies a body (an expression) that would be evaluated with the contents of the modifiable read. A `read` takes place only inside the scope of a modifiable (`mod`) and ends with a `write` to that modifiable. Type systems can enforce correct usage of these primitives [Carlsson 2002; Acar et al. 2006, 2009].

Having annotated the code for `map` to operate on modifiable lists, we can now evaluate it. As this evaluation takes place, we construct a dynamic dependency graph to represent the evaluation. The dynamic dependency graph consists of a call tree of evaluated `read` operations and the data dependencies between `read` operations and the modifiables that they read. The nodes and edges are ordered to enable determining when in the evaluation the actions represented by the dependence graph takes place. Dynamic dependency graphs are closely related to computations graphs discussed previously in this section. We can use computations graphs to represent dynamic dependency graphs by tracking not all function calls but just the calls to `read` operations. For example, we can use Figure 2 to illustrate the dynamic dependency graph for the evaluation, `map (fn x => x+1) [|2,3,4|]`. In the illustration, modifiables are shown as dark circles, memory cells are shown as squares, and `read` operations are shown as empty circles. The straight edges between read operations illustrate the control flow between the reads. The curved edges represent read/write from/into modifiables.

After we construct a dynamic dependency graph, we can change the input by mutating the contents of the modifiables and ask the output and the dynamic dependency graph to be updated by calling change propagation. Intuitively, change propagation mimics a complete re-evaluation of the program with the changed data but only re-evaluates the `read` operations that depend on the changes. To this end, change propagation maintains a work queue of reads to be re-evaluated; the work queue initially contains the `read`'s that directly depend on the changed modifiables. When re-evaluated a `read` can change other data (by writing to memory), effectively inserting the `read`'s depending on the changed data into the work queue. For correctness, change propagation re-evaluates the `read`'s in the work queue in the same order as they are originally evaluated. This is important because re-evaluation can affect, via data and control dependencies, `read`'s that come after. Since it is impossible to know a priori which `read`'s will be evaluated, change propagation deletes all the descendants of a `read` from the DDG along with their dependencies. When change propagation completes, it yields a DDG that is isomorphic to the DDG that would be obtained by re-evaluating the program from scratch with the modified data.

Limitations of change propagation. Consider evaluating
`map (fn x => x+1) [|2,3,4|]`,
and then inserting a new element `5` at the end of the input list. We can compute the new output by evaluating
`map (fn x => x+1) [|2,3,4,5|]`,
which, in the general case, would take linear time in the length of the list. In self-adjusting computation, instead or re-evaluating, we can use change propagation by starting with the DDG for `map (fn x => x+1) [|2,3,4|]`, inserting `5` into the input by mutating the appropriate modifiable, and then performing change propagation. Figure 4 (bottom left) illustrates such a change propagation , highlighting the new and affected parts of the dynamic dependency graph. To insert the new key, we mutate the modifiable containing the tail of the cell `4`. This change affects the recursive `map` call on the changed tail (more precisely the `read` within), which when evaluated would map the inserted element, and the empty list (`nil`) to update the new output. Since change propagation evaluates the body of the `map` once on the new element and once on the empty list, it would update the output in constant time. More generally, it is not difficult to see that change propagation performs

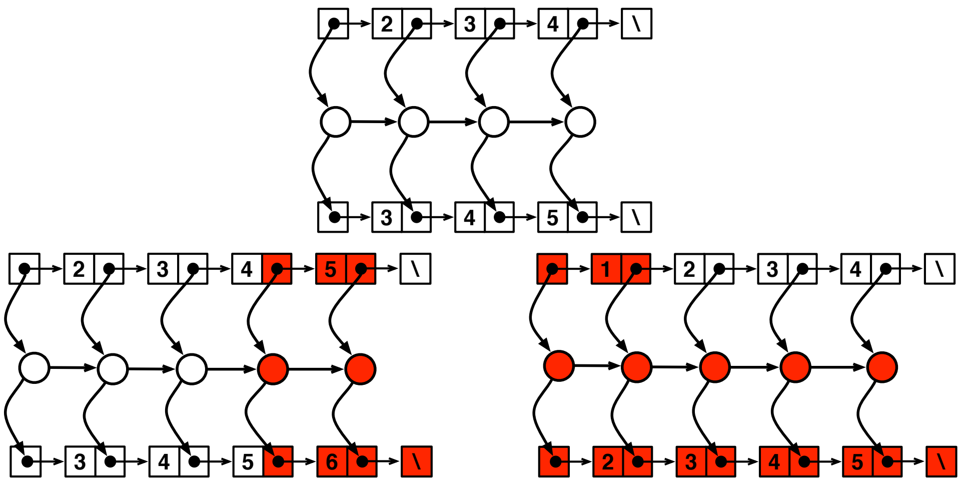

Fig. 4: Change propagation: a dynamic dependency graph (top), and deep changes (bottom left) and shallow changes (bottom right).

well for deep changes, because such changes affect function calls that have a small number of descendants that would need to re-evaluated.

Unfortunately, change propagation does not perform well with shallow changes. To see this, consider evaluating

```
map (fn x => x+1) [|2,3,4|]
```

and then inserting the new element `1` at the head of the input list. We can compute the new output by evaluating `map (fn x => x+1) [|1,2,3,4|]`, which, in the general case, would take linear time in the length of the list. We can also use change propagation by starting with the dynamic dependency graph for `map (fn x => x+1) [|2,3,4|]`, inserting `1` into the input, and then performing change propagation. Figure 4 (bottom right) illustrates such a change propagation, highlighting the new and affected parts of the dynamic dependency graph. To insert the new key, we modify the beginning of the list. This change affects the first recursive `map` call (more precisely, the `read` within), which when evaluated maps the inserted element, as well as the rest of the input list, essentially re-computing the output. Thus in this case, change propagation requires linear time. It is not difficult to see that in the general case change propagation performs poorly with shallow changes, because such changes have many descendants that need to be re-evaluated.

By generalizing these examples, we can show that when averaged over all possible insertions and deletions (e.g., at first, second, third location etc.), change propagation requires asymptotically linear time. Thus, change propagation yields no asymptotic improvement in evaluation time compared to re-evaluating the program.

Memoization. The classic idea of memoization or function caching offers another approach to re-using computation by remembering results of functions calls and re-using them when possible. Memoization can be applied automatically by memoizing every function call, but it is often more efficient to allow the programmer to control memoization by allowing them to annotate the expressions to be memoized. Figure 5 shows the code for the `map` function, where the body of the function is memoized by using the `memo` keyword.

```
datatype 'a list = nil | cons 'a * ('a list)
map :: (a -> b) -> ('a list) -> ('b list)
fun map f l =
  memo(
    case l of
      nil => nil
      cons(h,t) => cons(f h, map f t))
```

Fig. 5: The code for `map` with memoization.

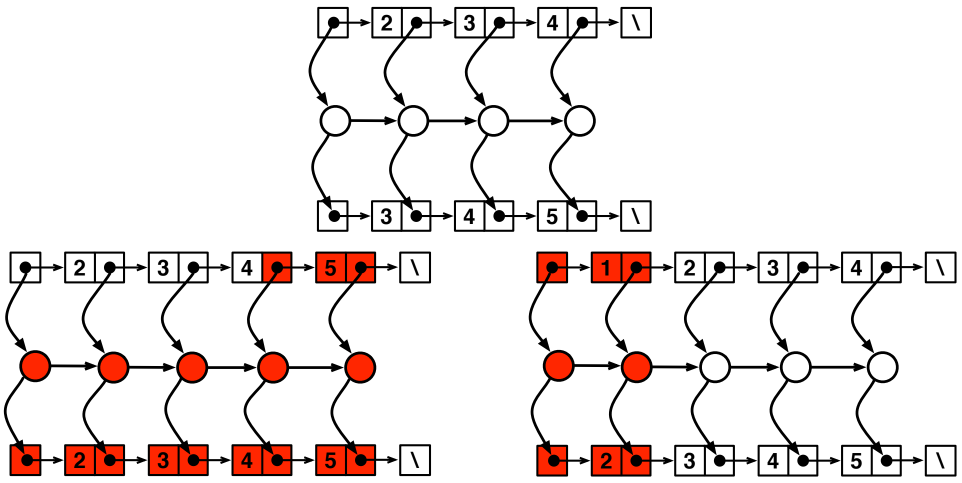

Fig. 6: Memoization: computation graph (top), deep changes (bottom left) and shallow changes (bottom right).

With such memoization, whenever `map` is called with the same argument, the result can be re-used instead of re-computing it unnecessarily. To support such re-use of results, we store a mapping from memoized expressions and the values of their free variables (arguments) to the results in some kind of table data structure that provides fast searches.

Memoization can help in incremental computation in cases where making small changes to input data requires a comparatively small change in the set of function calls performed; i.e., when many calls are repeated. As we now describe, as with change propagation, the effectiveness of memoization depends on how data is changed. More specifically, we show an interesting duality between memoization and change propagation: memoization performs poorly with deep changes (where change propagation performs well) and performs well with shallow changes (where change propagation performs poorly).

Let's consider our two examples involving a deep and a shallow change starting with the evaluation
`map (fn x => x+1) [2,3,4]`.
When we perform this evaluation with memoization, we effectively store the results of the calls to `map` with `[2,3,4]`, `[3,4]`, `[3,4]`, `[4]`, and `[]`. Suppose now we change the input by inserting a new element 5 at the end. As illustrated in Figure 6 (bottom left), when we evaluate

`map (fn x => x+1) [2,3,4,5]`,
we cannot re-use any of the results except for the empty list case, because the inputs will never match (they will all contain the new element `5`). Consider now inserting the new element, `1`, at the head of the list. This time, as illustrated in Figure 6 (bottom right), we can re-use all function calls except for the first call that operates on the new element (`1`).

By generalizing these example, we can show that memoization performs well for shallow changes but performs poorly for deep changes. For the `map` function specifically, we can show that when averaged over all possible insertions and deletions (e.g., at first, second, third location etc.), memoization requires asymptotically linear time. Thus memoization yields no asymptotic improvement in evaluation time compared to re-evaluating an program.

Memoizaing change propagation. As described above neither change propagation nor memoization alone are effective in ensuring efficient incremental updates. They are, however, duals in how they perform: change propagation performs well with deep changes, whereas memoization performs well with shallow changes. This suggests that it might be possible to obtain an effective technique by using memoization and change propagation in combination.

One way to apply memoization and change propagation would be to use memoization for shallow changes and change propagation for deep changes. Unfortunately, many changes are neither deep nor shallow. As an example, imagine inserting a new element in the middle of the input list. Such a change is not deep, because affected nodes can have many descendant in the call tree. Nor is it shallow, because affected nodes can have many ancestor in the call tree. As another example, consider inserting an element at the head and another at the end of the list as shown in Figure 8; such a change is also neither deep nor shallow. Thus, such an orthogonal application of change propagation and memoization does not provide an efficient approach to computation re-use.

There is another major challenge that must be overcome for memoization and change propagation to be applied to the same computation: memoization requires purely functional programming but change propagation is an imperative algorithm and mutates memory. The purity assumption is necessary for memoization, because otherwise the arguments of a function do not necessarily determine its result. One way to generalize memoization would be to take a "brute-force" approach and track all data dependencies of a function call, including those that are updated imperatively. This approach, however, is not effective because it requires checking for equality of all stored dependencies when performing a memo lookup. Such an equality test can require linear time in the size of the reachable data. To the best our knowledge, there are no known techniques for improving the complexity of equality checks (in the purely functional setting, there is). Linear-time equality test introduce a large overhead to computation. For example, when we evaluate `map` with this approach, many calls to `map` would require linear time (in the size of the input) just to compare the arguments, because the tail of the list would be accessible via the argument. Thus the memoized `map` function can take linear time even in the case of a perfect match, wiping out the benefits of result re-use.

To apply memoization and change propagation to the same computation, we integrate them closely so that they can work together to maximize re-use of results. To apply the proposed technique, the programmer to operate on all changeable data by using primitives

```
datatype 'a modlist' = nil | cons 'a * ('a modlist' mod)
datatype 'a modlist = 'a modlist'
fun map (f: a -> b) (l: 'a modlist) =
  memo (mod (read l as l' in
    case l of
      nil => write nil
      cons(h,t) => write (cons(f h, map f t))))
```

Fig. 7: The code for self-adjusting map with modifiables.

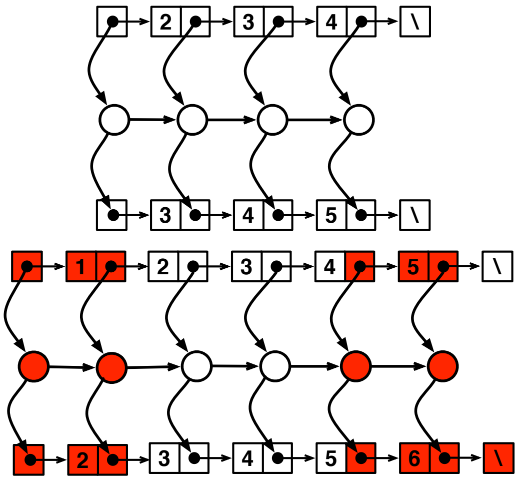

Fig. 8: Change propagation + memoization: a dynamic dependency graph (top), and its update.

on modifiables, and memoizes desired expressions via a simple `memo` keyword. Figure 7 shows the code for `map`. To take advantage of modifiables, we convert the input list into a modifiable list; to take advantage of memoization, we memoize the body of the function. Memoizing the body of the function effectively memoizes all recursive calls to `map`.

To support effective updates under incremental changes to data, we represent computations with their dynamic dependency graphs and also remember for each memoized expression the dynamic dependency graph of that expression. Remembering the dynamic dependency graph enables using memoization, even as the memory is mutated imperatively, by allowing us to adapt re-used computations to mutations via change propagation. As a result, when matching memoized expressions, we can rely on syntactic or tag/label equality without having to compare the whole of data that the expression may depend on—more precisely, equality checks are agnostic to the contents of the store or memory.

As an example consider the case shown in Figure 8 and suppose that we perform change propagation after inserting the elements `1` and `5`. Change propagation will start by re-executing the read inside the first call to `map`, which will map the new element to the output and recursively call `map` on the tail. This recursive call has as its input a list that is similar to its input in the previous invocation but not quite the same, because the element `5` appears at the end. Our memoization mechanism will still match and re-use its dynamic dependency graph by matching the (label of) modifiable holding the input list. Since, however, the

Values	v	$::=$	$() \mid n \mid x \mid l \mid (v_1,v_2) \mid \texttt{in}_{\texttt{l}}\ v \mid \texttt{in}_{\texttt{r}}\ v \mid$ $\texttt{fun}_{\texttt{s}}\ f(x)\ \texttt{is}\ e_s \mid \texttt{fun}_{\texttt{c}}\ f(x)\ \texttt{is}\ e_c$
Primitive Operations	o	$::=$	$\texttt{not} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots$
Expression	e	$::=$	$e_s \mid e_c$
Stable Expressions	e_s	$::=$	$v \mid o(v_1,\ldots,v_n) \mid \texttt{mod}\ e_c \mid \texttt{memo}_{\texttt{s}}\ e_s \mid$ $\texttt{apply}_{\texttt{s}}(v_1,v_2) \mid \texttt{let}\ x = e_s\ \texttt{in}\ e'_s \mid$ $\texttt{let} x_1 \times x_2 = v\, \texttt{in} e_s \mid$ $\texttt{case} v \texttt{of}\ \{\texttt{in}_{\texttt{l}}\,(x_1) \Rightarrow e_s\, , \texttt{in}_{\texttt{r}}\,(x_2) \Rightarrow e'_s\}$
Changeable Expressions	e_c	$::=$	$\texttt{write}(v) \mid \texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c \mid \texttt{memo}_{\texttt{c}}\ e_c \mid$ $\texttt{apply}_{\texttt{c}}(v_1,v_2) \mid \texttt{let}\ x = e_s\ \texttt{in}\ e_c \mid$ $\texttt{let} x_1 \times x_2 = v\, \texttt{in} e_c \mid$ $\texttt{case} v \texttt{of}\ \{\texttt{in}_{\texttt{l}}\,(x_1) \Rightarrow e_c\, , \texttt{in}_{\texttt{r}}\,(x_2) \Rightarrow e'_c\}$
Program	p	$::=$	e_s

Fig. 9: The abstract syntax of AML.

input is different than the previous (it has a new tail containing 5), the computation must be updated.[1] We update the re-used computation by a recursive invocation of change propagation on the dynamic dependency graph reused. In the example, change propagation will execute the final recursive call to `map`, which will map the final element, reuse the computation for the empty list, and return. Thus, we have updated the output by performing small local computations only for the fragments of the computation (highlighted in the figure) affected by the change, regardless of the number of elements in the middle of the list.

3 The Language

We describe a language, called AML, that combines the features of an adaptive functional language (AFL) [Acar et al. 2006] with memoization. The syntax of the language extends that of AFL with **memo** constructs for memoizing expressions. The dynamic semantics integrates change propagation and evaluation to ensure correct reuse of computations under mutations. As explained before, our results do not rely on typing properties of AML. We therefore omit a type system but identify a minimal set of conditions under which evaluation is consistent. In addition to the memoizing and change-propagating dynamic semantics, we give a pure interpretation of AML that provides no reuse of computations.

3.1 Abstract syntax

The abstract syntax of AML is given in Figure 9. We use meta-variables x, y, and z (and variants) to range over an unspecified set of variables, and meta-variable l (and variants)

[1] Note that with classical memoization, where we reuse results of function calls, such a re-use would not be possible.

```
funs map (f,x) is
  memo (mod (
    read x as x' in
      case x of { inl()⇒ write (inl()),
                  inr y ⇒ let h × t = y in
                          let h' = applys (f,h) in
                          let t' = applys (map,(f,t)) in
                            write(inr(h',t'))
                          end end end
                }
    end))
```

Fig. 10: The code for `map` in AML.

to range over a separate, unspecified set of locations—the locations are modifiable references. The syntax of AML is restricted to “2/3-cps”, or “named form”, to streamline the presentation of the dynamic semantics.

Expressions are classified into three categories: values, *stable* expressions, and *changeable* expressions. Values are constants, variables, locations, and the introduction forms for sums, products, and functions. The value of a stable expression is not sensitive to modifications to the inputs, whereas the value of a changeable expression may be directly or indirectly affected by them.

The familiar mechanisms of functional programming are embedded in AML as stable expressions. Stable expressions include the `let` construct, the elimination forms for products and sums, application of stable functions, and the creation of new modifiables. A *stable function* is a possibly recursive function whose body is a stable expression. The application of a stable function, $\texttt{apply}_{\texttt{s}}(v_1,v_2)$, where $v_1 = \texttt{fun}_{\texttt{s}}\ f(x)\ \texttt{is}\ e_s$ is a stable expression. The expression $\texttt{mod}\ e_c$ allocates a modifiable reference and initializes its contents by executing the changeable expression e_c. Note that the modifiable itself is stable, even though its contents is subject to change. A memoized stable expression is written $\texttt{memo}_{\texttt{s}}\ e_s$.

Changeable expressions always execute in the context of an enclosing `mod`-expression that provides the implicit target location to which every changeable expression writes (evaluation of every changeable expression ends by writing to the target location). The changeable expression $\texttt{write}(v)$ writes the value v into the target. The expression $\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c$ 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* is a possibly recursive function whose body is a changeable expression. A changeable function is stable as a value. The application of a changeable function is a changeable expression. A memoized changeable expression is written $\texttt{memo}_{\texttt{c}}\ e_c$. The changeable expressions include the `let` expression for ordering evaluation and the elimination forms for sums and products. These differ from their stable counterparts because their bodies consists of changeable expressions.

Figure 10 illustrates the code for `map` in AML. Since AML is untyped, the code contains no typing information. Since the language is untyped, expressing recursive data structures such as lists merely by using sum types becomes possible.

3.2 Stores, well-formed expressions, and lifting

Evaluation of an AML expression takes place in the context of a store, written σ (and variants), defined as a finite map from locations l to values v. We write $\mathtt{dom}(\sigma)$ for the domain of a store, and $\sigma(l)$ for the value at location l, provided $l \in \mathtt{dom}(\sigma)$. We write $\sigma[l \leftarrow v]$ to denote the extension of σ with a mapping of l to v. If l is already in the domain of σ, then the extension replaces the previous mapping.

$$\begin{aligned} \sigma[l \leftarrow v](l') &= \begin{cases} v & \text{if } l = l' \\ \sigma(l') & \text{if } l \neq l' \text{ and } l' \in \mathtt{dom}(\sigma) \end{cases} \\ \mathtt{dom}(\sigma[l \leftarrow v]) &= \mathtt{dom}(\sigma) \cup \{l\} \end{aligned}$$

We say that an expression e is *well-formed* in store σ if 1) all locations reachable from e in σ are in $\mathtt{dom}(\sigma)$ (“no dangling pointers”), and 2) the portion of σ reachable from e is free of cycles. If e is well-formed in σ, then we can obtain a “lifted” expression e' by recursively replacing every reachable location l with its stored value $\sigma(l)$. The notion of lifting will be useful in the formal statement of our main theorems (Section 4).

We use the judgment $e, \sigma \xrightarrow{\mathtt{wf}} e', L$ to say that e is well-formed in σ, that e' is e lifted in σ, and that L is the set of locations reachable from e in σ. The rules for deriving such judgments are shown in Figure 11. Any finite derivation of such a judgment implies well-formedness of e in σ.

We will use two notational shorthands for the rest of the paper: by writing $e \uparrow \sigma$ or $\mathtt{reach}\,(e, \sigma)$ we implicitly assert that there exist a location-free expression e' and a set of locations L such that $e, \sigma \xrightarrow{\mathtt{wf}} e', L$. The notation $e \uparrow \sigma$ itself stands for the lifted expression e', and $\mathtt{reach}\,(e, \sigma)$ stands for the set of reachable locations L. It is easy to see that e and σ uniquely determine $e \uparrow \sigma$ and $\mathtt{reach}\,(e, \sigma)$ (if they exist).

3.3 Dynamic semantics

The evaluation judgments of AML (Figures 14 and 15) consist of separate judgments for stable and changeable expressions. The judgment $\sigma, e \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s$ states that evaluation of the stable expression e relative to the input store σ yields the value v, the trace $\mathtt{T}_s$ (defined below), and the updated store σ'. Similarly, the judgment $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}_c$ states that evaluation of the changeable expression e relative to the input store σ writes its value to the target l, and yields the trace $\mathtt{T}_c$ together with the updated store σ'.

Since we do not employ a type system, the evaluation judgment is partial: there are many terms that cannot be evaluated using the presented judgments. Specifically, ill-typed terms that would be ruled out by a type system ”get stuck” and have no evaluation derivation.

A *trace* records the adaptive aspects of evaluation. Like the expressions whose evaluations they describe, traces come in stable and changeable varieties. The abstract syntax of

$$\frac{v \in \{\texttt{()}, n, x\}}{v, \sigma \xrightarrow{\texttt{wf}} v, \emptyset} \qquad \frac{l \in \texttt{dom}(\sigma) \quad \sigma(l), \sigma \xrightarrow{\texttt{wf}} v, L}{l, \sigma \xrightarrow{\texttt{wf}} v, \{l\} \cup L}$$

$$\frac{v_1, \sigma \xrightarrow{\texttt{wf}} v_1', L_1 \quad v_2, \sigma \xrightarrow{\texttt{wf}} v_2', L_2}{(v_1, v_2), \sigma \xrightarrow{\texttt{wf}} (v_1', v_2'), L_1 \cup L_2}$$

$$\frac{v, \sigma \xrightarrow{\texttt{wf}} v', L}{\texttt{in}_{\texttt{l}}\ v, \sigma \xrightarrow{\texttt{wf}} \texttt{in}_{\texttt{l}}\ v', L} \qquad \frac{v, \sigma \xrightarrow{\texttt{wf}} v', L}{\texttt{in}_{\texttt{r}}\ v, \sigma \xrightarrow{\texttt{wf}} \texttt{in}_{\texttt{r}}\ v', L}$$

$$\frac{e, \sigma \xrightarrow{\texttt{wf}} e', L}{\texttt{fun}_{\{\texttt{S},\texttt{C}\}}\ f(x)\ \texttt{is}\ e, \sigma \xrightarrow{\texttt{wf}} \texttt{fun}_{\{\texttt{S},\texttt{C}\}}\ f(x)\ \texttt{is}\ e', L}$$

$$\frac{v_1, \sigma \xrightarrow{\texttt{wf}} v_1', L_1 \quad \cdots \quad v_n, \sigma \xrightarrow{\texttt{wf}} v_n', L_n}{o\,\texttt{(}v_1, \ldots, v_n\texttt{)}, \sigma \xrightarrow{\texttt{wf}} o\,\texttt{(}v_1', \ldots, v_n'\texttt{)}, L_1 \cup \cdots \cup L_n}$$

$$\frac{e, \sigma \xrightarrow{\texttt{wf}} e', L}{\texttt{memo}_{\{\texttt{S},\texttt{C}\}}\ e, \sigma \xrightarrow{\texttt{wf}} \texttt{memo}_{\{\texttt{S},\texttt{C}\}}\ e', L}$$

$$\frac{v_1, \sigma \xrightarrow{\texttt{wf}} v_1', L_1 \quad v_2, \sigma \xrightarrow{\texttt{wf}} v_2', L_2}{\texttt{apply}_{\{\texttt{S},\texttt{C}\}}\texttt{(}v_1, v_2\texttt{)}, \sigma \xrightarrow{\texttt{wf}} \texttt{apply}_{\{\texttt{S},\texttt{C}\}}\texttt{(}v_1', v_2'\texttt{)}, L_1 \cup L_2}$$

$$\frac{e_1, \sigma \xrightarrow{\texttt{wf}} e_1', L \quad e_2, \sigma \xrightarrow{\texttt{wf}} e_2', L'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2, \sigma \xrightarrow{\texttt{wf}} \texttt{let}\ x = e_1'\ \texttt{in}\ e_2', L \cup L'}$$

$$\frac{v, \sigma \xrightarrow{\texttt{wf}} v', L \quad e, \sigma \xrightarrow{\texttt{wf}} e', L'}{\texttt{let}\,x_1 \times x_2\ =\ v\,\texttt{in}\,e, \sigma \xrightarrow{\texttt{wf}} \texttt{let}\,x_1 \times x_2\ =\ v'\,\texttt{in}\,e', L \cup L'}$$

$$\frac{v, \sigma \xrightarrow{\texttt{wf}} v', L \quad e_1, \sigma \xrightarrow{\texttt{wf}} e_1', L_1 \quad e_2, \sigma \xrightarrow{\texttt{wf}} e_2', L_2}{\begin{array}{l}\texttt{case}\,v\,\texttt{of}\ \{\texttt{in}_{\texttt{l}}\,\texttt{(}x_1\texttt{)}\ \Rightarrow\ e_1\ \texttt{,}\ \texttt{in}_{\texttt{r}}\,\texttt{(}x_2\texttt{)}\ \Rightarrow\ e_2\}, \sigma \xrightarrow{\texttt{wf}} \\ \qquad \texttt{case}\,v'\,\texttt{of}\ \{\texttt{in}_{\texttt{l}}\,\texttt{(}x_1\texttt{)}\ \Rightarrow\ e_1'\ \texttt{,}\ \texttt{in}_{\texttt{r}}\,\texttt{(}x_2\texttt{)}\ \Rightarrow\ e_2'\}, L \cup L_1 \cup L_2\end{array}}$$

$$\frac{e_c, \sigma \xrightarrow{\texttt{wf}} e_c', L}{\texttt{mod}\ e_c, \sigma \xrightarrow{\texttt{wf}} \texttt{mod}\ e_c', L} \qquad \frac{v, \sigma \xrightarrow{\texttt{wf}} v', L}{\texttt{write(}v\texttt{)}, \sigma \xrightarrow{\texttt{wf}} \texttt{write(}v'\texttt{)}, L}$$

$$\frac{v, \sigma \xrightarrow{\texttt{wf}} v', L \quad e_c, \sigma \xrightarrow{\texttt{wf}} e_c', L'}{\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c, \sigma \xrightarrow{\texttt{wf}} \texttt{read}\ v'\ \texttt{as}\ x\ \texttt{in}\ e_c', L \cup L'}$$

Fig. 11: Well-formed expressions and lifts.

traces is given by the following grammar:

$$\begin{array}{lll} \textit{Stable} & \mathtt{T}_s ::= & \varepsilon \mid \mathtt{mod}\ l \leftarrow \mathtt{T}_c \mid \mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_s \\ \textit{Changeable} & \mathtt{T}_c ::= & \mathtt{write}\ v \mid \mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_c \mid \mathtt{read}_{l \to x=v.e}\ \mathtt{T}_c \end{array}$$

A stable trace records the sequence of allocations of modifiables that arise during the evaluation of a stable expression. The trace $\mathtt{mod}\ l \leftarrow \mathtt{T}_c$ records the allocation of the modifiable l and the trace of the initialization code for l. The trace $\mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_s'$ results from evaluating 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, $\mathtt{write}\ v$, records the storage of the value v in the target. A sequence $\mathtt{let}\ \mathtt{T}_s\ \mathtt{T}_c$ 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 $\mathtt{read}_{l \to x=v.e}\ \mathtt{T}_c$ specifies the location read (l), the value read (v), the context of use of its value ($x.e$) and the trace ($\mathtt{T}_c$) of the remainder of the evaluation within the scope of that read. A read-trace records the dependency of the target on the value of the location read. The context $x.e$ specifies the expression that depends on the contents of the location, the sources; change propagation re-evaluates the expressions when the sources changes.

We define the set of allocated locations of a trace $\mathtt{T}$, denoted $\mathtt{alloc}(\mathtt{T})$, as follows:

$$\begin{array}{lcl} \mathtt{alloc}(\varepsilon) & = & \emptyset \\ \mathtt{alloc}(\mathtt{write}\ v) & = & \emptyset \\ \mathtt{alloc}(\mathtt{mod}\ l \leftarrow \mathtt{T}_c) & = & \{l\} \cup \mathtt{alloc}(\mathtt{T}_c) \\ \mathtt{alloc}(\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2) & = & \mathtt{alloc}(\mathtt{T}_1) \cup \mathtt{alloc}(\mathtt{T}_2) \\ \mathtt{alloc}(\mathtt{read}_{l \to x=v.e}\ \mathtt{T}_c) & = & \mathtt{alloc}(\mathtt{T}_c) \end{array}$$

For example, if $\mathtt{T}_{\text{sample}} = \mathtt{let}\ (\mathtt{mod}\ l_1 \leftarrow \mathtt{write}\ 2)\ (\mathtt{read}_{l_1 \to x=2.e}\ \mathtt{write}\ 3)$, then $\mathtt{alloc}(\mathtt{T}_{\text{sample}}) = \{l_1\}$.

Well-formedness, lifts, and primitive operations. We require that primitive operations preserve well-formedness. In other words, when a primitive operation is applied to some arguments, it does not create dangling pointers or cycles in the store, nor does it extend the set of locations reachable from the argument. Formally, this property can be stated as follows.

$$\begin{array}{l} \text{If } \forall i.v_i, \sigma \xrightarrow{\mathtt{wf}} v_i', L_i \text{ and } v = o(v_1, \ldots, v_n), \\ \text{then } v, \sigma \xrightarrow{\mathtt{wf}} v', L \text{ such that } L \subseteq \bigcup_{i=1}^n L_i. \end{array}$$

Moreover, primitive operations must be insensitive to the identity of locations. In the case of primitive operations we formalize this by postulating that they commute with lifts:

$$\begin{array}{l} \text{If } \forall i.v_i, \sigma \xrightarrow{\mathtt{wf}} v_i', L_i \text{ and } v = o(v_1, \ldots, v_n), \\ \text{then } v, \sigma \xrightarrow{\mathtt{wf}} v', L \text{ such that } v' = o(v_1', \ldots, v_n'). \end{array}$$

In short this can be stated as $o(v_1 \uparrow \sigma, \ldots, v_n \uparrow \sigma) = (o(v_1, \ldots, v_n)) \uparrow \sigma$.

For example, all primitive operations that operate only on non-location values preserve well formedness and commute with lifts.

Valid evaluations. We consider only evaluations of well-formed expressions e in stores σ, i.e., those e and σ where $e \uparrow \sigma$ and $\mathtt{reach}(e, \sigma)$ are defined. Well-formedness is critical

$$\frac{\begin{array}{c}\sigma, e_s \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T} \\ \mathtt{alloc}(\mathtt{T}) \cap \mathtt{reach}(e_s, \sigma) = \emptyset\end{array}}{\sigma, e_s \Downarrow^{\mathsf{S}}_{\text{ok}} v, \sigma', \mathtt{T}} \;\textbf{(valid/s)} \qquad \frac{\begin{array}{c}\sigma, l \leftarrow e_c \Downarrow^{\mathsf{C}} \sigma', \mathtt{T} \\ \mathtt{alloc}(\mathtt{T}) \cap \mathtt{reach}(e_c, \sigma) = \emptyset \\ l \notin \mathtt{reach}(e_c, \sigma) \cup \mathtt{alloc}(\mathtt{T})\end{array}}{\sigma, l \leftarrow e_c \Downarrow^{\mathsf{C}}_{\text{ok}} \sigma', \mathtt{T}} \;\textbf{(valid/c)}$$

Fig. 12: Valid evaluations.

for proving correctness. First, the requirement that the reachable portion of the store is acyclic ensures that the approach is consistent with purely functional programming. Second, the requirement that all reachable locations are in the store ensures that evaluations do not cause disaster by allocating a "fresh" location that happens to be reachable. We note that it is possible to omit the well-formedness requirement by giving a type system and a type safety proof. This approach limits the applicability of the theorem to only type-safe programs. Because of the imperative nature of the dynamic semantics, a type safety proof for AML is also complicated. We therefore choose to formalize well-formedness separately.

Our approach requires showing that evaluation preserves well-formedness. To establish well-formedness inductively, we define *valid evaluations*. We say that an evaluation of an expression e in the context of a store σ is *valid*, if

1. e is well-formed in σ,
2. the locations allocated during evaluation are disjoint from locations that are initially reachable from e (i.e., those that are in $\mathtt{reach}(e,\sigma)$), and
3. the target location of a changeable evaluation is contained neither in $\mathtt{reach}(e,\sigma)$ nor in the locations allocated during evaluation.

We use $\Downarrow^{\mathsf{S}}_{\text{ok}}$ instead of $\Downarrow^{\mathsf{S}}$ and $\Downarrow^{\mathsf{C}}_{\text{ok}}$ instead of $\Downarrow^{\mathsf{C}}$ to indicate valid stable and changeable evaluations, respectively. The rules for deriving valid evaluation judgments are shown in Figure 12.

The Oracle. The dynamic semantics for AML uses an oracle to model memoization. Figure 13 shows the evaluation rules for the oracle. For a stable or a changeable expression e, we write an oracle miss as $\sigma, e \uparrow^{\mathsf{S}}$ or $\sigma, l \leftarrow e_c \uparrow^{\mathsf{C}}$, respectively. The treatment of oracle hits depends on whether the expression is stable or changeable. For a stable expression, it returns the value and the trace of a valid evaluation of the expression in some store. For a changeable expression, the oracle returns a trace of a valid evaluation of the expression in some store with some destination. Note that the target location that is part of the evaluation of the changeable expression is dropped. This enables re-use of a changeable computation in the context of another target location. This is important for efficiency. Indeed, otherwise, changeable computations can only be used in the context of the same target location. For example, a sequence of tail recursive function calls may all have to be repeated when the target location does not match.

The key difference between the oracle and conventional approaches to memoization is that the oracle is free to return the trace (and the value, for stable expressions) of a computation that is consistent with any store—not necessarily with the current store. Since the evaluation whose results are being returned by the oracle can take place in a different store than the current store, the trace and the value (if any) returned by the oracle cannot

$$\frac{}{\sigma, e_s \uparrow^{\mathtt{S}}}\ \textbf{(miss/s)} \qquad \frac{\sigma_0, e_s \Downarrow^{\mathtt{S}}_{\mathrm{ok}} v, \sigma_0', \mathtt{T}}{\sigma, e_s \downarrow^{\mathtt{S}} v, \mathtt{T}}\ \textbf{(hit/s)}$$

$$\frac{}{\sigma, e_c \uparrow^{\mathtt{C}}}\ \textbf{(miss/c)} \qquad \frac{\sigma_0, l \leftarrow e_c \Downarrow^{\mathtt{C}}_{\mathrm{ok}} \sigma_0', \mathtt{T}}{\sigma, e_c \downarrow^{\mathtt{C}} \mathtt{T}}\ \textbf{(hit/c)}$$

Fig. 13: The oracle.

$$\frac{}{\sigma, v \Downarrow^{\mathtt{S}} v, \sigma, \varepsilon}\ \textbf{(value)} \qquad \frac{v = \mathtt{app}(o, (v_1, \ldots, v_n))}{\sigma, o(v_1, \ldots, v_n) \Downarrow^{\mathtt{S}} v, \sigma, \varepsilon}\ \textbf{(prim.'s)}$$

$$\frac{l \notin \mathtt{alloc}(\mathtt{T}) \qquad \sigma, l \leftarrow e \Downarrow^{\mathtt{C}} \sigma', \mathtt{T}}{\sigma, \mathtt{mod}\ e \Downarrow^{\mathtt{S}} l, \sigma', \mathtt{mod}\ l \leftarrow \mathtt{T}}\ \textbf{(mod)}$$

$$\frac{\begin{array}{c}\sigma, e \uparrow^{\mathtt{S}} \\ \sigma, e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}\end{array}}{\sigma, \mathtt{memo}_{\mathtt{S}}\ e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}}\ \textbf{(memo/miss)} \qquad \frac{\begin{array}{c}\sigma, e \downarrow^{\mathtt{S}} v, \mathtt{T} \\ \sigma, \mathtt{T} \overset{\mathtt{S}}{\curvearrowright} \sigma', \mathtt{T}'\end{array}}{\sigma, \mathtt{memo}_{\mathtt{S}}\ e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}'}\ \textbf{(memo/hit)}$$

$$\frac{v_1 = \mathtt{fun}_{\mathtt{S}}\ f(x)\ \mathtt{is}\ e \qquad \sigma, [v_1/f, v_2/x]\ e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}}{\sigma, \mathtt{apply}_{\mathtt{S}}(v_1, v_2) \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}}\ \textbf{(apply)}$$

$$\frac{\sigma, e_1 \Downarrow^{\mathtt{S}} v_1, \sigma_1, \mathtt{T}_1 \qquad \sigma_1, [v_1/x]\ e_2 \Downarrow^{\mathtt{S}} v_2, \sigma_2, \mathtt{T}_2 \qquad \mathtt{alloc}(\mathtt{T}_1) \cap \mathtt{alloc}(\mathtt{T}_2) = \emptyset}{\sigma, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow^{\mathtt{S}} v_2, \sigma_2, \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2}\ \textbf{(let)}$$

$$\frac{\sigma, [v_1/x_1, v_2/x_2]\ e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}}{\sigma, \mathtt{let}\, x_1 \times x_2 = (v_1, v_2)\, \mathtt{in}\, e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma, [v/x_1]\ e_1 \Downarrow^{\mathtt{S}} v', \sigma', \mathtt{T}}{\sigma, \mathtt{case}\, \mathtt{in}_{\mathtt{l}}\ v\, \mathtt{of}\ \{\mathtt{in}_{\mathtt{l}}(x_1) \Rightarrow e_1\ ,\ \mathtt{in}_{\mathtt{r}}(x_2) \Rightarrow e_2\} \Downarrow^{\mathtt{S}} v', \sigma', \mathtt{T}}\ \textbf{(case/inl)}$$

$$\frac{\sigma, [v/x_2]\ e_2 \Downarrow^{\mathtt{S}} v', \sigma', \mathtt{T}}{\sigma, \mathtt{case}\, \mathtt{in}_{\mathtt{r}}\ v\, \mathtt{of}\ \{\mathtt{in}_{\mathtt{l}}(x_1) \Rightarrow e_1\ ,\ \mathtt{in}_{\mathtt{r}}(x_2) \Rightarrow e_2\} \Downarrow^{\mathtt{S}} v', \sigma', \mathtt{T}}\ \textbf{(case/inr)}$$

Fig. 14: Evaluation of stable expressions.

be incorporated into the evaluation directly. Instead, the dynamic semantics performs a change propagation on the trace returned by the oracle before incorporating it into the current evaluation (this is described below).

Stable Evaluation. Figure 14 shows the evaluation rules for stable expressions. Most rules are standard for a store-passing semantics except that they also return traces. The interesting rules are those for `let`, `mod`, and `memo`.

The `let` rule sequences evaluation of its two expressions, performs binding by substitution, and yields a trace consisting of the sequential composition of the traces of its sub-expressions. For the traces to be well-formed, the rule requires that they allocate disjoint

$$\frac{}{\sigma, l \leftarrow \mathtt{write}(v) \Downarrow^{\mathsf{C}} \sigma[l \leftarrow v], \mathtt{write}\ v}\ \textbf{(write)}$$

$$\frac{\sigma, l \leftarrow [\sigma(l')/x]\ e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}{\sigma, l \leftarrow \mathtt{read}\ l'\ \mathtt{as}\ x\ \mathtt{in}\ e \Downarrow^{\mathsf{C}} \sigma', \mathtt{read}_{l' \to x = \sigma(l').e}\ \mathtt{T}}\ \textbf{(read)}$$

$$\frac{\begin{array}{c}\sigma, e \uparrow^{\mathsf{C}} \\ \sigma, e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}\end{array}}{\sigma, l \leftarrow \mathtt{memo}_{\mathsf{C}}\ e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}\ \textbf{(memo/miss)} \qquad \frac{\begin{array}{c}\sigma, e \downarrow^{\mathsf{C}} \mathtt{T} \\ \sigma, l \leftarrow \mathtt{T} \overset{\mathsf{C}}{\curvearrowright} \sigma', \mathtt{T}'\end{array}}{\sigma, l \leftarrow \mathtt{memo}_{\mathsf{C}}\ e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}'}\ \textbf{(memo/hit)}$$

$$\frac{v_1 = \mathtt{fun}_{\mathsf{C}}\ f(x)\ \mathtt{is}\ e \qquad \sigma, l \leftarrow [v_1/f, v_2/x]\ e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}{\sigma, l \leftarrow \mathtt{apply}_{\mathsf{C}}(v_1, v_2) \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}\ \textbf{(apply)}$$

$$\frac{\sigma, e_1 \Downarrow^{\mathsf{S}} v, \sigma_1, \mathtt{T}_1 \quad \sigma_1, l \leftarrow [v/x]\ e_2 \Downarrow^{\mathsf{C}} \sigma_2, \mathtt{T}_2 \quad \mathtt{alloc}(\mathtt{T}_1) \cap \mathtt{alloc}(\mathtt{T}_2) = \emptyset}{\sigma, l \leftarrow \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow^{\mathsf{C}} \sigma_2, \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2}\ \textbf{(let)}$$

$$\frac{\sigma, l \leftarrow [v_1/x_1, v_2/x_2]\ e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}{\sigma, l \leftarrow \mathtt{let} x_1 \times x_2 = (v_1, v_2)\, \mathtt{in} e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma, l \leftarrow [v/x_1]\ e_1 \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}{\sigma, l \leftarrow \mathtt{case}\, \mathtt{in}_{\mathtt{l}}\ v\, \mathtt{of}\ \{\mathtt{in}_{\mathtt{l}}\,(x_1) \Rightarrow e_1\ ,\mathtt{in}_{\mathtt{r}}\,(x_2) \Rightarrow e_2\} \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}\ \textbf{(case/inl)}$$

$$\frac{\sigma, l \leftarrow [v/x_2]\ e_2 \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}{\sigma, \mathtt{case}\, \mathtt{in}_{\mathtt{r}}\ v\, \mathtt{of}\ \{\mathtt{in}_{\mathtt{l}}\,(x_1) \Rightarrow e_1\ ,\mathtt{in}_{\mathtt{r}}\,(x_2) \Rightarrow e_2\} \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}}\ \textbf{(case/inr)}$$

Fig. 15: Evaluation of changeable expressions.

sets of locations. The `mod` rule allocates a location l, adds it to the store, and evaluates its body (a changeable expression) with l as the target. To ensure that l is not allocated multiple times, the rule requires that l is not allocated in the trace of the body. Note that the allocated location does not need to be fresh—it can already be in the store, i.e., $l \in \mathtt{dom}(\sigma)$. Since every changeable expression ends with a `write`, it is guaranteed that an allocated location is written before it can be read.

The `memo` rule consults an oracle to determine if its body should be evaluated or not. If the oracle returns a miss, then the body is evaluated as usual and the value, the store, and the trace obtained via evaluation is returned. If the oracle returns a hit, then it returns a value v and a trace $\mathtt{T}$. To adapt the trace to the current store σ, the evaluation performs a change propagation on $\mathtt{T}$ in σ and returns the value v returned by the oracle, and the trace and the store returned by change propagation. Note that since change propagation can change the contents of the store, it can also indirectly change the (lifted) contents of v. As an example, consider performing a deep change and updating the output as shown in Figure 4. Change propagation will only evaluate several calls deep in the DDG. The re-evaluated calls will update the list by writing to modifiables embedded deep in the output list, updating consequently the output list, i.e., its lifted value.

$$\frac{}{\sigma, \varepsilon \overset{\mathsf{s}}{\curvearrowright} \sigma, \varepsilon}\ \textbf{(empty)}$$

$$\frac{\begin{array}{ccc} l & \notin & \mathtt{alloc}(\mathtt{T}') \\ \sigma, l \leftarrow \mathtt{T} & \overset{\mathsf{c}}{\curvearrowright} & \sigma', \mathtt{T}' \end{array}}{\sigma, \mathtt{mod}\ l \leftarrow \mathtt{T} \overset{\mathsf{s}}{\curvearrowright} \sigma', \mathtt{mod}\ l \leftarrow \mathtt{T}'}\ \textbf{(mod)} \qquad \frac{}{\sigma, l \leftarrow \mathtt{write}\ v \overset{\mathsf{c}}{\curvearrowright} \sigma[l \leftarrow v], \mathtt{write}\ v}\ \textbf{(write)}$$

$$\frac{\begin{array}{ccc} \sigma, \mathtt{T}_1 & \overset{\mathsf{s}}{\curvearrowright} & \sigma', \mathtt{T}_1' \\ \sigma', \mathtt{T}_2 & \overset{\mathsf{s}}{\curvearrowright} & \sigma'', \mathtt{T}_2' \\ \multicolumn{3}{c}{\mathtt{alloc}\left(\mathtt{T}_1'\right) \cap \mathtt{alloc}\left(\mathtt{T}_2'\right) = \emptyset} \end{array}}{\sigma, \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2 \overset{\mathsf{s}}{\curvearrowright} \sigma'', \mathtt{let}\ \mathtt{T}_1'\ \mathtt{T}_2'}\ \textbf{(let/s)} \qquad \frac{\begin{array}{ccc} \sigma, \mathtt{T}_1 & \overset{\mathsf{c}}{\curvearrowright} & \sigma', \mathtt{T}_1' \\ \sigma', l \leftarrow \mathtt{T}_2 & \overset{\mathsf{c}}{\curvearrowright} & \sigma'', \mathtt{T}_2' \\ \multicolumn{3}{c}{\mathtt{alloc}\left(\mathtt{T}_1'\right) \cap \mathtt{alloc}\left(\mathtt{T}_2'\right) = \emptyset} \end{array}}{\sigma, l \leftarrow (\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2) \overset{\mathsf{c}}{\curvearrowright} \sigma'', (\mathtt{let}\ \mathtt{T}_1'\ \mathtt{T}_2')}\ \textbf{(let/c)}$$

$$\frac{\sigma(l') = v \qquad \sigma, l \leftarrow \mathtt{T} \overset{\mathsf{c}}{\curvearrowright} \sigma', \mathtt{T}'}{\sigma, l \leftarrow \mathtt{read}_{l' \to v = x.e}\ \mathtt{T} \overset{\mathsf{c}}{\curvearrowright} \sigma', \mathtt{read}_{l' \to v = x.e}\ \mathtt{T}'}\ \textbf{(read/no ch.)}$$

$$\frac{\sigma(l') \neq v \qquad \sigma, l \leftarrow [\sigma(l')/x]e \Downarrow^{\mathsf{c}} \sigma', \mathtt{T}'}{\sigma, l \leftarrow \mathtt{read}_{l' \to x = v.e}\ \mathtt{T} \overset{\mathsf{c}}{\curvearrowright} \sigma', \mathtt{read}_{l' \to x = \sigma(l').e}\ \mathtt{T}'}\ \textbf{(read/ch.)}$$

Fig. 16: Change propagation judgments.

Changeable Evaluation. Figure 15 shows the evaluation rules for changeable expressions. Evaluations in changeable mode perform *destination passing*. The `let`, `memo`, and `apply` rules are similar to the corresponding rules in stable mode except that the body of each expression is evaluated in changeable mode. The `read` expression substitutes the value stored at $\sigma(l')$ (the location being read) for the bound variable x in e and continues evaluation in changeable mode. A `read` is recorded in the trace, along with the value read, the variable bound, and the body of the read. A `write` simply assigns its argument to the target in the store. The evaluation of memoized changeable expressions is similar to that of stable expressions.

Change propagation. Figure 16 shows the rules for change propagation. As with evaluation rules, change-propagation rules are partitioned into stable and changeable, depending on the kind of the trace being processed. The stable change-propagation judgment $\sigma, \mathtt{T}_s \overset{\mathsf{s}}{\curvearrowright} \sigma', \mathtt{T}_s'$ states that change propagating into the stable trace $\mathtt{T}_s$ in the context of the store σ yields the store σ' and the stable trace $\mathtt{T}_s'$. The changeable change-propagation judgment $\sigma, l \leftarrow \mathtt{T}_c \overset{\mathsf{c}}{\curvearrowright} \sigma', \mathtt{T}_c'$ states that change propagation into the changeable trace $\mathtt{T}_c$ with target l in the context of the store σ yields the changeable trace $\mathtt{T}_c'$ and the store σ'. The change propagation rules mimic evaluation by either skipping over the parts of the trace that remain the same in the given store or by re-evaluating the `read`s that read locations whose values are different in the given store. The rules are labeled with the expression forms they mimic.

If the trace is empty, change propagation returns an empty trace and the same store. The `mod` rule recursively propagates into the trace `T` of the body to obtain a new trace

$\mathtt{T}'$ and returns a trace where $\mathtt{T}$ is substituted by $\mathtt{T}'$ under the condition that the target l is not allocated in $\mathtt{T}'$. This condition is necessary to ensure the allocation integrity of the returned trace. The stable `let` rule propagates into its two parts $\mathtt{T}_1$ and $\mathtt{T}_2$ recursively and returns a trace by combining the resulting traces $\mathtt{T}'_1$ and $\mathtt{T}'_2$, provided that the resulting trace ensures allocation integrity. The `write` rule performs the recorded write in the given store by extending the target with the value recorded in the trace. This is necessary to ensure that the result of a reused changeable computation is recorded in the new store. The `read` rule depends on whether the contents of the location l' being read are the same in the store as the value v recorded in the trace. If the contents are the same as in the trace, then change propagation proceeds into the body $\mathtt{T}$ of the read and the resulting trace is substituted for $\mathtt{T}$. Otherwise, the body of the `read` is evaluated with the specified target. Note that this makes evaluation and change-propagation mutually recursive—evaluation calls change-propagation in the case of an oracle hit. The changeable `let` rule is similar to the stable `let`.

Most change-propagation judgments perform some consistency checks or otherwise propagate forward. Only when change propagation finds that the a location `read` has changed, does it re-run the body of the `read` expression (a changeable computation) and replace the corresponding trace.

Evaluation invariants. Valid evaluations of stable and changeable expressions satisfy the following invariants:

1. All locations allocated in the trace are also allocated in the result store, i.e., if $\sigma, e \Downarrow^{\mathsf{S}}_{\text{ok}} v, \sigma', \mathtt{T}$ or $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}}_{\text{ok}} \sigma', \mathtt{T}$, then $\mathtt{dom}(\sigma') = \mathtt{dom}(\sigma) \cup \mathtt{alloc}(\mathtt{T})$.
2. For stable evaluations, any location whose content changes is allocated during that evaluation, i.e., if $\sigma, e \Downarrow^{\mathsf{S}}_{\text{ok}} v, \sigma', \mathtt{T}$ and $\sigma'(l) \neq \sigma(l)$, then $l \in \mathtt{alloc}(\mathtt{T})$.
3. For changeable evaluations, a location whose content changes is either the target or gets allocated during evaluation, i.e, if $\sigma, l' \leftarrow e \Downarrow^{\mathsf{C}}_{\text{ok}} \sigma', \mathtt{T}$ and $\sigma'(l) \neq \sigma(l)$, then $l \in \mathtt{alloc}(\mathtt{T}) \cup \{l'\}$.

Memo-free evaluations. The oracle rules introduce non-determinism into the dynamic semantics. Lemmas 5 and 6 in Section 4 express the fact that this non-determinism is harmless: change propagation will correctly update all answers returned by the oracle and make everything look as if the oracle never produced any answer at all (meaning that only **memo/miss** rules were used).

We write $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset} v, \sigma', \mathtt{T}$ or $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}}_{\emptyset} \sigma', \mathtt{T}$ if there is a derivation for $\sigma, e \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}$ or $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}$, respectively, that does not use any **memo/hit** rule. We call such an evaluation *memo-free*. We use $\Downarrow^{\mathsf{S}}_{\emptyset,\text{ok}}$ in place of $\Downarrow^{\mathsf{S}}_{\text{ok}}$ and $\Downarrow^{\mathsf{C}}_{\emptyset,\text{ok}}$ in place of $\Downarrow^{\mathsf{C}}_{\text{ok}}$ to indicate that a valid evaluation is also memo-free.

3.4 Deterministic, purely functional semantics

By ignoring memoization and change-propagation, we can give an alternative, purely functional, semantics for location-free AML programs, which we present in Figure 17. This semantics gives a store-free, pure, deterministic interpretation of AML that provides for no computation reuse. Under this semantics, both stable and changeable expressions evaluate to values, `memo`, `mod` and `write` are simply identities, and `read` acts as another binding

$$\frac{v \;\neq\; l}{v \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}\,\textbf{(value)} \qquad \frac{v = \mathtt{app}(o,(v_1,\ldots,v_n))}{o(v_1,\ldots,v_n) \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}\,\textbf{(prim.)} \qquad \frac{e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v}{\mathtt{mod}\; e \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}\,\textbf{(mod)}$$

$$\frac{e \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}{\mathtt{memo}_{\mathsf{S}}\; e \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}\,\textbf{(memo)} \qquad \frac{\begin{array}{c}(v_1 = \mathtt{fun}_{\mathsf{S}}\; f(x)\;\mathtt{is}\; e)\\ {[v_1/f, v_2/x]}\; e \Downarrow^{\mathsf{S}}_{\mathrm{det}} v\end{array}}{\mathtt{apply}_{\mathsf{S}}(v_1,v_2) \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}\,\textbf{(apply)}$$

$$\frac{\begin{array}{ccc}e_1 & \Downarrow^{\mathsf{S}}_{\mathrm{det}} & v_1\\ {[v_1/x]}\; e_2 & \Downarrow^{\mathsf{S}}_{\mathrm{det}} & v_2\end{array}}{\mathtt{let}\; x = e_1\;\mathtt{in}\; e_2 \Downarrow^{\mathsf{S}}_{\mathrm{det}} v_2}\,\textbf{(let)} \qquad \frac{[v_1/x_1, v_2/x_2]\; e \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}{\mathtt{let}\, x_1 \times x_2 \;=\; (v_1,v_2)\,\mathtt{in}\, e \Downarrow^{\mathsf{S}}_{\mathrm{det}} v}\,\textbf{(let}\times\textbf{)}$$

$$\frac{[v/x_1]\; e_1 \Downarrow^{\mathsf{S}}_{\mathrm{det}} v'}{\mathtt{case}\,\mathtt{in}_{\mathtt{l}}\; v\,\mathtt{of}\;\{\mathtt{in}_{\mathtt{l}}\,(x_1) \Rightarrow e_1\,,\,\mathtt{in}_{\mathtt{r}}\,(x_2) \Rightarrow e_2\} \Downarrow^{\mathsf{S}}_{\mathrm{det}} v'}\,\textbf{(case/inl)}$$

$$\frac{[v/x_2]\; e_2 \Downarrow^{\mathsf{S}}_{\mathrm{det}} v'}{\mathtt{case}\,\mathtt{in}_{\mathtt{r}}\; v\,\mathtt{of}\;\{\mathtt{in}_{\mathtt{l}}\,(x_1) \Rightarrow e_1\,,\,\mathtt{in}_{\mathtt{r}}\,(x_2) \Rightarrow e_2\} \Downarrow^{\mathsf{S}}_{\mathrm{det}} v'}\,\textbf{(case/inr)}$$

$$\frac{}{\mathtt{write}(v) \Downarrow^{\mathsf{C}}_{\mathrm{det}} v}\,\textbf{(write)} \qquad \frac{[v/x]\; e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v'}{\mathtt{read}\; v\;\mathtt{as}\; x\;\mathtt{in}\; e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v'}\,\textbf{(read)}$$

$$\frac{e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v}{\mathtt{memo}_{\mathsf{C}}\; e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v}\,\textbf{(memo)} \qquad \frac{\begin{array}{c}v_1 = \mathtt{fun}_{\mathsf{C}}\; f(x)\;\mathtt{is}\; e\\ {[v_1/f, v_2/x]}\; e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v\end{array}}{\mathtt{apply}_{\mathsf{C}}(v_1,v_2) \Downarrow^{\mathsf{C}}_{\mathrm{det}} v}\,\textbf{(apply)}$$

$$\frac{\begin{array}{ccc}e_1 & \Downarrow^{\mathsf{S}}_{\mathrm{det}} & v_1\\ {[v_1/x]}\; e_2 & \Downarrow^{\mathsf{C}}_{\mathrm{det}} & v_2\end{array}}{\mathtt{let}\; x = e_1\;\mathtt{in}\; e_2 \Downarrow^{\mathsf{C}}_{\mathrm{det}} v_2}\,\textbf{(let)} \qquad \frac{[v_1/x_1, v_2/x_2]\; e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v}{\mathtt{let}\, x_1 \times x_2 \;=\; (v_1,v_2)\,\mathtt{in}\, e \Downarrow^{\mathsf{C}}_{\mathrm{det}} v}\,\textbf{(let}\times\textbf{)}$$

$$\frac{[v/x_1]\; e_1 \Downarrow^{\mathsf{C}}_{\mathrm{det}} v'}{\mathtt{case}\,\mathtt{in}_{\mathtt{l}}\; v\,\mathtt{of}\;\{\mathtt{in}_{\mathtt{l}}\,(x_1) \Rightarrow e_1\,,\,\mathtt{in}_{\mathtt{r}}\,(x_2) \Rightarrow e_2\} \Downarrow^{\mathsf{C}}_{\mathrm{det}} v'}\,\textbf{(case/inl)}$$

$$\frac{[v/x_2]\; e_2 \Downarrow^{\mathsf{C}}_{\mathrm{det}} v'}{\mathtt{case}\,\mathtt{in}_{\mathtt{r}}\; v\,\mathtt{of}\;\{\mathtt{in}_{\mathtt{l}}\,(x_1) \Rightarrow e_1\,,\,\mathtt{in}_{\mathtt{r}}\,(x_2) \Rightarrow e_2\} \Downarrow^{\mathsf{C}}_{\mathrm{det}} v'}\,\textbf{(case/inr)}$$

Fig. 17: Purely functional semantics of stable (top) and changeable (bottom) expressions.

construct. Our correctness result states that the pure interpretation of AML yields results that are the same (up to lifting) as those obtained by AML's dynamic semantics (Section 4).

3.5 The `map` *example*

In Section 2 we used the `map` example to discuss the strengths and weaknesses of change propagation alone and described how our proposed techniques addresses these limitations. We briefly describe the relationship between the informal exposition used "there" in Section 2, and the formal treatment we presented "here" in this section. The traces presented

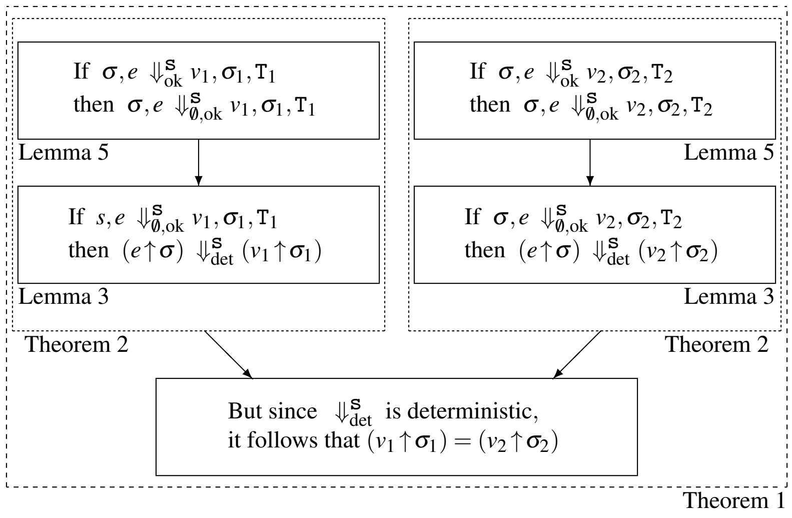
If $\sigma, e \Downarrow^{\mathsf{S}}_{\mathrm{ok}} v_1, \sigma_1, \mathtt{T}_1$ then $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset,\mathrm{ok}} v_1, \sigma_1, \mathtt{T}_1$
Lemma 5
If $s, e \Downarrow^{\mathsf{S}}_{\emptyset,\mathrm{ok}} v_1, \sigma_1, \mathtt{T}_1$ then $(e \uparrow \sigma) \Downarrow^{\mathsf{S}}_{\mathrm{det}} (v_1 \uparrow \sigma_1)$
Lemma 3
Theorem 2
If $\sigma, e \Downarrow^{\mathsf{S}}_{\mathrm{ok}} v_2, \sigma_2, \mathtt{T}_2$ then $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset,\mathrm{ok}} v_2, \sigma_2, \mathtt{T}_2$
Lemma 5
If $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset,\mathrm{ok}} v_2, \sigma_2, \mathtt{T}_2$ then $(e \uparrow \sigma) \Downarrow^{\mathsf{S}}_{\mathrm{det}} (v_2 \uparrow \sigma_2)$
Lemma 3
Theorem 2
But since $\Downarrow^{\mathsf{S}}_{\mathrm{det}}$ is deterministic, it follows that $(v_1 \uparrow \sigma_1) = (v_2 \uparrow \sigma_2)$
Theorem 1

Fig. 18: The structure of the proofs.

here make precise dynamic dependence graphs illustrated there. The change propagation judgments presented here make precise the change-propagation algorithm outlined there. The memoization oracle presented here make precise the idea of memoization returning computations as their dynamic dependency graphs instead of just results, by defining the oracle to return a trace not the result of an evaluation. The fact that the oracle does not care for the current store and just looks for a trace in an arbitrary store captures the idea that we are in an imperative setting, permitting the contents of the store to change.

As an example, consider evaluating `map [2,3,4]` and evaluating `map [1,2,3,4,5]` using our self-adjusting semantics as shown in Figure 8. Since the lists are actually represented as modifiable lists, and since `map` recursively walks down the list, when we call `map [2,3,4,5]` the oracle can return the trace for the evaluation `map [2,3,4]` because the expressions actually match up to the contents of the last cell. The trace returned by the oracle corresponds to the dynamic dependency graph returned by the memoization mechanism described in the overview section. Next, the semantics fires change propagation on the re-used trace in the context of the current store. Since the modifiable holding the last element is changed, change propagation fires `map [5]` recursively, mapping `5` in the output, and recursively calling `map []`, which would be matched trivially by the oracle. Since there are no further changes, the semantics yields the updated trace and the updated memory which contains the output for the changed list.

4 Consistency and Correctness

We now state consistency and correctness theorems for AML and outline their proofs in terms of several main lemmas. As depicted in Figure 18, consistency (Theorem 1) is a consequence of correctness (Theorem 2).

4.1 Main theorems

Consistency uses *structural equality* based on the notion of *lifts* (see Section 3.2) to compare the results of two potentially different evaluations of the same AML program under its non-deterministic semantics. Correctness, on the other hand, compares one such evaluation to a pure, functional evaluation. It justifies the claim that even with stores, memoization, and change propagation, AML is essentially a purely functional language.

Theorem 1 (Consistency)
If $\sigma, e \Downarrow^{\mathsf{S}}_{\text{ok}} v_1, \sigma_1, T_1$ *and* $\sigma, e \Downarrow^{\mathsf{S}}_{\text{ok}} v_2, \sigma_2, T_2$*, then* $v_1 \uparrow \sigma_1 = v_2 \uparrow \sigma_2$.

Theorem 2 (Correctness)
If $\sigma, e \Downarrow^{\mathsf{S}}_{\text{ok}} v, \sigma', T$*, then* $(e \uparrow \sigma) \Downarrow^{\mathsf{S}}_{\text{det}} (v \uparrow \sigma')$.

Recall that by our convention the use of the notation $v \uparrow \sigma$ implies well-formedness of v in σ. Therefore, part of the statement of consistency and correctness is the preservation of well-formedness during evaluation, and the inability of AML programs to create cyclic memory graphs.

4.2 Proof outline

The consistency theorem is proved in two steps. First, Lemmas 3 and 4 state that consistency is true in the restricted setting where all evaluations are memo-free.

Lemma 3 (purity/st.)
If $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset,\text{ok}} v, \sigma', T$*, then* $(e \uparrow \sigma) \Downarrow^{\mathsf{S}}_{\text{det}} (v \uparrow \sigma')$.

Lemma 4 (purity/ch.)
If $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}}_{\emptyset,\text{ok}} \sigma', T$*, then* $(e \uparrow \sigma) \Downarrow^{\mathsf{C}}_{\text{det}} (l \uparrow \sigma')$.

Second, Lemmas 5 and 6 state that for any evaluation there is a memo-free counterpart that yields an *identical* result and has *identical* effects on the store. Notice that this is stronger than saying that the memo-free evaluation is "equivalent" in some sense (e.g., under lifts). The statements of these lemmas are actually even stronger since they include a "preservation of well-formedness" statement. Preservation of well-formedness is required in the inductive proof.

Lemma 5 (memo-freedom/st.)
If $\sigma, e \Downarrow^{\mathsf{S}}_{\text{ok}} v, \sigma', T$*, then* $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset} v, \sigma', T$ *where* $\mathtt{reach}(v, \sigma') \subseteq \mathtt{reach}(e, \sigma) \cup \mathtt{alloc}(T)$.

Lemma 6 (memo-freedom/ch.)
If $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}}_{\text{ok}} \sigma', T$*, then* $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}}_{\emptyset} \sigma', T$ *where* $\mathtt{reach}(\sigma'(l), \sigma') \subseteq \mathtt{reach}(e, \sigma) \cup \mathtt{alloc}(T)$.

The proof for Lemmas 5 and 6 proceeds by simultaneous induction over the expression e. It is outlined in far more detail in Section 5. Both lemmas state that if there is a well-formed evaluation leading to a store, a trace, and a result (the value v in the stable lemma, or the target l in the changeable lemma), the same result (which will be well-formed itself) is obtainable by a memo-free run. Moreover, all locations reachable from the result were either reachable from the initial expression or were allocated during the evaluation. These conditions help to re-establish well-formedness in inductive steps.

The lemmas are true thanks to a key property of the dynamic semantics: allocated locations need not be completely "fresh" in the sense that they may be in the current store as long as they are neither reachable from the initial expression nor get allocated multiple times. This means that a location that is already in the store can be chosen for reuse by the `mod` expression (Figure 14). To see why this is important, consider as an example the expression: $\mathtt{memo}_{\mathtt{S}}\ (\mathtt{mod}\ (\mathtt{write(3)}))$ in σ. Suppose now that the oracle returns the value l and the trace T_0: $\sigma_0, \mathtt{mod}\ (\mathtt{write(3)})\ \Downarrow^{\mathtt{S}} l, \sigma_0', \mathtt{T}_0$. Even if $l \in \mathtt{dom}(\sigma)$, change propagation will simply update the store as $\sigma[l \leftarrow 3]$ and return l. In a memo-free evaluation of the same expression the oracle misses, and `mod` must allocate a location. Thus, if the evaluation of `mod` were restricted to using fresh locations only, it would allocate some $l' \notin \mathtt{dom}(\sigma)$, and return that. But since $l \in \mathtt{dom}(\sigma)$, $l \neq l'$.

5 The Proofs

This section presents a proof sketch for the four memo-elimination lemmas as well as the two lemmas comparing AML's dynamic semantics to the pure semantics (Section 4). We give a detailed analysis for the most difficult cases. These proofs have all been formalized and machine-checked in Twelf (see Section 6).

5.1 Proofs for memo-elimination

Informally speaking, the proofs for Lemmas 5 and 6, as well as Lemmas 8 and 9 all proceed by simultaneous induction on the derivations of the respective *result* evaluation judgments. The imprecision in this statement stems from the fact that, as we will see, there are instances where we use the induction hypothesis on something that is not really a sub-derivation of the given derivation. For this reason, a full formalization of the proof defines a metric on derivations which demonstrably decreases on each inductive step. Section 6 discusses the Twelf formalization in more detail.

5.1.1 Substitution

We will frequently appeal to the following *substitution lemma*. It states that well-formedness and lifts of expressions are preserved under substitution:

Lemma 7 (Substitution)
If $e, \sigma \xrightarrow{\mathtt{wf}} e', L$ *and* $v, \sigma \xrightarrow{\mathtt{wf}} v', L'$*, then* $[v/x]\ e, \sigma \xrightarrow{\mathtt{wf}} [v'/x]\ e', L''$ *with* $L'' \subseteq L \cup L'$*.*

The proof of the lemma proceeds by induction on the structure of e.

5.1.2 Hit-elimination lemmas

Since the cases for the **memo/hit** rules involve many sub-cases, it is instructive to separate these out into separate lemmas:

Lemma 8 (hit-elimination/stable)
If $\sigma_0, e \Downarrow^{\mathsf{S}}_{\text{ok}} v, \sigma'_0, T_0$ *and* $\sigma, T_0 \overset{\mathsf{S}}{\curvearrowright} \sigma', T$ *where* $\mathtt{reach}(e,\sigma) \cap \mathtt{alloc}(T) = \emptyset$,
then $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset} v, \sigma', T$ *with* $\mathtt{reach}(v,\sigma') \subseteq \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(T)$.

Lemma 9 (hit-elimination/changeable)
If $\sigma_0, l_0 \leftarrow e \Downarrow^{\mathsf{C}}_{\text{ok}} \sigma'_0, T_0$ *and* $\sigma, l \leftarrow T_0 \overset{\mathsf{C}}{\curvearrowright} \sigma', T$ *where* $\mathtt{reach}(e,\sigma) \cap \mathtt{alloc}(T) = \emptyset$ *and* $l \notin \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(T)$,
then $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}}_{\emptyset} \sigma', T$ *with* $\mathtt{reach}(\sigma'(l),\sigma') \subseteq \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(T)$.

5.1.3 Proof sketch for Lemma 5 (stable memo-freedom)

For the remainder of the current section we will ignore the added complexity caused by the need for a decreasing metric on derivations. Here is a sketch of the cases that need to be considered in the part of the proof that deals with Lemma 5:

- **value:** Since the expression itself is the value, with the trace being empty, this case is trivial.
- **primitives:** The case for primitive operations goes through straightforwardly using preservation of well-formedness.
- **mod:** Given $\sigma, \mathtt{mod}\ e \Downarrow^{\mathsf{S}}_{\text{ok}} l, \sigma', \mathtt{mod}\ l \leftarrow \mathtt{T}$ we have

  $$\mathtt{reach}(\mathtt{mod}\ e, \sigma) \cap \mathtt{alloc}(\mathtt{mod}\ l \leftarrow \mathtt{T}) = \emptyset.$$

  This implies that $l \notin \mathtt{reach}(\mathtt{mod}\ e, \sigma)$. By the evaluation rule **mod** it is also true that $\sigma, e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}$ and $l \notin \mathtt{alloc}(\mathtt{T})$. By definition of `reach` and `alloc` we also know that $\mathtt{reach}(e,\sigma) \cap \mathtt{alloc}(\mathtt{T}) = \emptyset$, implying $\sigma, e \Downarrow^{\mathsf{C}}_{\text{ok}} \sigma', \mathtt{T}$.
  By induction (using Lemma 6) we get $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}}_{\emptyset} \sigma', \mathtt{T}$ with $\mathtt{reach}(\sigma'(l),\sigma') \subseteq \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(\mathtt{T})$. Since l is the final result, we find that

  $$\begin{array}{rcl} \mathtt{reach}\left(l,\sigma'\right) & = & \mathtt{reach}\left(\sigma'(l),\sigma'\right) \cup \{l\} \\ & \subseteq & \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(\mathtt{T}) \cup \{l\} \\ & = & \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(\mathtt{mod}\ l \leftarrow \mathtt{T}). \end{array}$$

- **memo/hit:** Since the result evaluation is supposed to be memo-free, there really is no use of the **memo/hit** rule there. However, a **memo/miss** in the memo-free trace can be the result of eliminating a **memo/hit** in the original run. We refer to this situation here, which really is the heart of the matter: a use of the **memo/hit** rule for which we have to show that we can eliminate it in favor of some memo-free evaluation. This case has been factored out as a separate lemma (Lemma 8), which we can use here inductively.
- **memo/miss** The case of a retained **memo/miss** is completely straightforward, using the induction hypothesis (Lemma 5) on the subexpression e in $\mathtt{mod}\ e$.
- **let** The difficulty here is to establish that the second part of the evaluation is valid. Given

  $$\sigma, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow^{\mathsf{S}}_{\text{ok}} v_2, \sigma'', \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2$$

  we have $L \cap \mathtt{alloc}(\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2) = \emptyset$
  where $L = \mathtt{reach}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, \sigma)$.

By the evaluation rule **let** it is the case that $\sigma, e_1 \Downarrow^{\mathsf{S}} v_1, \sigma', \mathtt{T}_1$ where $\mathtt{alloc}(\mathtt{T}_1) \subseteq \mathtt{alloc}(\mathtt{T})$. Well-formedness of the whole expression implies well-formedness of each of its parts, so $\mathtt{reach}(e_1, \sigma) \subseteq L$ and $\mathtt{reach}(e_2, \sigma) \subseteq L$. This means that $\mathtt{reach}(e_1, \sigma) \cap \mathtt{alloc}(\mathtt{T}_1) = \emptyset$, so $\sigma, e_1 \Downarrow^{\mathsf{S}}_{\mathrm{ok}} v_1, \sigma', \mathtt{T}_1$. Using the induction hypothesis (Lemma 5) this implies

$$\sigma, e_1 \Downarrow^{\mathsf{S}}_{\emptyset} v_1, \sigma', \mathtt{T}_1$$

and $\mathtt{reach}(v_1, \sigma') \subseteq \mathtt{reach}(e_1, \sigma) \cup \mathtt{alloc}(\mathtt{T}_1)$.
Since $\mathtt{reach}(e_2, \sigma) \subseteq L$ we have $\mathtt{reach}(e_2, \sigma) \cap \mathtt{alloc}(\mathtt{T}_1) = \emptyset$. Store σ' is equal to σ up to $\mathtt{alloc}(\mathtt{T}_1)$, so $\mathtt{reach}(e_2, \sigma) = \mathtt{reach}(e_2, \sigma')$. Therefore, by substitution (Lemma 7) we get

$$\begin{array}{rcl}
\mathtt{reach}\left([v_1/x]\, e_2, \sigma'\right) & \subseteq & \mathtt{reach}\left(e_2, \sigma'\right) \cup \mathtt{reach}\left(v_1, \sigma'\right) \\
& \subseteq & \mathtt{reach}(e_2, \sigma) \cup \mathtt{reach}\left(v_1, \sigma'\right) \\
& \subseteq & \mathtt{reach}(e_2, \sigma) \cup \mathtt{reach}(e_1, \sigma) \\
& & \cup \mathtt{alloc}(\mathtt{T}_1) \\
& = & L \cup \mathtt{alloc}(\mathtt{T}_1)
\end{array}$$

Since $\mathtt{alloc}(\mathtt{T}_2)$ is disjoint from both L and $\mathtt{alloc}(\mathtt{T}_1)$, this means that $\sigma', [v_1/x]\, e_2 \Downarrow^{\mathsf{S}}_{\mathrm{ok}} v_2, \sigma'', \mathtt{T}_2$. Using the induction hypothesis (Lemma 5) a second time we get

$$\sigma', [v_1/x]\, e_2 \Downarrow^{\mathsf{S}}_{\emptyset} v_2, \sigma'', \mathtt{T}_2,$$

so by definition

$$\sigma, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow^{\mathsf{S}}_{\emptyset} v_2, \sigma'', \mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2.$$

It is then also true that

$$\begin{array}{rcl}
\mathtt{reach}\left(v_2, \sigma''\right) & \subseteq & \mathtt{reach}\left([v_1/x]\, e_2, \sigma'\right) \cup \mathtt{alloc}(\mathtt{T}_2) \\
& \subseteq & L \cup \mathtt{alloc}(\mathtt{T}_1) \cup \mathtt{alloc}(\mathtt{T}_2) \\
& = & L \cup \mathtt{alloc}(\mathtt{let}\ \mathtt{T}_1\ \mathtt{T}_2),
\end{array}$$

which concludes the argument.

The remaining cases all follow by a straightforward application of Lemma 7 (substitution), followed by the use of the induction hypothesis (Lemma 5).

5.1.4 *Proof sketch for Lemma 6 (Changeable memo-freedom)*

- **write:** Given $\sigma, l \leftarrow \mathtt{write}(v) \Downarrow^{\mathsf{C}}_{\mathrm{ok}} \sigma[l \leftarrow v], \mathtt{write}\ v$ we clearly also have $\sigma, l \leftarrow \mathtt{write}(v) \Downarrow^{\mathsf{C}}_{\emptyset} \sigma[l \leftarrow v], \mathtt{write}\ v$. First we need to show that $\sigma'(l)$ is well-formed in $s' = \sigma[l \leftarrow v]$. This is true because $\sigma'(l) = v$ and l is not reachable from v in σ, so the update to l cannot create a cycle. Moreover, this means that the locations reachable from v in σ' are the same as the ones reachable in σ, i.e., $\mathtt{reach}(v, \sigma) = \mathtt{reach}(v, \sigma')$. Since nothing is allocated, $\mathtt{alloc}(\mathtt{write}\ v) = \emptyset$, it follows that $\mathtt{reach}(\sigma'(l), \sigma') \subseteq \mathtt{reach}(v, \sigma) \cup \mathtt{alloc}(\mathtt{write}\ v)$.

- **read:** For the case of $\sigma, l \leftarrow \texttt{read}\, l' \,\texttt{as}\, x \,\texttt{in}\, e \Downarrow^{\mathtt{C}}_{\mathrm{ok}} \sigma', \mathtt{T}$ we observe that by definition of well-formedness $\sigma(l')$ is also well-formed in σ. From here the proof proceeds by an application of the substitution lemma, followed by a use of the induction hypothesis (Lemma 6).

- **memo/hit:** Again, this is the case of a **memo/miss** which is the result of eliminating the presence of a **memo/hit** in the original evaluation. Like in the stable setting, we have factored this out as a separate lemma (Lemma 9).

- **memo/miss:** As before, the case of a retained use of **memo/miss** is handled by straightforward use of the induction hypothesis (Lemma 6).

- **let:** The proof for the **let** case in the changeable setting is tedious but straightforward and proceeds along the lines of the proof for the **let** case in the stable setting. Lemma 5 is used inductively for the first sub-expression, Lemma 6 for the second (after establishing validity using the substitution lemma).

The remaining cases follow by application of the substitution lemma and the use of the induction hypothesis (Lemma 6).

5.1.5 Proof of Lemma 8 (stable hit-elimination)

- **value:** Immediate.
- **primitives:** Immediate.
- **mod:** The case of **mod** requires some attention, since the location being allocated may already be present in σ, a situation which, however, is tolerated by our relaxed evaluation rule for $\texttt{mod}\, e$. We show the proof in detail, using the following calculations which establishe the conclusions (lines $(16,19)$) from the preconditions (lines $(1,2,3)$):

(1)	$\sigma_0, \mathtt{mod}\ e \ \Downarrow^{\mathtt{S}}_{\mathrm{ok}}\ l, \sigma_0', \mathtt{mod}\ l \leftarrow \mathtt{T}_0$
(2)	$\sigma, \mathtt{mod}\ l \leftarrow \mathtt{T}_0 \ \overset{\mathtt{S}}{\curvearrowright}\ \sigma', \mathtt{mod}\ l \leftarrow \mathtt{T}$
(3)	$\mathtt{reach}(e,\sigma) \cap \mathtt{alloc}(\mathtt{T}) = \emptyset$ $l \notin \mathtt{alloc}(\mathtt{T}) \cup \mathtt{reach}(e,\sigma)$
(4) by (1)	$\sigma_0, l \leftarrow e \ \Downarrow^{\mathtt{C}}\ \sigma_0', \mathtt{T}_0$
(5) by (1)	$\mathtt{alloc}(\mathtt{mod}\ l \leftarrow \mathtt{T}_0) \cap \mathtt{reach}(e,\sigma_0) = \emptyset$
(6) by (5)	$\mathtt{alloc}(\mathtt{T}_0) \cap \mathtt{reach}(e,\sigma_0) = \emptyset$
(7) by (5)	$l \notin \mathtt{reach}(e,\sigma_0)$
(8) by (1), **mod**	$l \notin \mathtt{alloc}(\mathtt{T}_0)$
(9) by (4, 6, 7, 8)	$\sigma_0, l \leftarrow e \ \Downarrow^{\mathtt{C}}_{\mathrm{ok}}\ \sigma_0', \mathtt{T}_0$
(10) by (2), $\mathtt{mod}$	$\sigma, l \leftarrow \mathtt{T}_0 \ \overset{\mathtt{C}}{\curvearrowright}\ \sigma', \mathtt{T}$
(11) by (3)	$\mathtt{reach}(e,\sigma) \cap \mathtt{alloc}(\mathtt{T}) = \emptyset$
(12) by (3)	$l \notin \mathtt{reach}(e,\sigma)$
(13) by (3)	$l \notin \mathtt{alloc}(\mathtt{T})$
(14) by $(9-13)$, IH	$\sigma, l \leftarrow e \ \Downarrow^{\mathtt{C}}_{\emptyset}\ \sigma', \mathtt{T}$
(15) by $(9-13)$, IH	$\mathtt{reach}(\sigma'(l),\sigma') \subseteq \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(\mathtt{T})$
(16) by (8, 14), **mod**	$\sigma, \mathtt{mod}\ e \ \Downarrow^{\mathtt{S}}_{\emptyset}\ l, \sigma', \mathtt{mod}\ l \leftarrow \mathtt{T}$
(17) by (7, 8, 15)	$l \notin \mathtt{reach}(\sigma'(l),\sigma')$
(18) by (17)	$\mathtt{reach}(l,\sigma') = \mathtt{reach}(\sigma'(l),\sigma') \cup \{l\}$
(19) by (15, 18)	$\mathtt{reach}(l,\sigma') \subseteq \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(\mathtt{T}) \cup \{l\}$ $= \mathtt{reach}(e,\sigma) \cup \mathtt{alloc}(\mathtt{mod}\ l \leftarrow \mathtt{T})$

- **memo/hit:** This case is proved by two consecutive applications of the induction hypothesis, one time to obtain a memo-free version of the original evaluation $\sigma_0, e \Downarrow^{\mathtt{S}}_{\emptyset} v, \sigma_0', \mathtt{T}_0$, and then starting from that the memo-free final result.
  It is here where straightforward induction on the derivation breaks down, since the derivation of the memo-free version of the original evaluation is not a sub-derivation of the overall derivation. In the formalized and proof-checked version (Section 6) this is handled using an auxiliary metric on derivations.
- **memo/miss:** This is the case in which the original evaluation of $\mathtt{memo}_{\mathtt{S}}\ e$ did not use the oracle and evaluated e directly. We prove the result by applying the induction hypothesis (Lemma 8).
- **let:** We consider the evaluation of $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$. Again, the main challenge here is to establish that the evaluation of $[v_1/x]\ e$, where v_1 is the result of e_1, is well-formed. The argument is tedious but straightforward and proceeds much like that in the proof of Lemma 5.

All remaining cases are handled simply by applying the substitution lemma (Lemma 7) and then using the induction hypothesis (Lemma 8).

5.1.6 Proof of Lemma 9 (changeable hit-elimination)

- **write:** We have $e = \texttt{write}(v)$ and $\texttt{T}_0 = \texttt{T} = \texttt{write}\ v$. Therefore, trivially, $\sigma, l \leftarrow e \Downarrow^{\texttt{C}}_{\emptyset} \sigma', \texttt{T}$ with $\sigma' = \sigma[l \leftarrow v]$. Also, $\texttt{reach}(\texttt{write}(v), \sigma) = \texttt{reach}(v, \sigma) = L$. Therefore, $\texttt{reach}(\sigma'(l), \sigma') = L$ because $l \notin L$. Of course, $L \subseteq L \cup \texttt{alloc}(\texttt{T})$.
- **read/no ch.:** We handle **read** in two parts. The first part deals with the situation where there is no change to the location that has been read. In this case we apply the substitution lemma to establish the preconditions for the induction hypothesis and conclude using Lemma 9.
- **read/ch.:** If change propagation detects that the location being read contains a new value, it re-executes the body of $\texttt{read}\ l'\ \texttt{as}\ x\ \texttt{in}\ e$. Using substitution we establish the pre-conditions of Lemma 6 and conclude by using the induction hypothesis.
- **memo/hit:** Like in the proof for Lemma 8, the **memo/hit** case is handled by two cascading applications of the induction hypothesis (Lemma 9).
- **memo/miss:** Again, the case where the original evaluation did not get an answer from the oracle is handled easily by using the induction hypothesis (Lemma 9).
- **let:** We consider the evaluation of $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. As before, the challenge is to establish that the evaluation of $[v_1/x]\ e$, where v_1 is the (stable) result of e_1, is well-formed. The argument is tedious but straightforward and proceeds much like that in the proof of Lemma 6.

All remaining cases are handled by the induction hypothesis (Lemma 9), which becomes applicable after establishing validity using the substitution lemma.

5.2 Proofs for equivalence to pure semantics

The proofs for Lemmas 3 and 4 proceed by simultaneous induction on the derivation of the memo-free evaluation. The following two subsections outline the two major parts of the case analysis.

5.2.1 Proof sketch for Lemma 3 (stable evaluation)

We proceed by considering each possible stable evaluation rule:

- **value:** Immediate.
- **primitives:** Using the condition on primitive operations that they commute with lifts, this is immediate.
- **mod:** Consider $\texttt{mod}\ e_c$. The induction hypothesis (Lemma 4) on the evaluation of e_c directly gives the required result.
- **memo:** Since we consider memo-free evaluations, we only need to consider the use of the **memo/miss** rule. The result follows by direct application of the induction hypothesis (Lemma 3).
- **let:** We have $\sigma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow^{\texttt{S}}_{\emptyset} v_2, \sigma'', \texttt{let}\ \texttt{T}_1\ \texttt{T}_2$. Because of validity of the original evaluation, we also have $\texttt{let}\ x = e_1\ \texttt{in}\ e_2, \sigma \xrightarrow{\texttt{wf}} L$, where

$L \cap \texttt{alloc}(\texttt{let T}_1\ \texttt{T}_2) = \emptyset$. Therefore, $\sigma, e_1 \Downarrow^{\texttt{S}}_{\emptyset} v_1, \sigma', \texttt{T}_1$ where $e_1, \sigma \xrightarrow{\texttt{wf}} L_1$ and $L_1 \cap \texttt{alloc}(\texttt{T}) = \emptyset$ because $L_1 \subseteq L$ and $\texttt{alloc}(\texttt{T}_1) \subseteq \texttt{alloc}(\texttt{let T}_1\ \texttt{T}_2)$. By induction hypothesis (Lemma 3) we get $(e_1 \uparrow \sigma) \Downarrow^{\texttt{S}}_{\text{det}} (v_1 \uparrow \sigma')$.

We can establish validity for $\sigma', [v_1/x]\ e_2 \Downarrow^{\texttt{S}}_{\emptyset} v_2, \sigma'', \texttt{T}_2$ the same way we did in the proof of Lemma 5, so by a second application of the induction hypothesis we get $([v_1/x]\ e_2 \uparrow \sigma') \Downarrow^{\texttt{S}}_{\text{det}} (v_2 \uparrow \sigma'')$. But by substitution (Lemma 7) we have $([v_1/x]\ e_2) \uparrow \sigma' = [(v_1 \uparrow \sigma')/x]\ (e_2 \uparrow \sigma')$. Using the evaluation rule **let/p** this gives the desired result.

The remaining cases follow straightforwardly by applying the induction hypothesis (Lemma 3) after establishing validity using the substitution lemma.

5.2.2 Proof sketch for Lemma 4 (changeable evaluation)

Here we consider each possible changeable evaluation rule:

- **write:** Immediate by the definition of lift.
- **read:** Using the definition of lift and the substitution lemma, this follows by an application of the induction hypothesis (Lemma 4).
- **memo:** Like in the stable setting, this case is handled by straightforward application of the induction hypothesis, because no memo hit needs to be considered.
- **let:** The let case is again somewhat tedious. It proceeds by first using the induction hypothesis (Lemma 3) on the stable sub-expression, then re-establishing validity using the substitution lemma, and finally applying the induction hypothesis a second time (this time in form of Lemma 4).

All other cases are handled by an application of the induction hypothesis (Lemma 4) after establishing validity using the substitution lemma.

6 Mechanization in Twelf

To increase our confidence in the proofs for the correctness and the consistency theorems, we have encoded the AML language and the proofs in Twelf [Pfenning and Schürmann 1999] and machine-checked the proofs. We follow the standard *judgments as types* methodology [Harper et al. 1993], and check our theorems using the Twelf metatheorem checker. For full details on using Twelf in this way for proofs about programming languages, see Harper and Licata's paper [Harper and Licata 2007].

The LF encoding of the syntax and semantics of AML corresponds very closely to the paper judgments. (In an informal sense; we have not proved formally that the LF encoding is *adequate*, and take adequacy to be evident.) However, in a few cases we have altered the judgments, driven by the needs of the mechanized proof. For example, on paper we write memo-free and general evaluations as different judgments, and silently coerce memo-free to general evaluations in the proof. We could represent the two judgments by separate LF type families, but the proof would then require a lemma to convert one judgment to the other. Instead, we define a type family to represent general evaluations, and a separate type family, indexed by evaluation derivations, to represent the judgment that an evaluation derivation is memo-free.

The proof of consistency (a metatheorem in Twelf) corresponds closely to the paper proof in overall structure. The proof of memo-freedom consists of four mutually-inductive lemmas: memo-freedom for stable and changeable expressions (Lemma 5 and Lemma 6), and versions of these with an additional change propagation following the evaluation (needed for the hit cases). In the hit cases for these latter lemmas, we must eliminate two change propagations: we call the lemma once to eliminate the first, then a second time on the output of the first call to eliminate the second. Since the evaluation in the second call is not a subderivation of the input, we must give a separate termination metric. The metric is defined on evaluation derivations and simply counts the number of evaluations in the derivations, including those inside of change propagations. In an evaluation which contains change propagations, there are "garbage" evaluations which are removed during hit-elimination. Therefore, hit-elimination reduces this metric (or keeps it the same, if there were no change propagations to remove). We add arguments to the lemmas to account for the metric, and simultaneously prove that the metric is smaller in each inductive call, in order for Twelf to check termination.

Aside from this structural difference due to termination checking, the main difference from the paper proof is that the Twelf proof must of course spell out all the details which the paper proof leaves to the reader to verify. In particular, we must encode "background" structures such as finite sets of locations, and prove relevant properties of such structures. While we are not the first to use these structures in Twelf, we have found it difficult to reuse libraries at present due to poor library support for Twelf. Our needs are also somewhat specialized: because we need to prove properties about stores which differ only on a set of locations, it is convenient to encode stores and location sets in a slightly unusual way: location sets are represented as lists of bits, and stores are represented as lists of value options; in both representations the *n*th list element corresponds to the *n*th location. This makes it easy to prove the necessary lemmas by parallel induction over the lists.

The full Twelf code for the proofs (as a tar archive) can be found at the url `http://www.umut-acar.org/publications/jfp2013-twelf-proof.tar`. The Twelf proof is also reachable via the first author's web page at `http://www.umut-acar.org`. The Twelf code archive consists of a number of individual Twelf files. In the rest of this section, we present some more detail on how we mechanized the semantics the proof in Twelf, which we hope will guide the interested reader in understanding the Twelf code. When referring to the Twelf code, we mention the names of Twelf files that contains the code fragments of interest. For improved readability we typeset the left and right arrows in text mode with rendered arrows.

6.1 The syntax and the semantics

The language syntax (Figure 9) is given in the Twelf file `syntax.elf`, type families `val`, `es`, and `ec`. They follow the standard Twelf approach to encoding syntax ([Harper and Licata 2007]); in particular the binding forms are encoded using higher-order abstract syntax.

Traces (Section 3.3) are given in `trace.elf`, type families `trs` and `trc`. This is again a standard syntax encoding. This file also contains the allocated locations of a trace (type families `trs-gen` and `trc-gen`).

Well-formed expressions (Figure 11) are given in `wf-ex.elf`, type families `wf-val`, `wf-es`, and `wf-ec`. For example, the rule for well-formedness of a location (upper right in Figure 11) is

```
wf-val-loc :wf-val (val-loc L) S V' X1+X2
              ← st-lookup S L V
              ← wf-val V S V' X1
              ← ls-sing L X2
              ← ls-union X1 X2 X1+X2.
```

This rule says that a `(val-loc L)` (that is, a location viewed as a value) is well-formed in store `S`, lifts to `V'`, and reaches locations `X1+X2`, provided that

- `L` is bound to `V` in `S`
- `V` is well-formed in `S`, lifts to `V'`, and reaches locations `X1`
- `L` as a singleton set is `X2`
- and the union of `X1` and `X2` is `X1+X2`.

In other words, modulo renaming of variables and expanding some notational shorthand, this says exactly what the paper rule says.

The well-formedness rule for stable `let` illustrates working with higher-order syntax in Twelf:

```
wf-es-let :wf-es (es-let Es1 Es2) S (es-let Es1' Es2') X
             ← wf-es Es1 S Es1' X1
             ← ({v}{d : var v} wf-es (Es2 v) S (Es2' v) X2)
             ← ls-union X1 X2 X.
```

This rule says that `(es-let Es1 Es2)` is well-formed in store `S`, lifts to `(es-let Es1' Es2')`, and reaches locations `X`, provided that:

- `Es1` is well-formed in `S`, lifts to `Es1'`, reaches `X1`
- `Es2` is well-formed in `S`, lifts to `Es2'`, reaches `X2`
- the union of `X1` and `X2` is `X`.

The `{v}{d: var v}` deals with a technical detail of higher-order abstract syntax: it cannot directly represent expressions with free variables. The expression `Es2` may contain `x` as a free variable, but it is encoded as a lambda expression, so `(wf-es Es2 [...])` is not well-typed. To get around this, we assume that there is some value `v` (writing `{v}` or equivalently `{v : val}`) and ”tag” it as a variable—we assume a derivation `d` of the judgment `(var v)` (writing `{d : var v}`); we apply `Es2` to `v` (the Twelf way to say `[v/x] Es2`); and we include a well-formedness rule to treat variables, which corresponds to the paper rule:

```
wf-val-var :wf-val V S V ls-nil
              ← var V.
```

This rule says that an arbitrary value `V` is well-formed in store `S`, lifts to `V`, and reaches the empty set of locations, provided that it is tagged as a variable. This is a standard Twelf technique for treating the variable case in paper definitions.

Evaluation (Figures 14 and 15) are given in `eval.elf`, type families `evals` and `evalc`; change propagation (Figure 16) is also in `eval.elf`, type families `cps` and `cpc`.

A complicated-looking rule on paper is the `let` rule for stable expressions; in Twelf this is encoded as

```
evals-let :evals S (es-let Es1 Es2) V2 S2 (trs-let Ts1 Ts2)
            ← evals S Es1 V1 S1 Ts1
            ← evals S1 (Es2 V1) V2 S2 Ts2
            ← trs-gen Ts1 G1
            ← trs-gen Ts2 G2
            ← ls-disjoint G1 G2.
```

This rule says that expression `(es-let Es1 Es2)` (i.e. `let x = e1 in e2`) evaluates to `V2` in store `S`, yielding a new store `S2` and the trace `(trs-let Ts1 Ts2)`, provided that

- `Es1` evaluates to `V1` in store `S`, yielding a new store `S1` and the trace `Ts1`
- `Es2` applied to `V1` (i.e., `[V1/x] Es2`) evaluates to `V2` in store `S1`, yielding a new store `S2` and the trace `Ts2`
- the allocated locations of `Ts1` is `G1`
- the allocated locations of `Ts2` is `G2`, and
- `G1` and `G2` are disjoint.

In other words, this again says exactly what the paper rule says.

Valid evaluations are also given in `eval.elf`, type families `wf-evals` and `wf-evalc`. In contrast to the paper definitions, these are higher-order judgments in Twelf: a derivation of the general evaluation judgment is an argument to the valid evaluation judgment rather than a premise. The reason for this is that in the proof we silently coerce valid evaluations to general evaluations, but this is not permitted in Twelf—instead we work with general evaluations, with a separate derivation witnessing their validity. The rule for stable evaluation is

```
wf-evals_ :wf-evals Es’ R G (Devals : evals S Es V S’ Ts)
            ← wf-es Es S Es’ R
            ← trs-gen Ts G
            ← ls-disjoint R G.
```

This rule says that a derivation of `(evals S Es V S’ Ts)` (i.e. `(Es)` evaluates to `V` in store `S`, yielding a new store `S’` and trace `Ts`) is valid provided that `Es` reaches locations `R`, `Ts` allocates locations `G`, and `R` and `G` are disjoint.

The memo/hit rule for stable evaluation illustrates working with this higher-order judgment:

```
evals-memo-hit :cps S Ts1 S’ Ts
                → {Devals : evals S1 Es V S1’ Ts1}
                wf-evals Es’ R G Devals
                → evals S (es-memo Es) V S’ Ts.
```

This says that `(es-memo Es)` evaluates to `V` in store `S`, yielding a new store `S’` and trace `Ts`, provided that:

- change propagation of trace `Ts1` in store `S` yields a new store `S’` with trace `Ts`
- we have a derivation `Devals` that `Es` evaluates to `V` in store `S1`, yielding a new store `S1’` and trace `Ts1`
- `Devals` is a valid evaluation.

(Here we have inlined the paper oracle judgment—the oracle premise becomes a valid evaluation premise.) The syntax {`Devals : evals [...]`} just names the derivation of the evals premise so we can refer to it in the `wf-evals` premise. (This is a bit confusing syntactically—elsewhere we use the `A ← B` syntax, which has a logic-programming flavor; here we use `B → A` for consistency with the definition preceding reference of `Devals`; either is acceptable in Twelf.)

Memo-free evaluation is also given in `eval.elf`, type families `cln-evals` and `cln-evalc`. These are also higher-order judgments, which witness that the argument (general) evaluation derivation contains no `evals-memo-hit` or `evalc-memo-hit` cases.

Purely functional evaluation (Figure 17) is given in `pure.elf`, type families `evals-pure` and `evalc-pure`. These are very straightforward translations of the paper rules.

6.2 Theorems

The Twelf view of a theorem (in Twelf jargon, a ”metatheorem”, because it is stated in the Twelf metalogic, not in the encoded logic) comprises several parts: a relation among type families (judgments); a set of cases defining the relation; a ”mode declaration” describing which arguments to the relation are inputs and which outputs; a ”worlds declaration” describing in what contexts (”worlds”) the theorem holds; and a ”totality declaration” asserting that for all possible input terms, cases are provided which yield appropriate output terms—i.e. the relation is total.

For example, Theorem 2 (Correctness) is given (in `consistency.thm`, type family `wf-evals-imp-pure`) as follows:

```
wf-evals-imp-pure : {Devals : evals _ Es V S' _}
  wf-evals Es' R G Devals →
%%
  wf-val V S' V^ RV →
  evals-pure Es' V^ →
  type.
%mode wf-evals-imp-pure  +X1 +X2 -X3 -X4.
```

In other words, given derivations of judgments that

- `Es` evaluates in store `S` to value `V`, resulting in new store `S'` (and some trace which is ignored)
- the evaluation is valid (recall the `wf-evals` is a higher-order judgment)

return derivations of judgments that

- `V` is well-formed in store `S'`, and lifts to `V^`
- `Es'` (the lift of `Es` in `S`) evaluates under the pure semantics to `V^`

Twelf deals directly only with relations, so where the on-paper proof uses functional notation, the Twelf translation must expand the shorthand. In this case, the proposition that $(e \uparrow \sigma)\ \Downarrow^{\mathsf{S}}_{\text{det}}\ (v \uparrow \sigma')$ expands to three relations:

- e lifts to some e' in σ
- v lifts to some v' in σ'

- e' evaluates under the pure semantics to v'

The first is given as a subderivation of the `wf-evals` derivation `(wf-es Es S Es' R)`; the second and third are outputs of `wf-evals-imp-pure`.

For auxiliary judgments (in this case `wf-es`), whether they are inputs or outputs to a metatheorem (and whether they are given separately or as subderivations of another argument) are matters of convenience and proof engineering. From a relational point of view such variations do not state identical theorems, but each is a plausible reading of the on-paper theorem (which is not precise about its inputs and outputs).

The `wf-evals-imp-pure` relation is defined by one clause, which appeals to several lemmas:

```
      - :wf-evals-imp-pure Devals (wf-evals_ Dld Dtg Dwe) Dwv' Devals'
         ← can-evals-met Devals Dem
         ← evals-imp-cln-evals _ Devals Dem (wf-evals_ Dld Dtg Dwe)
      Devals'' _ _ Dce
         ← evals-imp-pure-evals
              Devals''
              (wf-evals_ Dld Dtg Dwe) Dce Dwv' Devals'.
```

We use `evals-imp-cln-evals` (corresponding to Lemma 5) to turn a general evaluation derivation into a clean evaluation derivation, and `evals-imp-pure-evals` (corresponding to Lemma 3) to turn the clean evaluation derivation into a pure evaluation derivation. In order to apply these lemmas we provide a metric argument to show the induction is well-founded, and we use `can-evals-met` to generate a suitable metric term.

The meat of the proof is in Lemmas 3 and 4 (given in `purity.thm`, type families `evals-imp-pure-evals` and `evalc-imp-pure-evalc`) and Lemmas 5 and 6 (given in `memo-elim.thm`, type families `evals-imp-cln-evals`, `evalc-imp-cln-evalc`, `evals-cps-imp-cln-evals`, and `evalc-cpc-imp-cln-evalc`;
the latter two are strengthenings of Lemmas 5 and 6 with an additional change propagation.

Here is `evals-cps-imp-cln-evals`:

```
 evals-cps-imp-cln-evals :
   N
   {Devals : evals S1 Es V S1' Ts1}
   evals-met Devals Nevals →
   wf-evals Es1' R1 G1 Devals →
   {Dcps : cps S Ts1 S' Ts}
   cps-met Dcps Ncps →
   sum Nevals Ncps N →
   wf-es Es S Es' R →
   trs-gen Ts G →
   ls-disjoint R G →
 %%{Devals' : evals S Es V S' Ts}
   evals-met Devals' N' →
   leq N' N →
   cln-evals Devals' →
 type.
 %mode evals-cps-imp-cln-evals +X1 +X2 +X3 +X4 +X5 +X6 +X7 +X8 +X9 +X10
 -X11 -X12 -X13 -X14.
```

Given a valid evaluation of Es (the `evals` and `wf-evals` arguments) and a change propagation to be applied to the trace (the `cps` argument) along with a derivation that the

```
- :evals-cps-imp-cln-evals
   _
   (evals-memo-hit Dcps0 Devals Dwfe)
   (evals-met-hit (Dsum0 : sum N1 N2 N1+N2) Dcm0 Dem)
   (wf-evals_ (Dld0 : ls-disjoint R0 G0)
              Dtg0
              (wf-es-memo Dwe0))
   Dcps
   Dcm
   (sum-s (Dsum : sum N1+N2 N3 N1+N2+N3))
   (wf-es-memo Dwe)
   Dtg
   Dld
%%
   (evals-memo-miss Devals')
   (evals-met-miss Dem')
   (leq-s Dleq')
   (cln-evals-miss Dce')
   ← sum-imp-leq Dsum (Dleq6 : leq N1+N2 N1+N2+N3) _
   ← leq-reduces _ _ Dleq6
   ← evals-cps-imp-cln-evals
       _ Devals Dem Dwfe Dcps0 Dcm0 Dsum0 Dwe0 Dtg0 Dld0
       Devals2 Dem2 (Dleq2 : leq N4 N1+N2) _
   ← can-sum _ _ (Dsum3 : sum N4 N3 N4+N3)
   ← leq-refl _ Dleq5
   ← sum-monotone
        Dleq2 Dleq5 Dsum3 Dsum (Dleq4 : leq N4+N3 N1+N2+N3)
   ← leq-reduces _ _ Dleq4
   ← evals-cps-imp-cln-evals
      _ Devals2 Dem2
                (wf-evals_ Dld0 Dtg0 Dwe0)
                Dcps Dcm Dsum3 Dwe Dtg Dld
       Devals' Dem' (Dleq3 : leq N5 N4+N3) Dce'
   ← leq-trans Dleq3 Dleq4 (Dleq' : leq N5 N1+N2+N3)
.
```

Fig. 19: An evaluation derivation where the outermost term is a memo hit.

locations reached by Es are disjoint from those allocated by the final trace, return a clean evaluation of Es. The remaining inputs and outputs (`evals-met`, `cps-met`, `sum`, `leq`) deal with the metric required to show termination.

Figure 19 shows the case for an evaluation derivation where the outermost term is a memo hit. The evaluation derivation includes a change propagation derivation (`Dcps0`). We therefore make two inductive calls, the first to eliminate `Dcps0`, the second to eliminate the input change propagation derivation (`Dcps`). The rest of the case (the calls to `sum-imp-leq`, `leq-reduces`, `can-sum`, `leq-refl`, `sum-monotone`, and `leq-trans`) involves the termination metric. Twelf has no built-in support for mathematical reasoning, so the termination reasoning is given—rather painfully—in relational terms over unary natural numbers.

7 Implementation

The language and semantics proposed in this paper has been adapted to Standard ML and C languages and been implemented by extending these languages with needed language primitives and run-time system support [Acar et al. 2009; Hammer et al. 2009, 2011; Ley-Wild et al. 2008b, 2009]. Both approaches specify carefully the oracle by limiting it to return only certain computation and are able to prove stronger soundness results than we do in this paper by proving also termination. The C-based approach also makes it possible to integrate a form of garbage collection into change propagation efficiently.

To realize the semantics efficiently, the implementations use a graph representation of traces that allow doing work proportional to re-evaluated expressions. In addition, function calls are cached as a mapping from function names and arguments to their results. During an evaluation, the oracle consults the function-call caches to determine whether a function call about to be evaluated can be re-used. The oracle determines that a function call can be re-used if the function and the arguments involved in the call are identical. In addition, the oracle guarantees that a call is never used more than once in an evaluation. To determine equality, it suffices to consider tag or label equality, which can be performed very efficiently. To ensure that function calls are not used multiple times, the implementations rely on a total ordering of all function calls. Experimental evaluations [Acar et al. 2009; Hammer et al. 2009; Ley-Wild et al. 2008b] show that the proposed semantics can be implemented with modest overheads in complete runs where all data is new and yield massive speedups in cases when the data changes incrementally.

8 Related Work

The problem of adapting computations to small changes to their data has been studied extensively in several communities. There are many hundreds of papers on this problem in several research communities including the algorithms and programming-languages communities. Fortunately there are several excellent surveys of work that present an in depth reviews of previous work, e.g., for the algorithms community [Demetrescu et al. 2005a,b; Guibas 2004; Chiang and Tamassia 1992], and for (primarily) programming-languages community [Ramalingam and Reps 1993]. In this section, we review only the closely related work and refer the reader to aforementioned surveys for more citations.

8.1 Incremental Computation

The work on incremental computation aims to devise general-purpose, language-based techniques for enabling programs to automatically respond to modifications to their data. The most effective techniques are based on dependency graphs, memoization, and partial evaluation, which we briefly review below.

Dependency-graph techniques record the dependencies among data in a computation, so that a change-propagation algorithm can update the computation when the input is changed. Demers, Reps, and Teitelbaum [Demers et al. 1981] and Reps [Reps 1982a,b] introduced the idea of *static dependency graphs* and presented a change-propagation algorithm for them. Hoover generalized the approach outside the domain of attribute grammars [Hoover 1987]. Yellin and Strom used the dependency graph ideas within the INC language [Yellin and Strom 1991], and extended it by having incremental computations within each of its array primitives.

The key limitation of static dependency graphs is that they do not permit the change-propagation algorithm to update the dependency structure. This limitation restricts the types of computations to which static-dependency graphs can be applied, making them inapplicable in general purpose computational models. For example, the INC language, which uses static dependency graphs for incremental updates, does not permit recursion. Static dependency graphs, however, can be effective in certain structured, syntax-directed applications such as in incremental evaluation of attribute grammars (e.g., [Demers et al. 1981; Hedin 1992]).

The limitations of static dependency graphs motivated researchers to look into alternatives. Pugh and Teitelbaum [Pugh and Teitelbaum 1989] applied the classic idea of memoization [Bellman 1957; McCarthy 1963; Michie 1968] (also called function caching) to incremental computation. Memoization is more general than static dependency graphs and is applicable to any purely functional program. Since the work of Pugh and Teitelbaum, others have investigated applications of various forms of memoization to incremental computation [Abadi et al. 1996; Liu et al. 1998; Heydon et al. 2000; Acar et al. 2003]. The idea behind memoization is to remember function calls and their results, and reuse them whenever possible. In the context of incremental computation, memoization can improve efficiency when re-executions of a program with similar inputs perform similar function calls. While this is indeed the case in some cases, in many cases it is not: changes to the input can prevent large parts of the computation from being reused. We have given an example for this in Section 2; other examples are discussed elsewhere [Acar et al. 2009]. Intuitively, the problem with memoization is that all function calls that consume changed data and all their ancestors in the function call tree needs to be re-executed.

Other approaches to incremental computation are based on partial evaluation. Sundaresh and Hudak's approach [Sundaresh and Hudak 1991] requires the user to fix the partition of the input that the program will be specialized on and partially evaluates the program with respect to this partition, allowing data outside of the partition to be changed incrementally. Field [Field 1991], and Field and Teitelbaum [Field and Teitelbaum 1990] presented techniques for incremental computation in the context of the lambda calculus. Their approach is similar to Hudak and Sundaresh's, but they present formal reduction systems that optimally use partially evaluated results.

8.2 Dynamic Algorithms

The algorithms community approached the problem of allowing computation to adapt to incremental changes to data from a different perspective. Instead of seeking general-purpose techniques that apply to a broad range of applications, researchers in the algorithms community develop *dynamic algorithms* or *dynamic data structures* (e.g., [Sleator and Tarjan 1985; Chiang and Tamassia 1992; Eppstein et al. 1999]) for solving specific problems individually. Dynamic algorithms enable computing a desired property while allowing the user to modify the input (e.g., inserting/deleting elements). For example, a dynamic algorithm for computing a sorted order allows the user to insert/delete points into/from the input set while keeping the output sorted.

Since a dynamic algorithm is carefully designed to take advantage of the structural properties of the specific problem considered, it is often very efficient, and can add significant, e.g, linear-time, improvements over the static version. Previous research shows, however, that dynamic algorithms can be quite complex and difficult to design, analyze, and implement even for problems that are simple in the static case, where changes to data are not allowed. A striking example is the two-dimensional convex-hull problem, whose static version has been solved in the early days of computer science (e.g., [Graham 1972]). The dynamic (incremental) version of the same problem required many more decades of research ultimately culminating in an optimal (in the amortized sense) solution [Overmars and van Leeuwen 1981; Brodal and Jacob 2002]. The convex-hull problem is not an outlier; other examples include minimum spanning trees [Frederickson 1985; Eppstein et al. 1997; Henzinger and King 1997, 1999; Holm et al. 2001]), and dynamic trees [Sleator and Tarjan 1983], which continues to be studied [Sleator and Tarjan 1983; Cohen and Tamassia 1991; Alstrup et al. 1997; Frederickson 1997; Tarjan and Werneck 2007].

8.3 Self-Adjusting Computation

The term "incremental computation" is often used to refer to techniques that aim at updating computations under small, incremental changes to their data. We use term "self-adjusting computation" to refer to a specific technique (as proposed here) that uses a combination of change propagation on dynamic dependence graphs and memoization to solve the incremental-computation problem. Before we discuss previous work in more detail, we would like to clarify our use of this term. We feel that "self-adjusting" represents the approach well and avoids the terminology confusion involving the term "incremental", which is used to refer to different concepts in different communities. Specifically in the programming languages community, the term "incremental" is used to refer to small changes to data. In the algorithms community, the same term refers to a class of algorithms which only admits "additions" (e.g., insertions) to data but allow no deletions. For algorithms that allow both insertions and deletions, the algorithms community use the term "dynamic", which is a term that is often used to refer to the run-time (as opposed to the compilation time) in programming languages. Thus the terms "incremental" and "dynamic" both mean different things to the key communities interested in the broader problem of incremental computation.

Research on self-adjusting computation started with the invention of dynamic dependency graphs (DDGs) and a change propagation algorithm that can update the dependency structure dynamically as it executes [Acar et al. 2006]. Dynamic dependency graphs can be constructed from any purely functional program and can be kept up to date as data changes by a change-propagation algorithm that inserts and deletes dependencies as necessary. The motivation behind the research on self-adjusting computation has been to achieve the efficiency of dynamic algorithms without the high design, analysis, and implementation complexity required to achieve that efficiency.

Although dynamic dependency graph are general purpose, they are effective only for certain classes of data changes (Section 2). This paper presents a semantics for self-adjusting computation based on a memoizing change propagation technique that can dramatically improve re-use of computations; the paper is the journal version of a conference paper [Acar [redacted] et al. 2007] published earlier. The semantics presented here established the foundation for followup work on self-adjusting computation and related topics. A paper proposed algorithms for realizing the semantics efficiently and implemented it as a Standard ML library [Acar et al. 2009]. Earlier implementations of self-adjusting computation include the SML library [Acar et al. 2006] and an Haskell implementation by Carlsson [Carlsson 2002]. Other work generalized the approach to support imperative references [Acar et al. 2008]. These results then led to the development of the CEAL [Hammer et al. 2009, 2011] and Delta ML languages [Ley-Wild et al. 2008b], which provide direct language support for self-adjusting computation.

Aforementioned realizations of the semantics presented here have led to follow-up work on a relatively broad set of applications ranging from simpler computational
benchmarks [Acar et al. 2009] to more sophisticated applications in computational geometry and machine learning, where various important (also open) problems have been solved (e.g., [Sümer et al. 2011; Acar et al. 2010]). The algorithmic results use stability analysis [Acar 2005; Ley-Wild et al. 2009], which offers an algorithmic cost model for the technique presented in this paper. The applications show that the proposed approach can provide asymptotically optimal updates in theory while also delivering significant efficiency improvements in practice. The algorithmic results rely on a particular implementation of the oracle which enables analyzing the efficiency of change propagation [Ley-Wild et al. 2009]. Such a restricted oracle also allows proving slightly stronger theorems on semantics, making it possible to reason also about termination.

Shankar and Bodik's work on DITTO adapts self-adjusting computation techniques as proposed here and implemented in future work [Acar et al. 2009] to the specific problem of checking data structural invariants [Shankar and Bodik 2007]. They refine the general-purpose computation reuse model proposed here by developing heuristics that work well for the problem of checking invariants for data structures written in the Java language.

Some earlier [Acar et al. 2004; Hammer et al. 2007] and more recent work [Burckhardt et al. 2011] realized that many parallel algorithms are amenable to self-adjusting computation and developed techniques for taking advantage of both simultaneously.

8.4 Functional Reactive Programming

Functional reactive programming [Elliott and Hudak 1997; Elliott 1998; Wan and Hudak 2000; Wan et al. 2001] aims to provide a functional, even declarative interface to developing *reactive* programs that can respond to interactive events, often called *behaviors* and *signals*. A reactive program evaluates in time steps, each of which returns a result and a new program to be evaluated at the next time step. Since a functional reactive program returns a new program to be evaluated at the next step, it can be viewed as a kind of "self-modifying" code. In functional-reactive-programming terminology, "switch" expressions refer to expressions that provide this kind of code modification ability. Since each time step involves small updates to data, it seems possible for incremental computation, and self-adjusting computation specifically, to be used to improve the efficiency of functional reactive programming.

Much previous work on functional reactive programming, however, do not employ incremental computation techniques, focusing instead on taming the time and space consumption of the "one-shot", static approach. Previous work shows that, due to their expressive power, implementing functional reactive programming efficiently can be challenging; the terms "time leaks" and "space leaks" refer to two specific problems that arise when trying bound the time and the space complexity of implementations. More recent work took some steps in the direction of connecting functional programming and incremental computation [Cooper and Krishnamurthi 2006]. Considering a language without switching, Cooper and Krishnamurthi [Cooper and Krishnamurthi 2006], record dependencies between signals and behaviors and the rest of data and use a propagation algorithm to update the computation when the values of the signals change. As also noticed by the authors [Cooper and Krishnamurthi 2004], it appears possible to encode switchless functional reactive programming in self-adjusting computation. The idea behind such an encoding would be to use modifiables to represent signals (time-varying values) and rely on the dependence tracking and change propagation mechanisms of self-adjusting computation for automatic incremental updates. Understanding whether incremental computation can be theoretically and practically effective for functional reactive programs remains to be an interesting open problem.

9 Conclusion

We present a general semantics for integrating memoization and change propagation where memoization is modeled as a non-deterministic oracle, and computation reuse is possible in the presence of mutation. Mutations arise for two reasons. First the semantics permits the store to be modified between two runs while allowing computations to be reused between two such runs—this models dynamic data changes. Second, the techniques for change propagation mutate the store by selectively re-executing pieces of the first run to derive the second run. The key idea behind the semantics is to enable the reuse of computations themselves by adapting reused computations to mutations via recursive applications of change propagation. Our main theorem shows that the semantics is consistent with deterministic, purely functional programming. By giving a general, oracle-based semantics for combining memoization and change propagation, we cover a variety of possible techniques for implementing self-adjusting-computation. By proving the semantics correct with minimal assumptions, we identify the properties that correct implementations must satisfy.

References

Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

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

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

Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 2004.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6):990–1034, 2006.

Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. In *European Symposium on Programming*, 2007.

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, 2008.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–53, 2009.

Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*, 2010.

Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming*, pages 270–280, 1997.

Richard Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symposium on Cloud Computing*, 2011.

Gerth Stolting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 617–626, 2002.

Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

Magnus Carlsson. Monads for incremental computing. In *International Conference on Functional Programming*, pages 26–35, 2002.

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

R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–61, 1991.

Gregory H. Cooper and Shriram Krishnamurthi. FrTime: Functional Reactive Programming in PLT Scheme. Technical Report CS-03-20, Department of Computer Science, Brown University, April 2004.

Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*, 2006.

Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Principles of Programming Languages*, pages 105–116, 1981.

Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005a.

Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. *Handbook on Data Structures and Applications*, chapter 35: Dynamic Trees. Dinesh Mehta and Sartaj Sahni (eds.), CRC Press Series, in Computer and Information Science, 2005b.

Conal Elliott. Functional implementations of continuous modeled animation. *Lecture Notes in Computer Science*, 1490:284–299, 1998.

Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.

David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *ACM Conference on LISP and Functional Programming*, pages 307–322, 1990.

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

Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14:781–798, 1985.

Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997.

R. L. Graham. An efficient algorithm for determining the convex hull of a finete planar set. *Information Processing Letters*, 1:132–133, 1972.

L. Guibas. Modeling motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004.

Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

Matthew Hammer, Georg Neis, Yan Chen, and Umut A. Acar. Self-adjusting stack machines. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

Görel Hedin. *Incremental Semantics Analysis*. PhD thesis, Department of Computer Science, Lund University, March 1992.

Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 594–604. Springer-Verlag, 1997.

Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Programming Language Design and Implementation*, pages 311–320, 2000.

Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

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

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. Technical Report CMU-CS-08-141, Department of Computer Science, Carnegie Mellon University, July 2008a.

Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. In *Int'l Conference on Functional Programming*, 2008b.

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

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

John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968.

Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages*, pages 315–328, 1989.

G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages*, pages 502–510, 1993.

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

Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 169–176, 1982b.

Ajeet Shankar and Rastislav Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Programming Language Design and Implementation*, 2007.

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

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

Özgür Sümer, Umut A. Acar, Alexander Ihler, and Ramgopal Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning*, 8:180–186, 2011.

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

Robert Tarjan and Renato Werneck. Dynamic trees in practice. In *Proceeding of the 6th Workshop on Experimental Algorithms (WEA 2007)*, pages 80—93, 2007.

Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *SIGPLAN Not.*, 36(10):146–156, 2001.

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.