A Consistent Semantics of Self-Adjusting Computation

Umut A. Acar[1], Matthias Blume[1], and Jacob Donham[2]

[1] Toyota Technological Institute
[2] Carnegie Mellon University

Abstract. This paper presents a semantics of self-adjusting computation and proves that the semantics is correct and consistent. The semantics integrates change propagation with the classic idea of memoization to enable reuse of computations under mutation to memory. During evaluation, reuse of a computation via memoization triggers a change propagation that adjusts the reused computation to reflect the mutated memory. Since the semantics combines memoization and change-propagation, it involves both non-determinism and mutation. 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 consistent with purely functional programming. We formalized the semantics and its meta-theory in the LF logical framework and machine-checked the proofs in Twelf.

1 Introduction

Self-adjusting computation is a technique for enabling programs to respond to changes to their data (e.g., inputs/arguments, external state, or outcome of tests). By automating the process of adjusting to any data change, self-adjusting computation generalizes incremental computation (e.g., [10, 18, 19, 12, 11, 17]). Previous work shows that the technique can speed up response time by orders of magnitude over recomputing from scratch [3, 7], closely match best-known (problem-specific) algorithms both in theory [2, 6] and in practice [7, 8].

The approach achieves its efficiency by combining two previously proposed techniques: change propagation [4], and memoization [5, 1, 17, 15]. Due to an interesting duality between memoization and change propagation, combining them is crucial for efficiency. Using each technique alone yields results that are far from optimal [3, 2]. The semantics of the combination, however, is complicated because the techniques are not orthogonal: conventional memoization requires purely functional programming, whereas change propagation crucially relies on mutation for efficiency. For this reason, no semantics of the combination existed previously, even though the semantics of change propagation [4] and memoization (e.g., [5, 17]) has been well understood separately.

This paper gives a general semantic framework that combines memoization and change propagation. By modeling memoization as a non-deterministic oracle, we ensure that the semantics applies to many different ways in which memoization, and thus the combination, can be realized. We prove two main theorems stating that the semantics is *consistent* and *correct* (Section 3). The consistency theorem states 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. The correctness theorem states that self-adjusting computation is consistent with purely functional programming by showing that evaluation returns the (observationally) same value as a purely functional evaluation. Our proofs do not make any assumptions about typing. Our results therefore apply in both typed and untyped settings. (All previous work on self-adjusting computation assumed strongly typed languages.)

To study the semantics we extend the *adaptive functional language* AFL [4] with a `memo` construct for memoization. We call this language AML (Section 2). 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. A `memo` expression is evaluated by first consulting the *memo-oracle*, which non-deterministically returns either a *miss* or a *hit*. Unlike in conventional memoization, hit returns a trace of the evaluation of the memoized expression, not just its result. To adjust the computation to the mutated memory, the semantics performs a change propagation on the returned trace. Change propagation and ordinary evaluation are, therefore, intertwined in a mutually recursive fashion to enable computation reuse under mutation.

The proofs for the correctness and consistency theorems (Section 3) 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 [13] and machine-checked the proofs using Twelf [16] (Section 4). 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 Twelf code is available at `http://www.cs.cmu.edu/~jdonham/aml-proof/`. We note

$$
\begin{array}{llcl}
\textit{Values} & v & ::= & \texttt{()} \mid n \mid x \mid l \mid (v_1, v_2) \mid \texttt{in}_\texttt{l}\ v \mid \texttt{in}_\texttt{r}\ v \mid \\
 & & & \texttt{fun}_\texttt{s}\ f(x)\ \texttt{is}\ e_s \mid \texttt{fun}_\texttt{c}\ f(x)\ \texttt{is}\ e_c \\
\textit{Prim. Op.} & o & ::= & \texttt{not} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots \\
\textit{Exp.} & e & ::= & e_s \mid e_c \\
\textit{St. Exp.} & e_s & ::= & v \mid o\texttt{(}v_1, \ldots, v_n\texttt{)} \mid \texttt{mod}\ e_c \mid \texttt{memo}_\texttt{s}\ e_s \mid \texttt{apply}_\texttt{s}\texttt{(}v_1, v_2\texttt{)} \mid \\
 & & & \texttt{let}\ x = e_s\ \texttt{in}\ e_s' \mid \texttt{let}\, x_1 \times x_2 \ = \ v\, \texttt{in}\, e_s \mid \\
 & & & \texttt{case}\, v\, \texttt{of}\ \texttt{in}_\texttt{l}\ \texttt{(}x_1\texttt{)}\ \Rightarrow\ e_s \mid \texttt{in}_\texttt{r}\ \texttt{(}x_2\texttt{)}\ \Rightarrow\ e_s'\ \texttt{end} \\
\textit{Ch. Exp.} & e_c & ::= & \texttt{write(}v\texttt{)} \mid \texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c \mid \texttt{memo}_\texttt{c}\ e_c \mid \texttt{apply}_\texttt{c}\texttt{(}v_1, v_2\texttt{)} \mid \\
 & & & \texttt{let}\ x = e_s\ \texttt{in}\ e_c \mid \texttt{let}\, x_1 \times x_2 \ = \ v\, \texttt{in}\, e_c \mid \\
 & & & \texttt{case}\, v\, \texttt{of}\ \texttt{in}_\texttt{l}\ \texttt{(}x_1\texttt{)}\ \Rightarrow\ e_c \mid \texttt{in}_\texttt{r}\ \texttt{(}x_2\texttt{)}\ \Rightarrow\ e_c'\ \texttt{end} \\
\textit{Program} & p & ::= & e_s
\end{array}
$$

Fig. 1. The abstract syntax of AML.

that checking the proofs in Twelf was not a merely an encoding exercise. In fact, our initial paper-and-pencil proof was not correct. In the process of making Twelf accept the proof, we simplified the rule systems, fixed the proof, and even generalized it. In retrospect, we feel that the use of Twelf was critical in obtaining the 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. In Section 5, we describe some implementation strategies for realizing the AML semantics. One of these strategies has been implemented and discussed elsewhere [3]. We note that this implementation is somewhat broader than the semantics described here because it allows re-use of memoized computations even when they match partially, via the so called `lift` construct. We expect that the techniques described here can be extended for the `lift` construct.

2 The Language

We describe a language, called AML, that combines the features of an adaptive functional language (AFL) [4] 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.

2.1 Abstract syntax

The abstract syntax of AML is given in Figure 1. We use meta-variables x, y, and z (and variants) to range over an unspecified set of variables, and meta-variable l (and variants) 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 directly or indirectly be 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, stable-function applications, and the creation of new modifiables. A *stable function* is a function whose body is a stable expression. The application of a stable function is a stable expression. The expression $\mathtt{mod}\ e_c$ allocates a modifiable reference and initializes it 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 $\mathtt{memo}_{\mathtt{S}}\ e_s$.

Changeable expressions always execute in the context of an enclosing `mod`-expression that provides the implicit target location that every changeable expression writes to. The changeable expression $\mathtt{write}(v)$ writes the value v into the target. The expression $\mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ e_c$ 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 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 $\mathtt{memo}_{\mathtt{C}}\ e_c$. 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.

2.2 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 3).

$$\frac{v \in \{(), n, x\}}{v, \sigma \xrightarrow{\texttt{wf}} v, \emptyset} \qquad \frac{l \in \texttt{dom}(\sigma) \quad \sigma(l), \sigma \xrightarrow{\texttt{wf}} v, L}{l, \sigma \xrightarrow{\texttt{wf}} v, \{l\} \cup L} \qquad \frac{v_1, \sigma \xrightarrow{\texttt{wf}} v_1', L_1 \quad v_2, \sigma \xrightarrow{\texttt{wf}} v_2', L_2}{(v_1, v_2), \sigma \xrightarrow{\texttt{wf}} (v_1', v_2'), L_1 \cup L_2}$$

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

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

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

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

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

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

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

$$\frac{e, \sigma \xrightarrow{\texttt{wf}} e', L}{\texttt{memo}_{\{\texttt{s},\texttt{c}\}}\ e, \sigma \xrightarrow{\texttt{wf}} \texttt{memo}_{\{\texttt{s},\texttt{c}\}}\ e', L}$$

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

Fig. 2. Well-formed expressions and lifts.

We use the judgment $e, \sigma \xrightarrow{\texttt{wf}} e', L$ 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 2. 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 $\texttt{reach}\,(e, \sigma)$ we implicitly assert that there exist a location-free expression e' and a set of locations L such that $e, \sigma \xrightarrow{\texttt{wf}} e', L$. The notation $e \uparrow \sigma$ itself stands for the lifted expression e', and $\texttt{reach}\,(e, \sigma)$ stands for the set of reachable locations L. It is easy to see that e and σ uniquely determine $e \uparrow \sigma$ and $\texttt{reach}\,(e, \sigma)$ (if they exist).

2.3 Dynamic semantics

The evaluation judgments of AML (Figures 5 and 6) 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 $\mathtt{T}_s$, 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'.

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 traces is given by the following grammar:

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

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 $\mathtt{read}_{l \rightarrow x = v.e}\ \mathtt{T}_c$ 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. This records the dependency of the target on the value of the location read.

The set of locations allocated (via `mod`) during the evaluation that produced a trace $\mathtt{T}$ is denoted $\mathtt{alloc}(\mathtt{T})$ (the full definition is given in the accompanying technical report [9]). For example, if $\mathtt{T}_{\text{sample}} = \mathtt{let}\ (\mathtt{mod}\ l_1 \leftarrow \mathtt{write}\ 2)\ (\mathtt{read}_{l_1 \rightarrow x = 2.e}\ \mathtt{write}\ 3)$, 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 states 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, no AML operation is permitted to be sensitive to the identity of locations. In the case of primitive operations we formalize this by postulating that they commute with lifts:

$$\begin{array}{l} \text{If } \forall i. v_i, \sigma \xrightarrow{\mathtt{wf}} v'_i, L_i \text{ and } v = o(v_1, \ldots, v_n), \\ \text{then } v, \sigma \xrightarrow{\mathtt{wf}} v', L \text{ such that } v' = o(v'_1, \ldots, v'_n). \end{array}$$

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

Fig. 3. Valid evaluations.

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 for proving correctness: the requirement that the reachable portion of the store is acyclic ensures that the approach is consistent with purely functional programming, 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 only to 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 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 3.

The Oracle. The dynamic semantics for AML uses an oracle to model memoization. Figure 4 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 depend 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.

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

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

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

Fig. 4. The oracle.

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

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

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

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

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

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

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

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

Fig. 5. Evaluation of stable expressions.

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 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 5 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

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

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

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

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

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

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

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

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

Fig. 6. Evaluation of changeable expressions.

traces of its sub-expressions. For the traces to be well-formed, the rule requires that they allocate disjoint 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 `T`. To adapt the trace to the current store σ, the evaluation performs a change propagation on `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.

Changeable Evaluation. Figure 6 shows the evaluation rules for changeable expressions. Evaluations in changeable mode perform *destination passing.* The `let`, `memo`, `apply` rules are similar to the corresponding rules in stable mode except that the body of each expression is evaluated in changeable mode. The

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

$$\frac{\begin{array}{c} l \not\in \texttt{alloc}\,(\texttt{T}') \\ \sigma, l \leftarrow \texttt{T} \overset{\mathsf{C}}{\curvearrowright} \sigma', \texttt{T}' \end{array}}{\sigma, \texttt{mod}\ l \leftarrow \texttt{T} \overset{\mathsf{S}}{\curvearrowright} \sigma', \texttt{mod}\ l \leftarrow \texttt{T}'}\ \textbf{(mod)} \qquad \frac{}{\sigma, l \leftarrow \texttt{write}\ v \overset{\mathsf{C}}{\curvearrowright} \sigma[l \leftarrow v], \texttt{write}\ v}\ \textbf{(write)}$$

$$\frac{\begin{array}{c} \sigma, \texttt{T}_1 \overset{\mathsf{S}}{\curvearrowright} \sigma', \texttt{T}_1' \\ \sigma', \texttt{T}_2 \overset{\mathsf{S}}{\curvearrowright} \sigma'', \texttt{T}_2' \\ \texttt{alloc}\,(\texttt{T}_1') \cap \texttt{alloc}\,(\texttt{T}_2') = \emptyset \end{array}}{\sigma, \texttt{let}\ \texttt{T}_1\ \texttt{T}_2 \overset{\mathsf{S}}{\curvearrowright} \sigma'', \texttt{let}\ \texttt{T}_1'\ \texttt{T}_2'}\ \textbf{(let/s)} \qquad \frac{\begin{array}{c} \sigma, \texttt{T}_1 \overset{\mathsf{C}}{\curvearrowright} \sigma', \texttt{T}_1' \\ \sigma', l \leftarrow \texttt{T}_2 \overset{\mathsf{C}}{\curvearrowright} \sigma'', \texttt{T}_2' \\ \texttt{alloc}\,(\texttt{T}_1') \cap \texttt{alloc}\,(\texttt{T}_2') = \emptyset \end{array}}{\sigma, l \leftarrow (\texttt{let}\ \texttt{T}_1\ \texttt{T}_2) \overset{\mathsf{C}}{\curvearrowright} \sigma'', (\texttt{let}\ \texttt{T}_1'\ \texttt{T}_2')}\ \textbf{(let/c)}$$

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

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

Fig. 7. Change propagation judgments.

`read` expression substitutes the value stored in σ at the location being read l' 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 7 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 $\sigma, \texttt{T}_s \overset{\mathsf{S}}{\curvearrowright} \sigma', \texttt{T}_s'$ states that change propagating into the stable trace $\texttt{T}_s$ in the context of the store σ yields the store σ' and the stable trace $\texttt{T}_s'$. The changeable change-propagation judgment $\sigma, l \leftarrow \texttt{T}_c \overset{\mathsf{C}}{\curvearrowright} \sigma', \texttt{T}_c'$ states that change propagation into the changeable trace $\texttt{T}_c$ with target l in the context of the store σ yields the changeable trace $\texttt{T}_c'$ 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` for the body to obtain a new trace $\texttt{T}'$ and returns a trace where `T` is substituted by $\texttt{T}'$ under the condition that the target l is not allocated in $\texttt{T}'$. This condition is necessary to ensure the allocation integrity of the returned trace. The stable `let` rule propagates into its two parts $\texttt{T}_1$ and $\texttt{T}_2$ recursively and returns a trace by combining the resulting traces $\texttt{T}_1'$ and $\texttt{T}_2'$ 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 re-used changeable computation is recorded in the new store. The `read` rule depends on whether the contents of the location l' being read is the same in the store as the value v recorded in the trace. If the contents is the same as in the trace, then change propagation proceeds into the body `T` of the read and the resulting trace is substituted for `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 and otherwise propagate forward. Only when a `read` finds that the location in question has changed, it re-runs the changeable computation that is in its body and replaces 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 $\sigma, e \Downarrow^{\mathtt{S}}_{\mathrm{ok}} v, \sigma', \mathtt{T}$ or $\sigma, l \leftarrow e \Downarrow^{\mathtt{C}}_{\mathrm{ok}} \sigma', \mathtt{T}$, 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 $\sigma, e \Downarrow^{\mathtt{S}}_{\mathrm{ok}} v, \sigma', \mathtt{T}$ 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 $\sigma, l' \leftarrow e \Downarrow^{\mathtt{C}}_{\mathrm{ok}} \sigma', \mathtt{T}$ 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 3 and 4 in Section 3 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 $\sigma, e \Downarrow^{\mathtt{S}}_{\emptyset} v, \sigma', \mathtt{T}$ or $\sigma, l \leftarrow e \Downarrow^{\mathtt{C}}_{\emptyset} \sigma', \mathtt{T}$ if there is a derivation for $\sigma, e \Downarrow^{\mathtt{S}} v, \sigma', \mathtt{T}$ or $\sigma, l \leftarrow e \Downarrow^{\mathtt{C}} \sigma', \mathtt{T}$, respectively, that does not use any **memo/hit** rule. We call such an evaluation *memo-free*. We use $\Downarrow^{\mathtt{S}}_{\emptyset,\mathrm{ok}}$ in place of $\Downarrow^{\mathtt{S}}_{\mathrm{ok}}$ and $\Downarrow^{\mathtt{C}}_{\emptyset,\mathrm{ok}}$ in place of $\Downarrow^{\mathtt{C}}_{\mathrm{ok}}$ to indicate that a valid evaluation is also memo-free.

2.4 Deterministic, purely functional semantics

By ignoring memoization and change-propagation, we can give an alternative, purely functional, semantics for location-free AML programs [9]. 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 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 3).

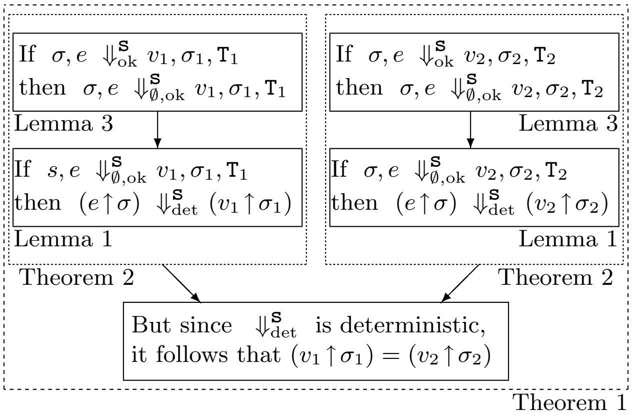
If $\sigma, e \Downarrow^{\mathsf{S}}_{\text{ok}} v_1, \sigma_1, \mathtt{T}_1$
then $\sigma, e \Downarrow^{\mathsf{S}}_{\emptyset,\text{ok}} v_1, \sigma_1, \mathtt{T}_1$
Lemma 3
If $s, e \Downarrow^{\mathsf{S}}_{\emptyset,\text{ok}} v_1, \sigma_1, \mathtt{T}_1$
then $(e \uparrow \sigma) \Downarrow^{\mathsf{S}}_{\text{det}}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

Fig. 8. The structure of the proofs.

3 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 8, consistency (Theorem 1) is a consequence of correctness (Theorem 2).

3.1 Main theorems

Consistency uses *structural equality* based on the notion of *lifts* (see Section 2.2) 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 saying 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 is the preservation of well-formedness during evaluation, and the inability of AML programs to create cyclic memory graphs.

3.2 Proof outline

The consistency theorem is proved in two steps. First, Lemmas 1 and 2 state that consistency is true in the restricted setting where all evaluations are memo-free.

Lemma 1 (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 2 (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 3 and 4 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 3 (memo-freedom/st.). *If* $\sigma, e \Downarrow^{\mathtt{S}}_{\mathrm{ok}} v, \sigma', T$, *then* $\sigma, e \Downarrow^{\mathtt{S}}_{\emptyset} v, \sigma', T$ *where* $\mathtt{reach}(v, \sigma') \subseteq \mathtt{reach}(e, \sigma) \cup \mathtt{alloc}(T)$.

Lemma 4 (memo-freedom/ch.). *If* $\sigma, l \leftarrow e \Downarrow^{\mathtt{C}}_{\mathrm{ok}} \sigma', T$, *then* $\sigma, l \leftarrow e \Downarrow^{\mathtt{C}}_{\emptyset} \sigma', T$ *where* $\mathtt{reach}(\sigma'(l), \sigma') \subseteq \mathtt{reach}(e, \sigma) \cup \mathtt{alloc}(T)$.

The proof for Lemmas 3 and 4 proceeds by simultaneous induction over the expression e. It is outlined in far more detail in the accompanying technical report [9]. 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 5). To see why this is important, consider as an example the evaluating of the expression: $\mathtt{memo}_{\mathtt{S}}$ (`mod` (`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 use fresh locations only, it would allocate some $l' \notin \mathtt{dom}(\sigma)$, and return that. But since $l \in \mathtt{dom}(\sigma)$, $l \neq l'$.

4 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 [16] and machine-checked the proofs. We follow the standard *judgments as types* methodology [13], 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 manuscript [14].

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 (see [9] for details) in overall structure. The proof of memo-freedom consists of four mutually-inductive lemmas: memo-freedom for stable and changeable expressions (Lemma 3 and Lemma 4), 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, Twelf has poor support for reusable libraries at present. Moreover, our needs are 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 nth list element corresponds to the nth location. This makes it easy to prove the necessary lemmas by parallel induction over the lists. The Twelf code can be found at `http://www.cs.cmu.edu/~jdonham/aml-proof/`

5 Implementation Strategies

The dynamic semantics of AML (Section 2) does not translate directly to an algorithm, not to mention an efficient one. [3] In particular, an algorithm consistent with the semantics must specify an oracle and a way to allocate locations

[3] This does not constitute a problem for our results, since our theorems and lemmas concern given derivations (not the problem finding them).

to ensure that all locations allocated in a trace are unique. We briefly describe a conservative strategy for implementing the semantics. The strategy ensures that

1. each allocated location is fresh (i.e., is not contained in the memory)
2. the oracle returns only traces currently residing in the memory,
3. the oracle never returns a trace more than once, and
4. the oracle performs function comparisons by using tag equality.

The first two conditions together ensure that each allocated location is unique. The third condition guarantees that no location can appear in the execution trace more than once. This condition is conservative, because it is possible that the parts of a trace returned by the oracle are thrown away (become unused) during change propagation. This strategy can be relaxed by allowing the change-propagation algorithm to return unused traces to the oracle. The last condition enables implementing oracle queries by comparing functions and their arguments by using tag equality. Since in the semantics, the oracle is non-deterministic, this implementation strategy is consistent with the semantics.

The conservative strategy can be implemented in such a way that the total space consumption is no more than that of a from-scratch run. Such an implementation has been completed and shown to be effective for a reasonably broad range applications [3, 7]. The implementation, however, places further restrictions on the oracle that are not required by the proof (e.g., computations must always be re-used in the same order).Our results shows that these restrictions are not necessary for correctness and can potentially be relaxed—such an implementation can be more broadly applicable.

We note that the described conservative implementation does not guarantee correctness, because it requires the programmer to supply all the free variables of memoized expressions. When the programmer misspecifies the free variables, the correctness guarantee fails. This problem can be addressed by a type system or detecting the free variables of memoized expressions automatically with a static analyzer.

6 Conclusion

Recent experimental results show that it is possible to adjust computations to changes to their data (e.g., inputs, outcomes of comparisons) efficiently by using a combination of change propagation and memoization. This paper formalizes a general semantics for combining memoization and change propagation where memoization is modeled as a non-deterministic oracle, and computation re-use is possible in the presence of mutation. Our main theorem shows that the semantics is consistent with deterministic, purely functional programming.

By giving a general 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. In particular, the results show that some assumptions made by existing implementations are not necessary for correctness and that they may be further improved.

References

1. M. Abadi, B. W. Lampson, and J.-J. Levy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.
2. U. A. Acar. *Self-Adjusting Computation.* PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.
3. U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.
4. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proc. of the 29th Ann. ACM Symp. on POPL*, pages 247–259, 2002.
5. U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Proc. of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.
6. U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.
7. U. A. Acar, G. E. Blelloch, K. Tangwongsan, and J. L. Vittes. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.
8. U. A. Acar, G. E. Blelloch, and J. L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*, 2005.
9. U. A. Acar, M. Blume, and J. Donham. A consistent semantics of self-adjusting computation. Technical Report CMU-CS-06-168, Department of Computer Science, Carnegie Mellon University, 2006.
10. M. Carlsson. Monads for incremental computing. In *Proc. of the 7th ACM SIGPLAN Intl. Conf. on Funct. Prog.*, pages 26–35. ACM Press, 2002.
11. A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.
12. J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LISP and Functional Programming*, pages 307–322, June 1990.
13. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
14. R. Harper and D. Licata. Mechanizing language definitions. (Submitted for publication.), April 2006.
15. D. Michie. 'memo' functions and machine learning. *Nature*, 218:19–22, 1968.
16. F. Pfenning and C. 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.
17. W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.
18. G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 502–510, Jan. 1993.
19. R. S. Sundaresh and P. Hudak. Incremental compilation via partial evaluation. In *Conf. Record of the 18th Ann. ACM Symp. on POPL*, pages 1–13, Jan. 1991.