Self-Adjusting Stack Machines

Matthew A. Hammer Georg Neis Yan Chen Umut A. Acar
Max Planck Institute for Software Systems
{hammer,neis,chenyan,umut}@mpi-sws.org

Abstract

Self-adjusting computation offers a language-based approach to writing programs that automatically respond to dynamically changing data. Recent work made significant progress in developing sound semantics and associated implementations of self-adjusting computation for high-level, functional languages. These techniques, however, do not address issues that arise for low-level languages, i.e., stack-based imperative languages that lack strong type systems and automatic memory management.

In this paper, we describe techniques for self-adjusting computation which are suitable for low-level languages. Necessarily, we take a different approach than previous work: instead of starting with a high-level language with additional primitives to support self-adjusting computation, we start with a low-level intermediate language, whose semantics is given by a stack-based abstract machine. We prove that this semantics is sound: it always updates computations in a way that is consistent with full reevaluation. We give a compiler and runtime system for the intermediate language used by our abstract machine. We present an empirical evaluation that shows that our approach is efficient in practice, and performs favorably compared to prior proposals.

Categories and Subject Descriptors D.1.0 [*Programming Techniques*]: General

General Terms Languages

1. Introduction

Many applications operate on data that changes incrementally, i.e., by a small amount, over time. Such incremental changes often require only incremental updates to the output, making it possible to respond to dynamically changing data more efficiently than recomputing the output from scratch. These improvements are often asymptotically significant, providing as much as a linear factor of speedup. To exploit this potential, one can develop "dynamic" or "kinetic" algorithms that are designed to deal with particular forms of changing input by taking advantage of the particular structure of the problem at hand [9, 14, 17]. This manual approach often yields updates that are asymptotically faster than full reevaluation, but carries inherent complexity and non-compositionality that makes the algorithms difficult to design, analyze, and use.

As an alternative to manual design of dynamic and kinetic algorithms, the programming languages community has developed techniques that either automate or mostly automate the process of translating an implementation of an algorithm for fixed input into a version for changing input. This is a challenging problem because the compiler is expected to improve the asymptotic complexity of the program. Many different approaches have been considered; for more detail on previous work we refer the reader to Ramalingam and Reps' survey [33] and to Section 9. Recent advances on self-adjusting computation [4, 6, 27] made substantial progress on this problem by proposing techniques that allow both purely functional and imperative programs to automatically respond to changes in their data. The approach has been shown to be effective in a reasonably broad range of areas including computational geometry, invariant checking, motion simulation, and machine learning (e.g., [3, 5, 34]) and has even helped solve challenging open problems [7].

Self-adjusting computation typically relies on programmer help to identify the data that can change over time, called *changeable data*, and the dependencies between this data and program code. This changeable data is typically stored in special memory cells referred to as *modifiable references* (*modifiables* for short), so called because they can undergo incremental modification. The read and write dependencies of modifiables are recorded in a dynamic *execution trace* (or *trace*, for short), which effectively summarizes the self-adjusting computation. When modifiables change, the trace is automatically edited through a *change propagation* algorithm: some portions of the trace are reevaluated (when the corresponding subcomputations are affected by a changed value), some portions are discarded (e.g., when reevaluation changes control paths) and some portions are reused (when a subcomputation remains unaffected, i.e.,

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

when it remains consistent with the values of modifiables). We typically say that a semantics for self-adjusting computation is *sound* (alternatively, *consistent*), if the change propagation mechanism always yields a result consistent with full reevaluation.

The initial approaches for self-adjusting computation offer programming interfaces within existing functional languages, namely, SML and Haskell, either via a library [6, 13] or with special compiler support [27]. However, in all these systems, self-adjusting programs have a purely-functional flavor, as modifiables must be written exactly once. Later, Acar et al. lifted this write-once restriction by giving a higher-order imperative semantics for self-adjusting computation [4].

Unfortunately, this imperative semantics is not well-suited for modeling low-level languages—by *low-level* we mean (here and throughout) stack-based imperative languages that lack strong type systems and automatic memory management. First, the imperative semantics assumes that only modifiables are mutable: all other data is implicitly assumed to be immutable. While a strong type system can enforce this policy, in a low-level setting, all data is mutable by default, and there is no strong type system to enforce other policies. Next, the imperative semantics implicitly assumes that all garbage is collected automatically. This includes garbage from the self-adjusting program itself, as well as from updating its trace via change propagation. Such automatic collection cannot be assumed for low-level languages. Finally, and perhaps most importantly, the imperative semantics provides no account of how execution traces should be incrementally edited by the system for reuse. Instead, the semantics effectively relies on an oracle to generate reusable traces, and leaves the internal behavior of this oracle unspecified. Consequently, the oracle hides many of the pratical issues that would otherwise arise, such as how memory allocation and collection interact with trace reuse.

Based on their imperative semantics, Acar et al. describe a library-based implementation for SML [4]. Following this library interface, CEAL [23] provides compiler support to write self-adjusting computations in C. However, because of the issues raised above, the soundness property proven for the semantics generally does not hold for CEAL programs unless they adhere to various correct-usage restrictions. In particular, CEAL programs must only mutate modifiables and local variables—global variables, return values[1], and user-defined data structures must be immutable (and hence, non-modifiable). Furthermore, since even immutable data must first be initialized in a low-level setting, and since this initialization is itself a case of mutation, CEAL programs are required to treat such initialization code in a special way. Namely, they must separate it into designated "initialization functions", as introduced in previous work on automatic memory management for self-adjusting computation [20].

[1] The imperative semantics restricts return types to unit (i.e., `void`).

Failing to follow the correct-usage restrictions given above, a CEAL program could crash, or alternatively, fail to provide correct updates. As a simple example, consider a trivial program that calls two functions: the first copies some input from modifiable m_{in} to a global variable g; the second copies the value of g into another modifiable m_{out} as output. The computational dependencies of modifiable references m_{in} and m_{out} are traced, but those of global variable g are not. Consequently, when m_{in} changes, m_{out} will not be updated, since doing so requires knowledge of its dependency on g. An analogous scenario can be constructed using any non-modifiable memory in place of global g (e.g., a user-defined data type).

At present, we are aware of no generally sound implementation of self-adjusting computation for low-level languages, nor a semantics that suggests one.

Self-adjusting stack machines. In this paper, we describe techniques for sound self-adjusting computation which are suitable for low-level languages. To achieve soundness without losing generality, we take a fundamentally different approach than previous work: instead of starting with a high-level language with additional primitives to support self-adjusting computation, we start with a low-level intermediate language called IL.

We give two semantics to IL by defining two abstract machines: the *reference machine* models conventional evaluation semantics, while the *tracing machine* models self-adjusting semantics. Each machine is defined by a *transition relation* between *machine configurations*. Our low-level setting is reflected by the reference machine's configurations: each consists of a store, a stack, an environment and a program. The tracing machine extends these configurations with an execution trace. We define *traced evaluation* and *change propagation* within the tracing machine by including transitions that incrementally edit the trace (i.e., transitions that either insert, remove or replay traced execution steps). We show that automatic memory management is a natural aspect of automatic change propagation by defining a notion of garbage collection.

Contributions. Our contributions are as follows:

1. We provide an abstract machine semantics for self-adjusting computation. This includes accounts of how change propagation interacts with a control stack, with return values and with memory management. We prove that this semantics is sound.
2. We describe and implement a compiler and runtime system for IL, the intermediate language used by our abstract machines. Additionally, we give two automatic optimizations to reduce the overhead of the approach.
3. We describe and implement a front-end that translates a large subset of C into IL, and perform an empirical evaluation of our implementation.

2. Overview

We introduce the challenges for giving self-adjusting computation support to programs written in low-level languages. In particular, we consider two example programs and consider strategies for incrementally updating their computations. We introduce our approach, in which we restructure these programs in IL, our intermediate language for self-adjusting computation. We informally describe a change propagation semantics for IL programs that addresses the challenges from the examples.

2.1 Example 1: Reducing Trees

For our first example, we consider a simple evaluator for expression trees, as expressed with user-defined C data structures. These expression trees consist of integer-valued leaves and internal nodes that represent the binary operations of addition and subtraction. Figure 1 shows their representation in C. The `tag` field (either `LEAF` or `BINOP`) distinguishes between the `leaf_val` and `binop` fields of the **union** `u`. Figure 2 gives a simple C function that evaluates these trees.

Suppose we first run `eval` with an expression tree as shown on the left in Figure 3; evaluating $((3+4)-0)+(5-6)$, the execution will return the value 6. Suppose we then change the expression tree to $((3+4)-0)+((5-6)+\mathbf{5})$ as shown in Figure 3 on the right. How shall change propagation efficiently update the output?

Strategy for change propagation. We first consider the computation's structure, of which Figure 4 gives a summary: the upper and lower versions summarize the computation before and after the change, respectively. Their structure reflects the stack behavior of `eval`, which divides each invocation into (up to) three fragments: Fragment one checks the tag of the node, returning the leaf value, if present, or else recurring on the left subtree (lines 2–5); fragment two recurs on the right subtree (line 6); and fragment three combines and returns the results (lines 7–8).

In Figure 4, each fragment is labeled with a tree node, e.g., $\mathbf{b}_2$ represents fragment two's execution on node **b**. The dotted horizontal arrows indicate pushing a code fragment on the stack for later. Solid arrows represent the flow of control from one fragment to the next; when diagonal, they indicate popping the stack to continue evaluation.

Based on these two computations' structure, we informally sketch a strategy for change propagation. First, since the left half of the tree is unaffected, the left half of the computation ($\mathbf{a}_1$–$\mathbf{b}_3$) is also unaffected, and as such, change propagation should reuse it. Next, since the right child for **a** has changed, the computation that reads this value, fragment $\mathbf{a}_2$, should be reevaluated. This reevaluation recurs to node **g**, whose subtree has not changed. Hence, change propagation should reuse the corresponding computation ($\mathbf{g}_1$–$\mathbf{g}_3$), including its return value, -1. Comparing $\mathbf{j}_1$–$\mathbf{j}_3$ against $\mathbf{g}_1$–$\mathbf{g}_3$, we see that **a**'s right subtree evaluates to 4 rather

```
typedef struct node_s* node_t;
struct node_s {
  enum { LEAF, BINOP } tag;
  union { int leaf_val;
          struct { enum { PLUS, MINUS } op;
                   node_t left, right; } binop;
        } u; };
```

Figure 1. Type declarations for expression trees in C.

```
1 int eval (node_t root) {
2   if (root->tag == LEAF)
3     return root->u.leaf_val;
4   else {
5     int l = eval (root->u.binop.left);
6     int r = eval (root->u.binop.right);
7     if (root->u.binop.op == PLUS) return (l + r);
8     else return (l - r);
9 } }
```

Figure 2. The `eval` function in C.

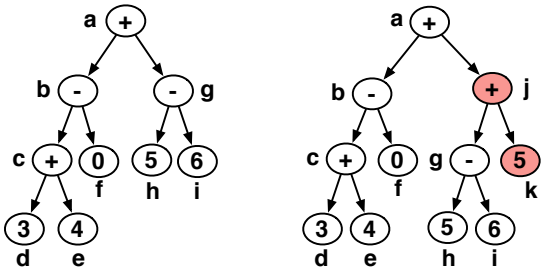

Figure 3. Example expression trees.

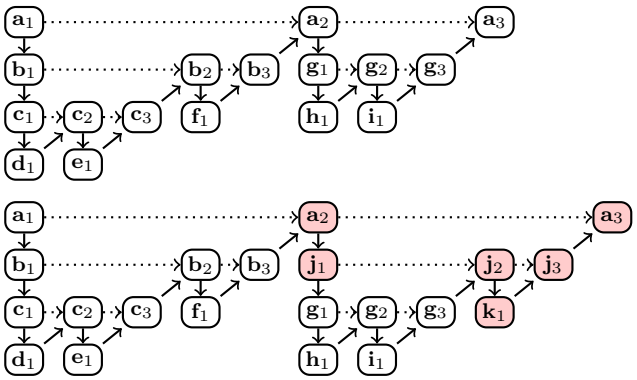

Figure 4. Example execution traces of `eval`.

```
1   let eval (root) = memo
2     let eval_right (l) =
3       let eval_op (r) = update
4         let op = read (root[OP]) in
5         if (op == PLUS) then pop (l+r)
6         else pop (l-r)
7       in
8       push eval_op do update
9         let right = read (root[RIGHT]) in
10        eval (right)
11    in
12    update
13      let tag = read (root[TAG]) in
14      if (tag == LEAF)
15        let leaf_val = read (root[LEAF_VAL]) in
16        pop (leaf_val)
17      else
18        push eval_right do update
19          let left = read (root[LEFT]) in
20          eval (left)
```

Figure 5. The `eval` function in IL.

```
int MAX;

void array_max(int* arr, int len) {
  while(len > 1) {
    for(int i = 0; i < len - 1; i += 2) {
      int m;
      max(arr[i], arr[i + 1], &m);
      arr[i / 2] = m;
    }
    len = len / 2;
  }
  MAX = arr[0];
}
```

Figure 6. Iteratively compute the maximum of an array.

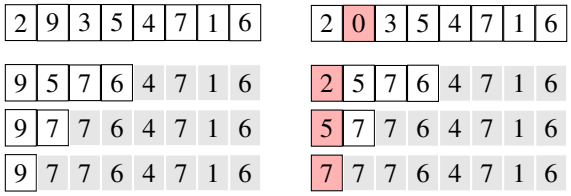

Figure 7. Snapshots of the array from Figure 6.

than -1. Hence, change propagation should reevaluate $\mathbf{a}_3$, to yield the new output of the program, 11.

Challenges. For change propagation to use the strategy sketched above, it must identify dependencies among data and the three-part structure of this code, including its call-/return dependencies. In particular, it must identify where previous computations should be reused, reevaluated or discarded[2]. In Section 2.3, we discuss how the IL code of Figure 5, which represents Figure 2, informs the change propagation strategy described above.

2.2 Example 2: Reducing Arrays

As a second example, Figure 6 gives C code for (destructively) computing the maximum element of an array. Rather than perform a single linear scan, it finds this maximum iteratively by performing a logarithmic number of *rounds*, in the style of a (sequentialized) data-parallel algorithm. For simplicity, we assume that the length of arrays is always a power of two. Each round combines pairs of adjacent elements in the array, producing a sub-sequence with half the length of the original. The remaining half of the array contains inactive elements no longer accessed by the function.

Rather than return values directly, we illustrate commonly used imperative features of C by returning them indirectly: function `max` returns its result by writing to a provided pointer, and `array_max` returns its result by assigning it to a special global variable `MAX`.

Figure 7 illustrates the computation for two (closely-related) example inputs. Below each input, each computation consists of three snapshots of the array, one per round. For readability, the inactive elements of the array are still shown but are greyed, and the differences between the right and left computation are highlighted on the right.

Strategy for change propagation. We use Figure 7 to develop a strategy for change propagation. Recall that each array snapshot summarizes one round of the outer `while` loop. Within each snapshot, each (active) cell summarizes one iteration of the inner `for` loop. That `array_max` uses an iterative style affects the structure of the computation, which consequently admits an efficient strategy for change propagation: reevaluate each affected iteration of the inner `for` loop, that is, those summarized by the highlighted cells in Figure 7.

It is simple to (manually) check that each active cell depends on precisely two cells in the previous round, affects at most one cell in the next round, and is computed independently of other cells in the same round. Hence, for a single input change, at most one such iteration is affected per round. Since the number of rounds is logarithmic in the length of the input array, this change propagation strategy is efficient.

Challenges. To efficiently update the computation, change propagation should reevaluate each affected iteration, being careful not to reevaluate any of the unaffected iterations.

2.3 Introduction to IL

The primary role of IL is to make precise the computational dependencies and possible change propagation behaviors of a low-level self-adjusting program. In particular, it is easy to answer the following questions for a program when expressed in IL:

- Which data dependencies are *local* versus *non-local*?
- Which code fragments are saved on the *control stack*?
- Which computation fragments are saved in the computation's *trace*, for later reevaluation or reuse?

We informally introduce the syntax and semantics of IL by addressing each of these questions for the examples in Sections 2.1 and 2.2. In Section 3, we make the syntax and semantics precise.

Static Single Assignment. To clearly separate local and non-local dependencies, IL employs a (functional variant of) static single assignment form (SSA) [10]. Within this representation, the control-flow constructs of C are represented by locally-defined functions, *local state* is captured by let-bound variables and function parameters, and all *non-local state* (memory content) is explicitly allocated within the store and accessed via **read**s and **write**s.

For example, we express the `for` loop from Figure 6 as the recursive function `for_loop` in Figure 8(a). This function takes an argument for each variable whose definition is dependent on the `for` loop's control flow[3], in this case, just the iteration variable `i`. Within the body of the loop, the local

[2] To see an example where computation is discarded, imagine the change in reverse; that is, changing the lower computation into the upper one.

[3] Where traditional SSA employs ϕ-operators to express control-dependent variable definitions, functional SSA uses ordinary function abstraction.

```
let for_loop (i) =
  let m_ptr = alloc(1) in
  let after_max() = update
    let m_val = read(m_ptr[0]) in
    let _ = write(arr[i/2], m_val) in
    if (i < len - 1)
      then for_loop(i + 2)
      else ...
  in
  push after_max do update
    let a = read(arr[i]) in
    let b = read(arr[i + 1]) in
    max(a, b, m_ptr)
in for_loop(0)
```

(a)

```
let for_loop (i) =
  let m_ptr = alloc(1) in
  let after_max() = update
    let m_val = read(m_ptr[0]) in
    let _ = write(arr[i/2], m_val) in
    memo
      if (i < len - 1)
        then for_loop(i + 2)
        else ...
  in
  push after_max do update
    let a = read(arr[i]) in
    let b = read(arr[i + 1]) in
    max(a, b, m_ptr)
in for_loop (0)
```

(b)

```
let for_loop (i) =
  let for_next () =
    if (i < len - 1) then for_loop(i + 2)
      else ...
  in
  push for_next do
    let m_ptr = alloc(1) in
    let after_max() = update
      let m_val = read(m_ptr[0]) in
      let _ = write(arr[i/2], m_val) in
      pop ()
    in
    push after_max do update
      let a = read(arr[i]) in
      let b = read(arr[i + 1]) in
      max(a, b, m_ptr)
in for_loop(0)
```

(c)

Figure 8. Three versions of IL code for the `for` loop in Figure 6; highlighting indicates their slight differences.

variable `m` is encoded by an explicit store allocation bound to a temporary variable `m_ptr`. Although not shown, global variable `MAX` is handled analogously. This kind of indirection is necessary whenever assignments can occur non-locally (as with global variables like `MAX`) or via pointer indirection (as with local variable `m`). By contrast, local variables `arr`, `i` and `len` are only assigned directly and locally, and consequently, each is a proper SSA variable in Figure 8(a). Similarly, in Figure 2 the assignments to `l` and `r` are direct, and hence, we express each as a proper SSA variable in Figure 5. We explain the other IL syntax from Figures 5 and 8(a) below (**push**, **pop**, **update**, **memo**).

Stack operations. As our first example illustrates (Section 2.1), the control stack necessarily breaks a computation into multiple fragments. In particular, before control flow follows a function call, it first pushes on the stack a code fragment (a local continuation) which later takes control when the call completes.

The stack operations of IL make this code fragmentation explicit: the expression **push** f **do** e saves function f (a code fragment expecting zero or more arguments) on the stack and continues by evaluating e; when this subcomputation **pop**s the stack, the saved function f is applied to the (zero or more) arguments of the **pop**.

In Figure 5, the two recursive calls to `eval` are preceded by **push**es that save functions `eval_right` and `eval_op`, corresponding to code fragments for evaluating the right subtree (fragment two) and applying the binary operator (fragment three), respectively. Similarly, in Figure 8(a), the call to `max` is preceded by a **push** that saves function `after_max`, corresponding to the code fragment following the call. We note that since `max` returns no values, `after_max` takes no arguments.

Reevaluation and reuse. To clearly mark which computations are saved in the trace—which in turn defines which computations can be reevaluated and reused—IL uses the special forms **update** and **memo**, respectively.

The IL expression **update** e, which we call an *update point*, has the same meaning as e, except that during change propagation, the computation of e can be recovered from the program's original computation and reevaluated. This reevaluation is necessary exactly when the original computation of e contains **read**s from the store that are no longer consistent within the context of new computation.

Dually, the IL expression **memo** e, which we call a *memo point*, has the same meaning as e, except that during reevaluation, a previous computation of e can be reused in place the present one, provided that they *match*. Two computations of the same expression e match if they begin in locally-equivalent states (same local state, but possibly different non-local state). This notion of memoization is similar to function caching [32] in that it reuses past computation to avoid reevaluation, but it is also significantly different in that impure code is supported, and non-local state need not match (a matching computation may contain inconsistent **read**s). We correct inconsistencies by reevaluating each inconsistent **read** within the reused computation.

We can insert **update** and **memo** points freely within an existing IL program without changing its meaning (up to reevaluation and reuse behavior). Since they allow more fine-grained reevaluation and reuse, one might want to insert them before and after every instruction in the program. Unfortunately, each such insertion incurs some tracing overhead, as **memo** and **update** points each necessitate saving a snapshot of local state.

Fortunately, we can automatically insert a smaller yet equally effective set of **update** points by focusing only on **read**s. Figures 5 and 8(a) show examples of this: since each **read** appears within the body of an **update** point, we can reevaluate these **read**s, including the code that depends on them, should they become inconsistent with memory. We say that each such **read** is *guarded* by an **update** point.

For memo points, however, it is less clear how to automatically strike the right balance between too many (too much overhead) and not enough (not enough reuse). Instead, we

expose surface syntax to the C programmer, who can insert them as statements (`memo;`) as well as expressions (e.g., `memo(f(x))`). In Section 2.4, we discuss where to place memo points within our running examples.

2.4 Change Propagation Strategies Revisited

In Sections 2.1 and 2.2, we sketched strategies for updating computations using change propagation. Based on the IL representations described in Section 2.3, we informally describe our semantics for change propagation in greater detail. The remainder of the paper makes this semantics precise and describes our current implementation.

Computations as traces. We represent computations using an execution trace, which records the **memo** and **update** points, store operations (**alloc**s, **read**s and **write**s), and stack operations (**push** and **pop**).

To a first approximation, change propagation of these traces has two aspects: reevaluating inconsistent subtraces, and reusing consistent ones. Operationally, these aspects mean that we need to decide not only which computations in the trace to reevaluate, but also where this reevaluation should cease.

Beginning a reevaluation. In order to repair inconsistencies in the trace, we begin reevaluations at **update** points that guard inconsistent **read**s. We identify **read**s as inconsistent when the memory location they depend on is affected by **write**s being inserted into or removed from the trace. That is, a **read** is identified as *affected* in one of two ways: when inserting a newly traced **write** (of a different value) that becomes the newly read value, or when removing a previously traced **write** that had been the previously read value. In either case, the **read** in question becomes inconsistent and cannot be reused in the trace without first being reevaluated. To begin such a reevalaution, we restore the local state from the trace and reevaluate within the context of the current memory and control stack, which generally both differ from those of the original computation.

Ending a reevaluation. We end a reevaluation in one of two ways. First, recall that we begin reevaluation with a different control stack than that used by the original computation. Hence, we will eventually encounter a **pop** that we cannot correctly reevaluate, as doing so requires knowing the contents of the original computation's stack. Instead, we cease reevaluation at such **pop**s. We justify this behavior below and describe how it still leads to a sound approach.

Second, as described in Section 2.3, when we encounter a **memo** point, we may find a matching computation to reuse. If so, we cease the current reevaluation and begin reevaluations that repair inconsistencies within the reused computation, if any.

Example 1 revisited. The strategy from Section 2.1 requires that the previous computation be reevaluated in some places, and reused in others. First, as Figure 5 shows, we note that however an input tree is modified, **update** points guard the computation's affected **read**s. We reevaluate these update points. For instance, in the given change (of the right subtree of **a**), line 9 has the first affected **read**, which is guarded by an **update** point on line 8; this point corresponds to $\mathbf{a}_2$, which we reevaluate first. Second, our strategy reuses computation $\mathbf{g}_1$–$\mathbf{g}_3$. To this end, we can insert a `memo` statement at the beginning of function `eval` in Figure 2 (not shown), resulting in the **memo** point shown on line 1 in Figure 5. Since it precedes each invocation, this **memo** point allows for the desired reuse of unaffected subcomputations.

Example 2 revisited. Recall that our strategy for Section 2.2 consists of reevaluating iterations of the inner `for` loop that are affected, and reusing those that are not. To begin each reevaluation within this loop (Figure 8(a)), we reevaluate their **update** points.

Now we consider where to cease reevaluation. Note that the **update** point in `after_max` guards a **read**, as well as the recursive use of `for_loop`, which evaluates the remaining (possibly unaffected) iterations of the loop. However, recall that we do not want reevaluation to continue with the remaining iterations—we want to reuse them.

We describe two ways to cease reevaluation and enable reuse. First, we can insert a `memo` statement at the end of the inner `for` loop in Figure 6, resulting in the **memo** point shown in Figure 8(b). Second, we can wrap the `for` loop's body with a *cut block*, written `cut{...}`, resulting in the additional **push**-**pop** pair in Figure 8(c). Cut blocks are optional but convenient syntactic sugar: their use is equivalent to moving a code block into a separate function (hence the **push**-**pop** pair in Figure 8(c)). Regardless of which we choose, the new **memo** and **pop** both allow us to cease reevaluation immediately after an iteration is reevaluated within Figures 8(b) and 8(c), respectively.

Call/return dependencies. Recall from Section 2.1 that we must be mindful of call/return dependencies among the recursive invocations. In particular, after reevaluating a subcomputation whose return value changes, the consumer of this return value (another subcomputation) is affected and should be reevaluated ($\mathbf{a}_3$ in the example).

Our general approach for call/return dependencies has three parts. First, when proving consistency (Section 4), we restrict our attention to programs whose subcomputations' return values do not change, a crucial property of programs that we make precise in Section 4. Second, in Section 5, we provide an automatic transformation of arbitrary programs into ones that have this property. Third, in Section 6.3, we introduce one simple way to refine this transformation to reduce the overhead that it adds to the transformed programs. With more aggressive analysis, we expect that further efficiency improvements are possible.

Contrasted with proving consistency for a semantics where a fixed approach for call/return dependencies is "baked in", our consistency proof is more general. It stip-

e	::=	e^{u}	Untraced expression
	|	e^{t}	Traced expression
e^{u}	::=	**let fun** $f(\overline{x})$.e_1 **in** e_2	Function definition
	|	**let** x **=** $\oplus(\overline{v})$ **in** e	Primitive operation
	|	**if** x **then** e_1 **else** e_2	Conditional
	|	$f\ (\overline{x})$	Function application
e^{t}	::=	**let** x **=** ι **in** e	Store instruction
	|	**memo** e	Memo point
	|	**update** e	Update point
	|	**push** f **do** e	Stack push
	|	**pop** $\overline{x}$	Stack pop
ι	::=	**alloc**(x)	Allocate an array of size x
	|	**read**$(x[y])$	Read yth entry at x
	|	**write**$(x[y],z)$	Write z as yth entry at x
v	::=	$n \mid x$	Natural numbers, variables

Figure 9. IL syntax.

ulates a property that can be guarenteed by either of the two transformations that we describe (Sections 5 and 6.3). Furthermore, it leaves the possibility open for future work to improve the currently proposed transformations, e.g., by employing more sophosticated static analysis to further reduce the overhead that they introduce.

2.5 Guide for the Paper

Section 3 presents the abstract machine semantics for IL, including our change propagation semantics. Section 4 presents our consistency theorem. Section 5 presents a *destination-passing style* transformation whose target programs meet our side condition for consistency. Section 6 gives compilation and runtime techniques for our semantics. Section 7 describes our implementation. Section 8 gives an empirical evaluation. Sections 9 and 10 give related work and conclude.

3. A Self-Adjusting Intermediate Language

We present IL, a self-adjusting intermediate language, as well as two abstract machines that evaluate IL syntax. We call these the *reference machine* and the *tracing machine*, respectively. As its name suggests, we use the first machine as a reference when defining and reasoning about the tracing machine. Each machine is defined by its own transition relation over similar machine components. The tracing machine mirrors the reference machine, but includes additional machine state components and transition rules that work together to generate and edit execution traces. This tracing behavior formalizes the notion of IL as a self-adjusting language.

3.1 Abstract Syntax of IL

Figure 9 shows the abstract syntax for IL. Programs in IL are expressions, which we partition into traced e^{t} and untraced e^{u}. This distinction does not constrain the language; it merely streamlines the technical presentation. Expressions in IL follow an administrative normal form (ANF) [19] where (nearly) all values are variables.

Expressions consist of function definitions, primitive operations, conditionals, function calls, store instructions (ι), **memo** points, **update** points, and operations for pushing (**push**) and popping (**pop**) the stack. Store instructions (ι) consist of operations for allocating (**alloc**), reading (**read**) and writing (**write**) memory. Values v include natural numbers and variables (but not function names). Each expression ends syntactically with either a function call or a stack **pop** operation. Since the form for function calls is syntactically in tail position, the IL program must explicitly push the stack to perform non-tail calls. Expressions terminate when they **pop** on an empty stack—they yield the values of this final pop.

Notice that IL programs are first-order: although functions can nest syntactically, they are not values; moreover, function names f, g, h are syntactically distinct from variables x, y, z. Supporting either first-class functions (functions as values) or function pointers is beyond the scope of the current work, though we believe our semantics could be adapted for these settings[4].

In the remainder, we restrict our attention to programs (environments ρ and expressions e) that are well-formed in the following sense:

1. They have a unique arity (the length of the value sequence they potentially return) that can be determined syntactically.
2. All variable and function names therein are distinct. (This can easily be implemented in a compiler targeting IL.) Consequently we don't have to worry about the fact that IL is actually dynamically scoped.

3.2 Machine Configurations and Transitions

In addition to sharing a common expression language (viz. IL, Section 3.1), the reference and tracing machines share common *machine components*; they also have related *transition relations*, which specify how these machines change their components as they run IL programs.

Machine configurations. Each machine configuration consists of a handful of components. Figure 10 defines the common components of two machines: a store (σ), a stack (κ), an environment (ρ) and a command (α_{r} for the reference machine, and α_{t} for the tracing machine). The tracing machine

[4] For example, to model function pointers, one could adapt this semantics to allow a function f to be treated as a value if f is closed by its arguments; this restriction models the way that functions in C admit function pointers, a kind of "function as a value", even though C does not include features typically associated with first-class functions (e.g. implicitly-created closures, partial application).

has an additional component—its trace—which we describe in Sections 3.4 and 3.5.

A store σ maps each store *entry* ($\ell[n]$) to either *uninitialized* contents (written $\perp$) or a machine value ν. Each entry $\ell[n]$ consists of a store location ℓ and a (natural number) offset n. In addition, a store may mark a location as garbage, denoted as $\ell \mapsto \diamond$, in which case all store entries for ℓ are undefined. These garbage locations are not used in the reference semantics; in the tracing machine, they help to define a notion of garbage collection. A stack κ is a (possibly empty) sequence of *frames*, where each frame $\lfloor \rho, f \rfloor$ saves an evaluation context that consists of an environment ρ and a function f (defined in ρ). An *environment* ρ maps variables to machine values and function names to their definitions.

In the case of the reference machine, a *(reference) command* α_{r} is either an IL expression e or a sequence of machine values $\overline{\nu}$; for the tracing machine, a *(tracing) command* α_{t} is either e, $\overline{\nu}$, or an additional command **prop**, which indicates that the machine is performing *change propagation* (i.e., replay of an existing trace).

Each *machine value* ν consists of a natural number n or a store location ℓ. Intuitively, we think of machine values as corresponding to machine words, and we think of the store as mapping location-offset pairs (each of which is itself a machine word) to other machine words.

For convenience, when we do not care about individual components of a machine configuration (or some other syntactic object), we often use underscores (_) to avoid giving them names. The quantification should always be clear from context.

Transition relations. In the reference machine, each machine configuration, written $\sigma, \kappa, \rho, \alpha_{\mathsf{r}}$, consists of four components: a store, a stack, an environment and a command, as described above. In Section 3.3, we formalize the following stepping relation for the reference machine:

$$\sigma, \kappa, \rho, \alpha_{\mathsf{r}} \xrightarrow{\mathsf{r}} \sigma', \kappa', \rho', \alpha_{\mathsf{r}}{}'$$

Intuitively, the command α_{r} tells the reference machine what to do next. In the case of an expression e, the machine proceeds by evaluating e, and in the case of machine values $\overline{\nu}$, the machine proceeds by popping a stack frame $\lfloor \rho, f \rfloor$ and using it as the new evaluation context. If the stack is empty, the machine terminates and the command $\overline{\nu}$ can be viewed as giving the machine's results. Since these results may consist of store locations, the complete extensional result of the machine must include the store (or at least, the portion reachable from $\overline{\nu}$).

The tracing machine has similar machine configurations, though it also includes a pair $\langle \Pi, T \rangle$ that represents the current trace, which may be in the midst of adjustment; we describe this component separately in Sections 3.4 and 3.5. In Section 3.6, we formalize the following stepping relation for the tracing machine:

$$\langle \Pi, T \rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{t}} \xrightarrow{\mathsf{t}} \langle \Pi', T' \rangle, \sigma', \kappa', \rho', \alpha_{\mathsf{t}}{}'$$

Store	σ	::=	$\varepsilon \mid \sigma[\ell[n] \mapsto \perp]$
		$\mid$	$\sigma[\ell[n] \mapsto \nu] \mid \sigma[\ell \mapsto \diamond]$
Stack	κ	::=	$\varepsilon \mid \kappa \cdot \lfloor \rho, f \rfloor$
Environment	ρ	::=	$\varepsilon \mid \rho[x \mapsto \nu]$
		$\mid$	$\rho[f \mapsto \textbf{fun}\ f(\overline{x}).e]$
Reference command	α_{r}	::=	$e \mid \overline{\nu}$
Tracing command	α_{t}	::=	$\alpha_{\mathsf{r}} \mid \textbf{prop}$
Machine value	ν	::=	$n \mid \ell$

Figure 10. Common machine components.

At a high level, this transition relation accomplishes several things: (1) it "mirrors" the semantics of the reference machine when evaluating IL expressions; (2) it traces this evaluation, storing the generated trace within its trace component; and (3) it allows previously-generated traces to be either reused (during change propagation), or discarded (when they cannot be reused). To accomplish these goals, the tracing machine distinguishes machine transitions for change propagation from those of normal execution by giving change propagation the distinguished command **prop**.

3.3 Reference Machine Transitions

Figure 11 specifies the transition relation for the reference machine, as introduced in Section 3.2. A function definition updates the environment, binding the function name to its definition. A primitive operation first converts each value argument v_i into a machine value ν_i using the environment. Here we abuse notation and write $\rho(v)$ to mean $\rho(x)$ when $v = x$ and n when $v = n$. The machine binds the result of the primitive operation (as defined by the abstract **primapp** function) to the given variable in the current environment. A conditional steps to the branch specified by the scrutinee. A function application steps to the body of the specified function after updating the environment with the given arguments. A store instruction ι steps using an auxiliary judgement (Figure 12) that allocates in, reads from and writes to the current store. An **alloc** instruction allocates a fresh location ℓ for which each offset (from 1 to the specified size) is marked as uninitialized. A **read** (resp. **write**) instruction reads (resp. writes) the store at a particular location and offset. A **push** expression saves a return context in the form of a stack frame $\lfloor \rho, f \rfloor$ and steps to the body of the **push**. A **pop** expression steps to a machine value sequence $\overline{\nu}$, as specified by a sequence of variables. If the stack is non-empty, the machine passes control to function f, as specified by the topmost stack frame $\lfloor \rho, f \rfloor$, by applying f to $\overline{\nu}$; it recovers the environment ρ before discarding this frame. Otherwise, if the stack is empty, the value sequence $\overline{\nu}$ signals the termination of the machine with results $\overline{\nu}$.

$$\frac{\rho' = \rho[f \mapsto \textbf{fun}\ f(\overline{x}).e_1]}{\sigma, \kappa, \rho, \textbf{let fun}\ f(\overline{x}).e_1\ \textbf{in}\ e_2 \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho', e_2}\ \textbf{R.1}$$

$$\frac{\rho(v_i)_{i=1}^{|\overline{v}|} = \overline{\nu} \qquad \rho' = \rho[x \mapsto \textbf{primapp}(\oplus, \overline{\nu})]}{\sigma, \kappa, \rho, \textbf{let}\ x = \oplus(\overline{v})\ \textbf{in}\ e \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho', e}\ \textbf{R.2}$$

$$\frac{\rho(x) \neq 0}{\sigma, \kappa, \rho, \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho, e_1}\ \textbf{R.3}$$

$$\frac{\rho(x) = 0}{\sigma, \kappa, \rho, \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho, e_2}\ \textbf{R.4}$$

$$\frac{\rho(f) = \textbf{fun}\ f(\overline{x}).e \qquad \rho' = \rho[x_i \mapsto \rho(x_i)]_{i=1}^{|\overline{x}|}}{\sigma, \kappa, \rho, f\ (\overline{x}) \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho', e}\ \textbf{R.5}$$

$$\frac{\sigma, \rho, \iota \xrightarrow{s} \sigma', \nu}{\sigma, \kappa, \rho, \textbf{let}\ x = \iota\ \textbf{in}\ e \xrightarrow{\mathsf{r}} \sigma', \kappa, \rho[x \mapsto \nu], e}\ \textbf{R.6}$$

$$\frac{}{\sigma, \kappa, \rho, \textbf{memo}\ e \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho, e}\ \textbf{R.7}$$

$$\frac{}{\sigma, \kappa, \rho, \textbf{update}\ e \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho, e}\ \textbf{R.8}$$

$$\frac{}{\sigma, \kappa, \rho, \textbf{push}\ f\ \textbf{do}\ e \xrightarrow{\mathsf{r}} \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \rho, e}\ \textbf{R.9}$$

$$\frac{\overline{\nu} = \rho(x_i)_{i=1}^{|\overline{x}|}}{\sigma, \kappa, \rho, \textbf{pop}\ \overline{x} \xrightarrow{\mathsf{r}} \sigma, \kappa, \varepsilon, \overline{\nu}}\ \textbf{R.10}$$

$$\frac{\rho(f) = \textbf{fun}\ f(\overline{x}).e \qquad \rho' = \rho[x_i \mapsto \nu_i]_{i=1}^{|\overline{x}|}}{\sigma, \kappa \cdot \lfloor \rho, f \rfloor, \varepsilon, \overline{\nu} \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho', e}\ \textbf{R.11}$$

Figure 11. Stepping relation for reference machine ($\xrightarrow{\mathsf{r}}$).

$$\frac{\ell \notin \mathsf{dom}(\sigma) \qquad \sigma' = \sigma[\ell[i] \mapsto \bot]_{i=1}^{\rho(x)}}{\sigma, \rho, \textbf{alloc}(x) \xrightarrow{s} \sigma', \ell}\ \textbf{S.1}$$

$$\frac{\sigma(\rho(x)[\rho(y)]) = \nu}{\sigma, \rho, \textbf{read}(x[y]) \xrightarrow{s} \sigma, \nu}\ \textbf{S.2}$$

$$\frac{\sigma' = \sigma[\rho(x)[\rho(y)] \mapsto \rho(z)]}{\sigma, \rho, \textbf{write}(x[y], z) \xrightarrow{s} \sigma', 0}\ \textbf{S.3}$$

Figure 12. Stepping relation for store instructions ($\xrightarrow{s}$).

3.4 The Structure of the Trace

The structure of traces used by the tracing machine is specified by Figure 13. They each consist of a (possibly empty) sequence of zero or more *trace actions* t. Each action records a transition for a corresponding traced expression e^{t}.

In the case of store instructions, the corresponding action indicates both the instruction and each machine value involved in its evaluation. For **alloc**s, the action $\mathsf{A}_{\ell,n}$ records the allocated location as well as its size (i.e., the range of offsets it defines). For **read**s ($\mathsf{R}^{\nu}_{\ell[n]}$) and **write**s ($\mathsf{W}^{\nu}_{\ell[n]}$) the action stores the location and offset being accessed, as well as the machine value being read or written, respectively. For

$$\begin{array}{rrcl}
\textit{Trace} & T & ::= & t \cdot T \mid \varepsilon \\
\textit{Trace Action} & t & ::= & \mathsf{A}_{\ell,n} \mid \mathsf{R}^{\nu}_{\ell[n]} \mid \mathsf{W}^{\nu}_{\ell[n]} \mid \mathsf{M}_{\rho,e} \mid \mathsf{U}_{\rho,e} \\
 & & \mid & (T) \mid \overline{\nu} \\
\textit{Trace Context} & \Pi & ::= & \varepsilon \mid \Pi \cdot t \mid \Pi \cdot \Box \mid \Pi \cdot \boxplus_T \mid \Pi \cdot \boxminus_T
\end{array}$$

Figure 13. Traces, trace actions and trace contexts.

memo expressions, the trace action $\mathsf{M}_{\rho,e}$ records the body of the memo point, as well as the current environment at this point; **update** expressions are traced analogously. For **push** expressions, the action (T) records the trace of evaluating the **push** body; it is significant that in this case, the trace action is not atomic: it consists of the arbitrarily large subtrace T. For **pop** expressions, the action $\overline{\nu}$ records the machine values being returned via the stack.

There is a close relationship between the syntax of traced expressions in IL and the structure of their traces. For instance, in nearly all traced expressions, there is exactly one subexpression, and hence their traces $t \cdot T$ contain exactly one subtrace, T. The exception to this is **push**, which can be thought of as specifying two subexpressions: the first subexpression is given by the body of the **push**, and recorded within the push action as (T); the second subexpression is the body of the function being pushed, which is evaluated when the function is later popped. Hence, push expressions generate traces of the form $(T) \cdot T'$, where T' is the trace generated by evaluating the pushed/popped function.

3.5 Trace Contexts and the Trace Zipper

As described above, our traces are not strictly sequential structures: they also consist of nested subtraces created by **push**. This fact poses a technical challenge for transition semantics (and by extension, an implementation). For instance, while generating such a subtrace, how should we maintain the context of the trace that will eventually enclose it?

To address this, the machine augments the trace with a *context* (Figure 13), maintaining in each configuration both a *reuse trace* T, which we say is *in focus*, as well as an unfocused trace context Π. The trace context effectively records a path from the focus back to the start of the trace. To move the focus in a consistent manner, the machine places additional markings $\Box$, $\boxminus$, $\boxplus$ into the context; two of these markings (viz. $\boxminus$, $\boxplus$) also carry a subtrace. We describe these markings and their subtraces in more detail below.

This pair of components $\langle \Pi, T \rangle$ forms a kind of *trace zipper*. More generally, a zipper augments a data structure with a *focus* (for zipper $\langle \Pi, T \rangle$, we say that T is *in focus*), the ability to perform local edits at the focus and the ability to move this focus throughout the structure [2, 26]. A particularly attractive feature of zippers is that the "edits" can be performed in a non-destructive, incremental fashion.

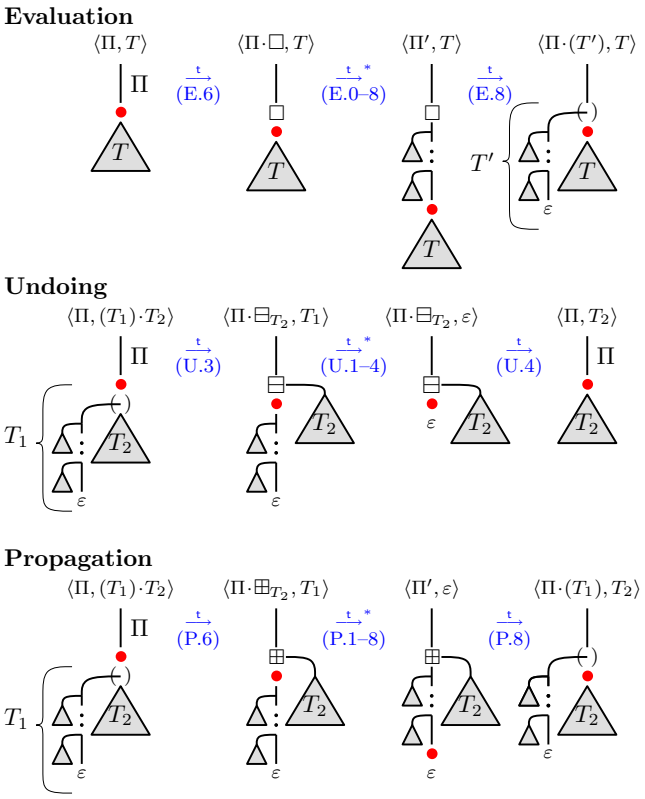

Figure 14. Tracing transition modes, across push actions.

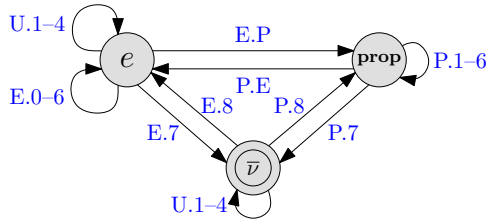

Figure 15. Tracing machine: commands and transitions.

To characterize focus movement using trace zippers, we define the *transition modes* of the tracing machine:

- **Evaluation** mirrors the transitions of the reference machine and generates new trace actions, placing them behind the focus, i.e., $\langle \Pi, T \rangle$ becomes $\langle \Pi \cdot t, T \rangle$.
- **Undoing** removes actions from the reuse trace, just ahead of the focus, i.e., $\langle \Pi, t \cdot T \rangle$ becomes $\langle \Pi, T \rangle$.
- **Propagation** replays the actions of the reuse trace; it moves the focus through it action by action, i.e., $\langle \Pi, t \cdot T \rangle$ becomes $\langle \Pi \cdot t, T \rangle$.

If we ignore **push** actions and their nested subtraces (T), the tracing machine moves the focus in the manner just described, either generating, undoing or propagating at most one trace action for each machine transition. However, since **push** actions consist of an entire subtrace T, the machine cannot generate, undo or propagate them in a single step. Rather, the machine must make a series of transitions, possibly interleaving transition modes. When this process completes and the machine moves its focus out of the subtrace, it is crucial that it does so in a manner consistent with its mode upon entering the subtrace. To this end, the machine may extend the context Π with one of three possible markings, each corresponding to a mode.

For each transition mode, Figure 14 gives both syntactic and pictorial representations of the focused traces and illustrates how the machine moves its focus. The transitions are labeled with corresponding transition rules from the tracing machine, but at this time the reader can ignore them. For each configuration, the (initial) trace context is illustrated with a vertical line, the focus is represented by a filled circle and the (initial) reuse trace is represented by a tree-shaped structure that hangs below the focus.

Evaluation. To generate a new subtrace in evaluation mode (via a **push**), the machine extends the context Π to $\Pi \cdot \square$; this effectively marks the beginning of the new subtrace. The machine then performs evaluation transitions that extend the context, perhaps recursively generating nested subtraces in the process (drawn as smaller, unlabeled triangles hanging to the left). After evaluating the **pop** matching the initial **push**, the machine *rewinds* the current context Π', moving the focus back to the mark $\square$, gathering actions and building a completed subtrace T'; it replaces the mark with a push action (T') (consisting of the completed subtrace), and it keeps reuse trace T in focus. We specify how this rewinding works in Section 3.6; intuitively, it simply moves the focus backwards, towards the start of the trace.

Undoing. To undo a subtrace T_1 of the reuse trace $(T_1) \cdot T_2$, the machine extends the context Π to $\Pi \cdot \boxminus_{T_2}$; this effectively saves the remaining reuse trace T_2 for either further undo transitions or for eventual reuse. Assuming that the machine undoes all of T_1, it will eventually focus on an empty trace ε. In this case, the machine can move the saved subtrace T_2 into focus (again, for either further undo transitions or for reuse).

Propagation. Finally, to propagate a subtrace T_1, the machine uses an approach similar to undoing: it saves the remaining trace T_2 in the context using a distinguished mark $\boxplus_{T_2}$, moves the focus to the end of T_1 and eventually places T_2 into focus. In contrast to the undo transitions, however, propagation transitions do not discard the reuse trace, but only move the focus by moving trace actions from the reuse trace into the trace context. Just as in evaluation mode, in propagation mode we rewind these actions from the context and move the focus back to the propagation mark ($\boxplus$).

We note that while our semantics characterizes change propagation using a step-by-step replay of the trace, this does not yield an efficient algorithm. In Section 6.1, we give an efficient implementation that is faithful to this replay semantics, but in which the change propagation transitions have zero cost.

Evaluation

E.0	$\langle \Pi, T\rangle, \sigma, \kappa, \rho, e^{\mathsf{u}}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi, T\rangle, \sigma, \kappa, \rho', e$	when	$\sigma, \kappa, \rho, e^{\mathsf{u}} \xrightarrow{\mathsf{r}} \sigma, \kappa, \rho', e$
E.1	$\langle \Pi, T\rangle, \sigma, \kappa, \rho,$ **let** $x =$ **alloc**(y) **in** e	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{A}_{\ell,\rho(y)}, T\rangle, \sigma', \kappa, \rho[x \mapsto \ell], e$	when	$\sigma, \rho,$ **alloc**$(y) \xrightarrow{s} \sigma', \ell$
E.2	$\langle \Pi, T\rangle, \sigma, \kappa, \rho,$ **let** $x =$ **read**$(y[z])$ **in** e	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{R}^{\nu}_{\rho(y)[\rho(z)]}, T\rangle, \sigma, \kappa, \rho[x \mapsto \nu], e$	when	$\sigma, \rho,$ **read**$(y[z]) \xrightarrow{s} \sigma, \nu$
E.3	$\langle \Pi, T\rangle, \sigma, \kappa, \rho,$ **let** $\_ =$ **write**$(x[y],z)$ **in** e	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{W}^{\rho(z)}_{\rho(x)[\rho(y)]}, T\rangle, \sigma', \kappa, \rho, e$	when	$\sigma, \rho,$ **write**$(x[y],z) \xrightarrow{s} \sigma', 0$
E.4	$\langle \Pi, T\rangle, \sigma, \kappa, \rho,$ **memo** e	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{M}_{\rho,e}, T\rangle, \sigma, \kappa, \rho, e$		
E.5	$\langle \Pi, T\rangle, \sigma, \kappa, \rho,$ **update** e	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{U}_{\rho,e}, T\rangle, \sigma, \kappa, \rho, e$		
E.6	$\langle \Pi, T\rangle, \sigma, \kappa, \rho,$ **push** f **do** e	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \Box, T\rangle, \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \rho, e$		
E.7	$\langle \Pi, T\rangle, \sigma, \kappa, \rho,$ **pop** $\overline{x}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \overline{\nu}, T\rangle, \sigma, \kappa, \varepsilon, \overline{\nu}$	when	$\overline{\nu} = \rho(x_i)_{i=1}^{\lvert\overline{x}\rvert}$
E.8	$\langle \Pi, T_2\rangle, \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \varepsilon, \overline{\nu}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi' \cdot (T_1), T_2'\rangle, \sigma, \kappa, \rho', e$	when	$\langle \Pi, T_2\rangle ; \varepsilon \circlearrowleft^* \langle \Pi' \cdot \Box, T_2'\rangle ; T_1$
				and	$\rho(f) =$ **fun** $f(\overline{x})$**.**e
				and	$\rho' = \rho[x_i \mapsto \nu_i]_{i=1}^{\lvert\overline{x}\rvert}$

Reevaluation and reuse

P.E	$\langle \Pi, \mathsf{U}_{\rho,e} \cdot T\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{U}_{\rho,e}, T\rangle, \sigma, \kappa, \rho, e$		
E.P	$\langle \Pi, \mathsf{M}_{\rho,e} \cdot T\rangle, \sigma, \kappa, \rho,$ **memo** e	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{M}_{\rho,e}, T\rangle, \sigma, \kappa, \varepsilon,$ **prop**		

Propagation

P.1	$\langle \Pi, \mathsf{A}_{\ell,n} \cdot T\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{A}_{\ell,n}, T\rangle, \sigma', \kappa, \varepsilon,$ **prop**	when	$\sigma, \varepsilon,$ **alloc**$(n) \xrightarrow{s} \sigma', \ell$
P.2	$\langle \Pi, \mathsf{R}^{\nu}_{\ell[n]} \cdot T\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{R}^{\nu}_{\ell[n]}, T\rangle, \sigma, \kappa, \varepsilon,$ **prop**	when	$\sigma, \varepsilon,$ **read**$(\ell[n]) \xrightarrow{s} \sigma, \nu$
P.3	$\langle \Pi, \mathsf{W}^{\nu}_{\ell[n]} \cdot T\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{W}^{\nu}_{\ell[n]}, T\rangle, \sigma', \kappa, \varepsilon,$ **prop**	when	$\sigma, \varepsilon,$ **write**$(\ell[n],\nu) \xrightarrow{s} \sigma', 0$
P.4	$\langle \Pi, \mathsf{M}_{\rho,e} \cdot T\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{M}_{\rho,e}, T\rangle, \sigma, \kappa, \varepsilon,$ **prop**		
P.5	$\langle \Pi, \mathsf{U}_{\rho,e} \cdot T\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \mathsf{U}_{\rho,e}, T\rangle, \sigma, \kappa, \varepsilon,$ **prop**		
P.6	$\langle \Pi, (T_1) \cdot T_2\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \boxplus_{T_2}, T_1\rangle, \sigma, \kappa, \varepsilon,$ **prop**		
P.7	$\langle \Pi, \overline{\nu}\rangle, \sigma, \kappa, \varepsilon,$ **prop**	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \overline{\nu}, \varepsilon\rangle, \sigma, \kappa, \varepsilon, \overline{\nu}$		
P.8	$\langle \Pi, \varepsilon\rangle, \sigma, \kappa, \varepsilon, \overline{\nu}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi' \cdot (T_1), T_2\rangle, \sigma, \kappa, \varepsilon,$ **prop**	when	$\langle \Pi, \varepsilon\rangle ; \varepsilon \circlearrowleft^* \langle \Pi' \cdot \boxplus_{T_2}, \varepsilon\rangle ; T_1$

Undoing

U.1	$\langle \Pi, \mathsf{A}_{\ell,n} \cdot T\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi, T\rangle, \sigma[\ell \mapsto \diamond], \kappa, \rho, \alpha_{\mathsf{r}}$		
U.2	$\langle \Pi, t \cdot T\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi, T\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$	when	$(t = \mathsf{R}^{\text{-}}_{\text{-}[\text{-}]} \mid \mathsf{W}^{\text{-}}_{\text{-}[\text{-}]} \mid \mathsf{M}_{\text{-},\text{-}} \mid \mathsf{U}_{\text{-},\text{-}} \mid \overline{\nu})$
U.3	$\langle \Pi, (T_1) \cdot T_2\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi \cdot \boxminus_{T_2}, T_1\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$		
U.4	$\langle \Pi \cdot \boxminus_T, \varepsilon\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$	$\xrightarrow{\mathsf{t}}$	$\langle \Pi, T\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$		

Figure 16. Stepping relation for tracing machine ($\xrightarrow{\mathsf{t}}$).

3.6 Tracing Machine Transitions

We use the components and transitions of the reference machine (Sections 3.2 and 3.3, respectively) as a basis for defining the transitions of the tracing machine. You may recall from Section 3.2 that the tracing machine extends the reference machine in two important ways.

First, the machine configurations of the tracing machine extend the reference configurations with an extra component $\langle \Pi, T\rangle$, the trace zipper (Section 3.5), which augments the trace structure T (Section 3.4) with a trace context and a movable focus.

Second, a tracing command α_{t} consists of either a reference command α_{r} or the additional propagation command **prop**, which indicates that the machine is doing change propagation. Using these two extensions of the reference machine, the tracing machine generates traces of execution (during evaluation transitions), discards parts of previously-generated traces (during undoing transitions), and reuses previously-generated traces (during propagation transitions).

These three transition modes (evaluation, undoing and propagation) can interact in ways that are not straightforward. Figure 15 helps illustrate their interrelationships, giving us a guide for the transition rules of the tracing machine. The arcs indicate the machine command before and after the machine applies the indicated transition rule. Figure 16 gives the complete transition relation for the tracing machine. Recall that each transition is of the form:

$$\langle \Pi, T\rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{t}} \xrightarrow{\mathsf{t}} \langle \Pi', T'\rangle, \sigma', \kappa', \rho', \alpha_{\mathsf{t}}{}'$$

We explain Figure 16 using Figure 15 as a guide. Under an expression command e, the machine can take both evaluation (**E.0–6**) and undo (**U.1–4**) transitions while remain-

ing in evaluation mode, as well as transitions **E.P** and **E.7**, which each change to another command. Under the propagation command **prop**, the machine can take propagation transitions (**P.1**–**6**) while remaining in propagation mode, as well as transitions **P.E** and **P.7**, which each change to another command.

Propagation can transition into evaluation (**P.E**) when it's focused on an **update** action that it (non-deterministically) chooses to activate; it may also (non-deterministically) choose to ignore this opportunity and continue propagation. Dually, evaluation can transition directly into propagation (**E.P**) when its command is a **memo** point that matches a **memo** point currently focused in the reuse trace (and in particular, the environment ρ must also match); it may also (non-deterministically) choose to ignore this opportunity and continue evaluation. We describe a deterministic algorithms for change propagation and memoization in Section 6.1.

Evaluation (respectively, propagation) transitions into a value sequence $\overline{\nu}$ after evaluating (respectively, propagating) a **pop** operation under **E.7** (respectively, **P.7**). Under the value sequence command, the machine can continue to undo the reuse trace (**U.1**–**4**). To change commands, it rewinds its trace context and either resumes evaluation (**E.8**) upon finding the mark $\square$, or resumes propagation (**P.8**) upon finding the mark $\boxplus$. The machine rewinds the trace using the following *trace rewinding* relation:

$$\begin{array}{lcl}
\langle \Pi \cdot t, T \rangle; T' & \circlearrowleft & \langle \Pi, T \rangle; t \cdot T' \\
\langle \Pi \cdot \boxminus_{T_2}, \varepsilon \rangle; T' & \circlearrowleft & \langle \Pi, T_2 \rangle; T' \\
\langle \Pi \cdot \boxminus_{T_2}, t \cdot T_1 \rangle; T' & \circlearrowleft & \langle \Pi, (t \cdot T_1) \cdot T_2 \rangle; T'
\end{array}$$

This relation simultaneously performs two functions. First, it moves the focus backwards across actions (towards the start of the trace) while moving these actions into a new subtrace T'; the first case captures this behavior. Second, when moving past a leftover undo mark $\boxminus_{T_2}$, it moves the subtrace T_2 back into the reuse trace; the second and third cases capture this behavior. Note that unlike $\boxminus$, there is no way to rewind beyond either $\square$ or $\boxplus$ marks. This is intentional: rewinding is meant to stop when it encounters either of these marks.

4. Consistency

In this section we formalize a notion of consistency between the reference machine and tracing machine. As a first step, we show that when run from scratch (without a reuse trace), the results of the tracing machine are consistent with the reference machine, i.e., the final machine values and stores coincide. To extend this property beyond from-scratch runs, it is necessary to make an additional assumption: we require each IL program run in the tracing machine to be *compositionally store agnostic* (CSA, see below). We then show that, for CSA IL programs, the tracing machine reuses computations in a consistent way: its final trace, store, and machine values are consistent with a from-scratch run of the tracing machine, and hence, they are consistent with a run of the reference machine. The accompanying technical report contains complete proofs [22].

4.1 Compositional Store Agnosticism (CSA)

The property of compositional store agnosticism characterizes the programs for which our tracing machine runs consistently. We build this property from a less general property that we call *store agnosticism*. Intuitively, an IL program is store agnostic iff, whenever an **update** instruction is performed during its execution, then the value sequence that will eventually be popped is already determined at this point and, moreover, independent of the current store.

Definition 4.1. Formally, we define $\mathsf{SA}(\sigma, \rho, e)$ to mean: If $\sigma, \epsilon, \rho, e \xrightarrow{\mathsf{r}}{}^{*} \_, \_, \rho', \textbf{update}\ e'$, then there exists $\overline{\nu}$ such that $\overline{w} = \overline{\nu}$ whenever $\_, \epsilon, \rho', e' \xrightarrow{\mathsf{r}}{}^{*} \_, \epsilon, \epsilon, \overline{w}$.

To see why this property is significant, recall how the tracing machine deals with intermediate results. In stepping rule **E.8**, the tracing machine mirrors the reference machine: it passes the results to the function on the top of the control stack. However, in stepping rule **P.8**, the tracing machine does *not* mirror the reference machine: it essentially discards the intermediate results and continues to process the remaining reuse trace. This behavior is not generally consistent with the reference machine: If **P.8** is executed after switching to evaluation mode (**P.E**) and performing some computation in order to adjust to a modified store, then the corresponding intermediate result may be different. However, if the subprogram that generated the reuse trace was store agnostic, then this new result will be the same as the original one; consequently, it is then safe to continue processing the remaining reuse trace.

Compositional store agnosticism is a generalization of store agnosticism that is preserved by execution.

Definition 4.2. We define $\mathsf{CSA}(\sigma, \rho, e)$ to mean: If $\sigma, \varepsilon, \rho, e \xrightarrow{\mathsf{r}}{}^{*} \sigma', \kappa, \rho', e'$, then $\mathsf{SA}(\sigma', \rho', e')$.

Lemma 4.1. *If* $\sigma, \varepsilon, \rho, e \xrightarrow{\mathsf{r}}{}^{*} \sigma', \kappa', \rho', e'$ *and* $\mathsf{CSA}(\sigma, \rho, e)$, *then* $\mathsf{CSA}(\sigma', \rho', e')$.

4.2 Consistency of the Tracing Machine

The first correctness property says that, when run from scratch (i.e. without a reuse trace), the tracing machine mirrors the reference machine.

Theorem 4.2 (Consistency of from-scratch runs)**.**
If $\langle \varepsilon, \varepsilon \rangle, \sigma, \varepsilon, \rho, \alpha_{\mathsf{r}} \xrightarrow{t}{}^{*} \langle \_, \_ \rangle, \sigma', \varepsilon, \varepsilon, \overline{\nu}$
then $\sigma, \varepsilon, \rho, \alpha_{\mathsf{r}} \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{\nu}$.

In the general case, the tracing machine does not run from scratch, but with a reuse trace generated by a from-scratch run. To aid readability for such executions we introduce some notation. We call a machine reduction *balanced* if the initial and final stacks are each empty, and the initial and final trace contexts are related by the trace rewinding

relation. If know that the stack and trace context components of a machine reduction meet this criteria, we can specify this (balanced) reduction more concisely.

Definition 4.3 (Balanced reductions)**.**

$$\frac{\langle \varepsilon, T\rangle, \sigma, \epsilon, \rho, \alpha_{\mathsf{r}} \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi, \varepsilon\rangle, \sigma', \epsilon, \epsilon, \overline{\nu} \quad \langle \Pi, \varepsilon\rangle; \varepsilon \circlearrowleft^{*} \langle \varepsilon, \varepsilon\rangle; T'}{T, \sigma, \rho, \alpha_{\mathsf{r}} \Downarrow T', \sigma', \overline{\nu}}$$

$$\frac{\langle \varepsilon, T\rangle, \sigma, \epsilon, \epsilon, \textbf{prop} \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi, \varepsilon\rangle, \sigma', \epsilon, \epsilon, \overline{\nu} \quad \langle \Pi, \varepsilon\rangle; \varepsilon \circlearrowleft^{*} \langle \varepsilon, \varepsilon\rangle; T'}{T, \sigma \curvearrowright T', \sigma', \overline{\nu}}$$

We now state our second correctness result. It uses an auxiliary function that collects garbage: $\sigma|_{\mathsf{gc}}(\ell) = \sigma(\ell)$ for $\ell \in \mathsf{dom}(\sigma|_{\mathsf{gc}}) = \{\ell \mid \ell \in \mathsf{dom}(\sigma) \text{ and } \sigma(\ell) \neq \diamond\}$.

Theorem 4.3 (Consistency)**.**
Suppose $\varepsilon, \sigma_1, \rho_1, \alpha_{\mathsf{r}1} \Downarrow T_1, \sigma_1', \overline{\nu}_1$ *and* $\mathsf{CSA}(\sigma_1, \rho_1, \alpha_{\mathsf{r}1})$.

1. *If* $T_1, \sigma_2, \rho_2, \alpha_{\mathsf{r}2} \Downarrow T_1', \sigma_2', \overline{\nu}_2$
   then $\varepsilon, \sigma_2|_{\mathsf{gc}}, \rho_2, \alpha_{\mathsf{r}2} \Downarrow T_1', \sigma_2'|_{\mathsf{gc}}, \overline{\nu}_2$
2. *If* $T_1, \sigma_2 \curvearrowright T_1', \sigma_2', \overline{\nu}_2$
   then $\varepsilon, \sigma_2|_{\mathsf{gc}}, \rho_1, \alpha_{\mathsf{r}1} \Downarrow T_1', \sigma_2'|_{\mathsf{gc}}, \overline{\nu}_2$

The first statement says that, when run with an arbitrary from-scratch generated trace T_1, the tracing machine produces a final trace, store and return value sequence that are consistent with a from-scratch run of the same program. The second statement is analogous, except that it concerns change propagation: when run over an arbitrary from-scratch generated trace T_1, the machine produces a result consistent with a from-scratch run of the program that generated T_1. Note that in each case the initial store may be totally different from the one used to generate T_1.

Finally, observe how each part of Theorem 4.3 can be composed with Theorem 4.2 to obtain a corresponding run of the reference machine.

Collecting the garbage. The tracing machine may undo portions of the reuse trace in order to adjust it to a new store. Whenever it undoes an allocation (rule **U.1**), it marks the corresponding location as garbage ($\ell \mapsto \diamond$).

In order for this to make sense we better be sure that these locations are not live in the final result, i.e., they neither appear in T_1' nor $\overline{\nu}_2$ nor are referenced from the live portion of σ_2'. In fact, this is a consequence of the consistency theorem: the from-scratch run in the conclusion produces the same T_1' and $\overline{\nu}_2$. Moreover, since its final store is $\sigma_2'|_{\mathsf{gc}}$, it is clear that these components and $\sigma_2'|_{\mathsf{gc}}$ itself cannot refer to garbage.

5. Destination-Passing Style

In Section 4.1, we defined the CSA property that the tracing machine requires of all programs for consistency. In this section, we describe a *destination-passing-style* transformation and show that it transforms arbitrary IL programs into CSA IL programs, while preserving their semantics. The idea is as follows: A DPS-converted program takes an additional parameter x that acts as its destination. Rather than return its results directly, the program then instead writes them to the memory specified by x.

$$\begin{array}{lcl}
[\![\textbf{let fun } f(\overline{x}).e_1 \textbf{ in } e_2]\!]_y & = & \textbf{let fun } f(\overline{x} \mathbin{@} z).[\![e_1]\!]_z \textbf{ in } [\![e_2]\!]_y \\
[\![\textbf{let } x = \oplus(\overline{y}) \textbf{ in } e]\!]_y & = & \textbf{let } x = \oplus(\overline{y}) \textbf{ in } [\![e]\!]_y \\
[\![\textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2]\!]_y & = & \textbf{if } x \textbf{ then } [\![e_1]\!]_y \textbf{ else } [\![e_2]\!]_y \\
[\![f\ (\overline{x})]\!]_y & = & f\ (\overline{x} \mathbin{@} y) \\
[\![\textbf{let } x = \iota \textbf{ in } e]\!]_y & = & \textbf{let } x = \iota \textbf{ in } [\![e]\!]_y \\
[\![\textbf{memo } e]\!]_y & = & \textbf{memo } [\![e]\!]_y \\
[\![\textbf{update } e]\!]_y & = & \textbf{update } [\![e]\!]_y \\
[\![\textbf{push } f \textbf{ do } e]\!]_y & = & \textbf{let fun } f'(z).\ \textbf{update} \\
\quad \text{when } \mathsf{Arity}(f) = n & & \quad\ \textbf{let } x_1 = \textbf{read}(z[1]) \textbf{ in } \cdots \\
& & \quad\ \textbf{let } x_n = \textbf{read}(z[n]) \textbf{ in} \\
& & \quad\ f\ (x_1, \ldots, x_n, y) \\
& & \textbf{in} \\
& & \textbf{push } f' \textbf{ do memo} \\
& & \quad\ \textbf{let } z = \textbf{alloc}(n) \textbf{ in } [\![e]\!]_z \\
[\![\textbf{pop } \overline{x}]\!]_y & & \\
\quad \text{when } |\overline{x}| = n & = & \textbf{let } _ = \textbf{write}(y[1], x_1) \textbf{ in } \cdots \\
& & \textbf{let } _ = \textbf{write}(y[n], x_n) \textbf{ in} \\
& & \textbf{pop } \langle y \rangle \\
[\![\varepsilon]\!] & = & \varepsilon \\
[\![\rho[x \mapsto \nu]]\!] & = & [\![\rho]\!][x \mapsto \nu] \\
[\![\rho[f \mapsto \textbf{fun } f(\overline{x}).e]]\!] & = & [\![\rho]\!][f \mapsto \textbf{fun } f(\overline{x} \mathbin{@} y).[\![e]\!]_y]
\end{array}$$

Figure 17. Destination-passing-style (DPS) conversion.

Figure 17 defines the DPS transformation for an expression e and a destination variable x, written $[\![e]\!]_x$. Naturally, to DPS-convert an expression closed by an environment ρ, we must DPS-convert the environment as well, written $[\![\rho]\!]$. In order to comply with our assumption that all function and variable names are distinct, the conversion actually has to thread through a set of already-used names. For the sake of readability we do not include this here.

Most cases of the conversion are straightforward. The interesting ones include function definition, function application, **push**, and **pop**. For function definitions, the conversion extends the function arguments with an additional destination parameter z (we write $\overline{x} \mathbin{@} z$ to mean $\overline{x}$ appended with z). Correspondingly, for application of a function f, the conversion additionally passes the current destination to f. For **push**es, we allocate a fresh destination z for the **push** body; we memoize this allocation with a **memo** point. When the **push** body terminates, instead of directly passing control to f, the program calls a wrapper function f' that reads the destination and finally passes the values to the actual function f. Since these **read**s may become inconsistent in subsequent runs, we prepend them with an **update** point. For **pop**s, instead of directly returning its result, the converted program writes it to its destination and then returns the latter.

As desired, the transformation yields CSA programs (here and later on we assume that n is the arity of the program being transformed):

Theorem 5.1 (DPS programs are CSA)**.**
$\mathsf{CSA}(\sigma, [\![\rho]\!], \textbf{let } x \textbf{ = alloc } (n) \textbf{ in } [\![e]\!]_x)$

Moreover, the transformation preserves the extensional semantics of the original program:

Theorem 5.2 (DPS preserves extensional semantics)**.**
If $\sigma_1, \varepsilon, \rho, e \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{\nu}$
then $\sigma_1, \varepsilon, [\![\rho]\!], \textbf{let } x \textbf{ = alloc } (n) \textbf{ in } [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma_1' \uplus \sigma_2', \varepsilon, \varepsilon, \ell$
with $\sigma_2'(\ell, i) = \nu_i$ *for all* i.

Because it introduces destinations, the transformed program allocates additional store locations σ_2'. These locations are disjoint from the original store σ_1', whose contents are preserved in the transformed program. If we follow one step of indirection, from the returned location to the values it contains, we recover the original results $\overline{\nu}$.

5.1 An Example

As a simple illustrative example, consider the source-level expression f(`max(*`p`,*`q`)`), which applies function f to the maximum of two dereferenced pointers $*p$ and $*q$. Our front end translates this expression into the following:

```
push f do
  update
    let x = read(p[0]) in
    let y = read(q[0]) in
    if x > y then pop x else pop y
```

Notice that the body of this **push** is not store agnostic—when the memory contents of either pointer is changed, the **update** body can evaluate to a different return value, namely the new maximum of x and y. To address this, the DPS transformation converts this fragment into the following:

```
let fun f'(m). update
  let m' = read(m[0]) in f(m', z)
in
push f' do memo
  let m = alloc(1) in
  update
    let x = read(p[0]) in
    let y = read(q[0]) in
    if x > y then
      let _ = write(m[0],x) in pop m
    else
      let _ = write(m[0],y) in pop m
```

Notice that instead of returning the value of either x or y as before, the body of the **push** now returns the value of m, a pointer to the maximum of x and y. In this case, the push body is indeed store agnostic—though x and y may change, the pointer value of m remains fixed, since it is defined outside of the **update** body.

The astute reader may wonder why we place the allocation of m within the bodies of the **push** and **memo** point, rather than "lift it" outside the definition of function f'. After all, by lifting it, we would not need to return m to f' via the stack **pop**—the scope of variable m would include that of function f'. We place the allocation of m where we do to promote reuse of nondeterminism: by inserting this **memo** point, the DPS transformation effectively associates local input state (the values of p and q) with the local output state (the value of m). Without this **memo** point, every **push** body will generate a fresh destination each time it is reevaluated, and in general, this nondeterministic choice will prevent reuse of any subcomputation, since this subcomputation's local state includes a distinct, previously chosen destination. To avoid this behavior and to allow these subcomputations to instead be reused during change propagation, the DPS conversion inserts **memo** points that enclose each (non-deterministic) allocation of a destination.

6. Compiling IL

While the semantics of IL are given by an abstract machine (Section 3), in actuality we want to run IL programs with a more conventional machine—e.g., a machine that does not support tracing or change propagation directly. As such, our compilation process can be thought of as building a specialized tracing machine for a given IL program. At a high level, realizing this machine requires realizing each of its components, i.e., realizing its store, stack, environment, trace and stepping rules.

6.1 Runtime data structures and algorithms

The primary role of the runtime system is to provide realized versions of the abstract machine's trace and store, an efficient search for matching **memo** points, and an efficient algorithm for change propagation. To give an efficient change propagation algorithm, it is crucial that the runtime trace and store be "entangled", i.e., mutually referential: the runtime store references certain runtime trace actions, and the runtime representation of **read** and **write** trace actions each reference the runtime store.

The runtime trace. At a high level, the trace provides an ordering to trace actions. For efficiency, we use a (total) order maintenance data structure [16] which bestows each trace action t an associated time stamp $s(t)$; these timestamps admit an efficient predicate for checking if $t_1 \leq t_2$ by checking if $s(t_1) \leq s(t_2)$. Concretely, a *trace node* is a record consisting of a time stamp s and (at least one) trace action t. As a refinement to this approach, below we also consider when and how several trace actions can share a single trace node. Most of the trace actions are straightforward to represent during runtime, though extra care is needed for **read** and **write** actions, which we describe below.

The runtime store. While the store of the abstract machine only retains the current value of each location-offset entry $\ell[i]$ (hereafter, just an *entry*), this generally requires traversing and replaying the entire trace during change prop-

agation, which is prohibitively expensive. As such, the runtime store takes a different tack: for each entry, it persistently maintains all the corresponding read and written values, across the entire trace, including the corresponding trace action. Given a particular point in the trace, we quickly access the current value of any entry based on when it was last read or written, in terms of the time stamps described above. For this purpose, the runtime uses a self-balancing search tree for each changeable store entry; each node of the tree corresponds to a **read** or **write** trace action.

The runtime memo table. In the abstract machine, memoization permits trace reuse by matching **memo** points in the reuse trace. In the runtime, a hash table indexes each such **memo** point in the trace. While evaluating a new **memo** point, the runtime system attempts to locate matches using this hash table. Once matched, the change propagation algorithm begins working on the reused trace.

Change propagation as an algorithm. In the abstract machine, change propagation has two purposes: to replay store effects (viz. **alloc**s, **write**s) and to ensure that every reused **read** action is consistent with the current store. However, the machine specifies change propagation as a complete traversal of the trace, while in practice this is not efficient. As a result, the algorithm performs change propagation somewhat differently, while still accomplishing its two high-level goals: replaying store effects and ensuring that **read**s are consistent.

First, to replay store effects, the algorithm relies on the runtime store being *retained* from one run to the next. That is, the final store of one run becomes the initial store of the next run. This retention is modulo changes in some store entries, and the reclamation of locations marked as garbage. Consequently, since the runtime store keeps every traced effect—not just the most recent ones, as in the abstract machine—it is not necessary to replay these effects for the benefit of updating the store.

Second, to find and reevaluate inconsistent **read** actions, the runtime maintains a priority queue Q of them, ordered by their appearance in the trace. Rather than find them one-by-one via trace traversal, when the runtime store is changed, it uses the runtime store representation described above to identify any inconsistent **read**s and enqueue the smallest enclosing **update** point into Q. To find this **update** point quickly, each **read** action maintains a reference to this (unique) enclosing **update** action. The change propagation algorithm consists of a loop that reevaluates the **update** points in Q, in trace order.

6.2 Compilation

We compile IL programs in several phases. First, we convert them into destination-passing style; this ensures that they will replay correctly during change propagation. Next, we implement each traced expression with a corresponding call into the runtime, described above. For most traced forms, this is very straightforward; however, handling the **update** and **memo** forms requires more care, which we discuss below. Finally, we translate the resulting IL program into our target language, C, and compile this code with `gcc`.

***Compiling* update *and* memo.** In contrast to the other traced forms, **memo** and **update** each save and restore the local state of an IL program—an environment ρ and an IL expression e. To compile these forms, the following questions arise: How much of the environment ρ should be recorded in the runtime trace and/or memo table? Once an IL expression e is translated into a target language, how do we reevaluate it during change propagation?

First, we address how we save the environment. At each of these points we use a standard analysis (e.g., [30]) to approximate the live variables $\mathsf{LV}(e)$ at each such e, and then save not ρ, but rather $\rho|_{\mathsf{LV}(e)}$, i.e., ρ limited to $\mathsf{LV}(e)$. This has two important consequences: we save space by not storing dead variables in the trace, and we (monotonically) increase the potential for **memo** matches, as non-matching values of dead variables do not cause a potential match to fail.

Second, we address the issue of fine-grained reevaluation. This poses a problem since languages such as C do not allow programs to jump to arbitrary control points, e.g., into the middle of a procedure. To address this limitation, we adapt the "lambda-lifting" technique used in earlier work [23]. Originally this technique transformed the control flow graphs of C code; we modify it for IL such that after being applied, all **update** points have the form **update** $f(\overline{x})$ where f is a top-level function and where variables $x_i \in \overline{x}$ close the body of f. In this form, we implement each **update** point as an explicitly-constructed function closure, i.e., a record consisting of a function pointer and values for its arguments.

6.3 Optimizations

We refine the basic approach above with two optimizations.

Trace node sharing (*share*). The basic runtime system (Section 6.1) assigns each trace action t to a distinct trace node, with a distinct time stamp s. Since each trace node brings some overhead, it is desirable if sequences of consecutive trace actions $\overline{t} = t_1, \ldots t_n$ can share a single trace node with a single time stamp. However, this optimization is complicated by a few issues.

First, how do we realize the comparison $t_i < t_j$ when t_i and t_j share a single time stamp? We can accomplish this by following the order of $\overline{t}$ when placing the actions into the trace node; this allows us to efficiently compare t_i with t_j by comparing their addresses.

Second, how do we avoid breaking the sequence when it uses a single trace node? This can happen in one of two ways: by either **memo**-matching some action in the middle of $\overline{t}$, thereby discarding its prefix; or by reevaluating an **update** point in the middle of $\overline{t}$ when this reevaluation takes a new control path. We avoid these scenarios by packing

sequence $\bar{t}$ into a single trace node only when the following criteria are met: if $\bar{t}$ contains a **memo** point, then it appears first; if $\bar{t}$ contains an **update** point, then the remaining suffix of $\bar{t}$ is generated by straight-line code.

Selective destination-passing style (*seldps*). The DPS conversion (Figure 17) introduces extra IL code for **push** and **pop** expressions: an extra **alloc**, **update**, **memo**, and some **write**s and **read**s. Since each of these expressions are traced, this can introduce considerable overhead for subcomputations that do not interact with changing data. In fact, without an **update** point, propagation over the trace of e will always yield the same return values. Moreover, it is clear from the definition of store agnosticism (Section 4.1) that any computation without an **update** point is trivially CSA, hence, there is no need to DPS-convert it. By doing a conservative static analysis, our compiler estimates whether each expression e appearing in the form **push** f **do** e can reach an **update** point during evaluation. If not, we do not apply the DPS conversion to **push** f **do** e. We refer to this refined transformation as *selective* DPS conversion.

7. Implementation and a C Front End

Our current implementation consists of a compiler and an associated runtime system, as outlined in Section 6. Additionally, we also implement the optimizations from Section 6.3. After compiling and optimizing IL, our implementation translates it to C, which we compile using `gcc`. In all, our compiler consists of a 10k line extension to CIL [31] and our runtime system consists of about 6k lines of C code.

As a front-end to IL, we support a C-like source language, $\mathsf{C}_{\mathsf{src}}$. We use CIL to parse $\mathsf{C}_{\mathsf{src}}$ source into a control-flow graph representation. To bridge the gap between this representation and IL, we utilize a known relationship between static single assignment (SSA) form and lexically-scoped, functional programming [10].

Before this translation, we move $\mathsf{C}_{\mathsf{src}}$ variables to the heap if either they are globally-scoped, aliased by a pointer (via $\mathsf{C}_{\mathsf{src}}$'s address-of operator, `&`), or are larger than a single machine word. When such variables come into scope, we allocate space for them in the heap (via **alloc**); for global variables, this allocation only happens once, at the start of execution.

As apart of the translation to IL, we automatically place **update** points before each **read** (or consecutive sequence of **read**s). Though in principle we can automatically place **memo** points anywhere, we currently leave their placement to the programmer by providing a **memo** keyword in $\mathsf{C}_{\mathsf{src}}$; this keyword can be used as a $\mathsf{C}_{\mathsf{src}}$ statement, as well as a wrapper around arbitrary $\mathsf{C}_{\mathsf{src}}$ expressions.

7.1 Current Limitations

Our source language $\mathsf{C}_{\mathsf{src}}$ is more restricted than C in a few ways, though most of these restrictions are merely for technical reasons and could be solved with further compiler engineering. First, while $\mathsf{C}_{\mathsf{src}}$ programs may use variadic functions provided by external libraries (e.g., `printf`), $\mathsf{C}_{\mathsf{src}}$ does not currently support the definition of new variadic functions. Furthermore, function argument and return types must be scalar (pointer or base types) and not composite types (`struct` and `union` types). Removing these restrictions may pose compiler engineering challenges, but should not require a fundamental change to our approach.

Second, our $\mathsf{C}_{\mathsf{src}}$ front-end assumes that the program's memory accesses are word aligned. This assumption simplifies the translation of pointer dereferencing and assignment in $\mathsf{C}_{\mathsf{src}}$ into the **read** and **write** instructions in IL, respectively. To lift this restriction, we could dynamically check the alignment of each pointer before doing the access, and decompose those accesses that are not word-aligned into one (or two) that are.

Third, as a more fundamental challenge, $\mathsf{C}_{\mathsf{src}}$ does not currently support features of C that change the stack discipline of the language, such as `setjmp`/`longjmp`. In C, these functions are often used to mimic the control operators and/or exception handling found in higher-level languages. Supporting these features is beyond the scope of this paper, but remains of interest for future work.

7.2 Mixing $\mathsf{C}_{\mathsf{src}}$ with Foreign C Code

To improve efficiency, programs written in $\mathsf{C}_{\mathsf{src}}$ can be mixed with foreign C code (e.g., from a standard C library). Since foreign C code is not traced, it allows those parts of the program to run faster, as they do not incur the tracing overhead that would otherwise be incurred within $\mathsf{C}_{\mathsf{src}}$. However, mixing $\mathsf{C}_{\mathsf{src}}$ and foreign C code contains potential pitfalls.

First, programs in this setting must (at least) adhere to the following correct usage restriction: each word in memory is either accessed exclusively by foreign C code (not by $\mathsf{C}_{\mathsf{src}}$ code) or exclusively by $\mathsf{C}_{\mathsf{src}}$ code (not by foreign C code). This restriction ensures that the foreign C code will not corrupt the memory of the $\mathsf{C}_{\mathsf{src}}$ program and vice versa, while still allowing memory blocks to be used in a mixed way (each block may contain several words, each of which may be accessed by either $\mathsf{C}_{\mathsf{src}}$ code or foreign C code). While a skilled programmer can observe this restriction (we mix foreign C code with $\mathsf{C}_{\mathsf{src}}$ code for some of our benchmarks), we currently do not enforce it with static or dynamic checks.

Second, beyond avoiding memory corruption, programs in this setting must also ensure the consistency of change propagation, which generally does not hold when $\mathsf{C}_{\mathsf{src}}$ is mixed with arbitrary foreign C code: since the foreign C code is not traced, it may behave strangely during reevaluation, for instance, if it uses any internal mutable state. One way to avoid such strange behavior is to avoid foreign C functions which keep internal state that persists between separate calls. On the other hand, sometimes such state is benign, as in the case of `printf`.

8. Evaluation

We empirically evaluate our approach by considering a number of benchmarks written in $\mathsf{C}_{\mathsf{src}}$ (Section 7), compiled with our compiler (Section 6). Our experiments are very encouraging, showing that our approach can yield asymptotic speedups, resulting in orders of magnitude speedups in practice; it does this while incurring only moderate overheads for pre-processing or initial executions. We evaluate our compiler and runtime optimizations (Section 6.3), showing that they improve performance of both from-scratch evaluation as well as of change propagation. Comparisons with previous work using the unsound CEAL library and the DeltaML language shows that our approach performs competitively.

8.1 Benchmarks and Measurements

Our benchmarks consist of expression tree evaluation (i.e., the example from Section 2), some list primitives, two sorting algorithms and several computational geometry algorithms. For our timings, we used a Linux box running on a 1.8 GHz Intel Xeon (4-core) processor with 512GB memory. All our benchmarks are sequential and are compiled with `gcc -O3` after translation to C.

For each benchmark, we measure the *from-scratch time*, the time to run the benchmark from-scratch on a particular input, and the average *update time*, the average time required by change propagation to update the output after inserting or deleting an element from its input. We compute this average by iterating over the initial input, deleting each input element, updating the output by change propagation, inserting the element again and updating the output by change propagation.

List primitives. These benchmarks include filter, map (performs integer additions per element), reverse, minimum (integer comparison), and sum (integer addition), and the sorting algorithms quicksort (string comparison) and mergesort (string comparison). We generate lists of n (uniformly) random integers as input for the list primitives. For sorting algorithms, we generate lists of n (uniformly) random, 32-character strings. We implement each list benchmark mentioned above by using an external C library for lists, which our compiler links against the self-adjusting code after compilation.

Computational geometry. These benchmarks include quickhull, diameter, and distance; quickhull computes the convex hull of a point set using the standard quickhull algorithm; diameter computes the diameter, i.e., the maximum distance between any two points of a point set; distance computes the minimum distance between two sets of points. Our implementations of diameter and distance use quickhull to compute first the convex hull and then compute the diameter and the distance of the points on the hull (the furthest away points lie on the convex hull). For quickhull and distance, input points are selected from a uniform distribution over the unit square in $\mathbb{R}^2$. For distance, we select equal numbers of points from two non-overlapping unit squares in $\mathbb{R}^2$. We represent real numbers with double-precision floating-point numbers. As with the list benchmarks, each computational geometry benchmark uses an external C library; in this case, the external library provides geometric primitives for creating points and lines, and computing simple properties about them (e.g., line-point distance).

Benchmark targets. In order to study the effectiveness of the compiler and runtime optimizations (Section 6.3), for each benchmark we generate several *targets*. Each target is the result of choosing to use some subset of our optimizations. Table 1 lists and describes each target that we consider. Before measuring the performance of these targets, we use regression tests to verify that their self-adjusting semantics are consistent with conventional (non-self-adjusting) versions. These tests empirically verify our consistency theorem (Theorem 4.3).

Target	Optimizations used
no-opt	Neither share nor seldps is used.
share	Same as no-opt except that certain trace actions can share trace nodes.
seldps	Same as no-opt except that the DPS transformation is selective—only certain functions are transformed.
opt	Both seldps and share are used.

Table 1. Targets and their optimizations (Section 6.3).

8.2 Optimizations

Figure 18 compares our targets' from-scratch running time and average update time. Each bar is normalized to the no-opt target. The rightmost column in each bar graph shows the mean over all benchmarks. To estimate the efficacy of an optimization X, we can compare target no-opt with the target where X is turned on.

In the mean, the fully optimized targets (opt) are nearly 30% faster from-scratch, and nearly 50% faster during automatic updates (via change propagation), when compared to the unoptimized versions (no-opt). These results demonstrate that our optimizations, while conceptually straightforward, are also practically effective: they significantly improve the performance of the self-adjusting targets, especially during change propagation.

8.3 Summary of Experimental Results

Table 8.1 summarizes the self-adjusting performance of the benchmarks by comparing them to conventional, non-self-adjusting C code. From left to right, the columns show the benchmark name, the input size we considered (N), the time to run the conventional (non-self-adjusting) version (Conv), the from-scratch time of the self-adjusting version (FS), the *preprocessing overhead* associated with the self-adjusting

Benchmark	N	Conv (sec)	FS (sec)	Overhead (FS / Conv)	Ave. Update (sec)	Speed-up (Conv / AU)
exptrees	10^6	0.18	1.53	8.5	1.3×10^{-5}	1.4×10^4
map	10^6	0.10	1.87	18.4	3.4×10^{-6}	3.0×10^4
reverse	10^6	0.10	1.81	18.4	2.6×10^{-6}	3.8×10^4
filter	10^6	0.13	1.42	10.7	2.7×10^{-6}	4.9×10^4
sum	10^6	0.14	1.35	9.6	9.3×10^{-5}	1.5×10^3
minimum	10^6	0.18	1.36	7.7	1.3×10^{-5}	1.4×10^4
quicksort	10^5	0.40	3.30	8.2	5.8×10^{-4}	6.9×10^2
mergesort	10^5	0.74	5.31	7.2	9.5×10^{-4}	7.8×10^2
quickhull	10^5	0.26	0.97	3.7	1.2×10^{-4}	2.2×10^3
diameter	10^5	0.26	0.90	3.4	1.5×10^{-4}	1.8×10^3
distance	10^5	0.24	0.81	3.4	3.0×10^{-4}	7.9×10^2

Table 2. Summary of benchmark results (using opt target of each benchmark).

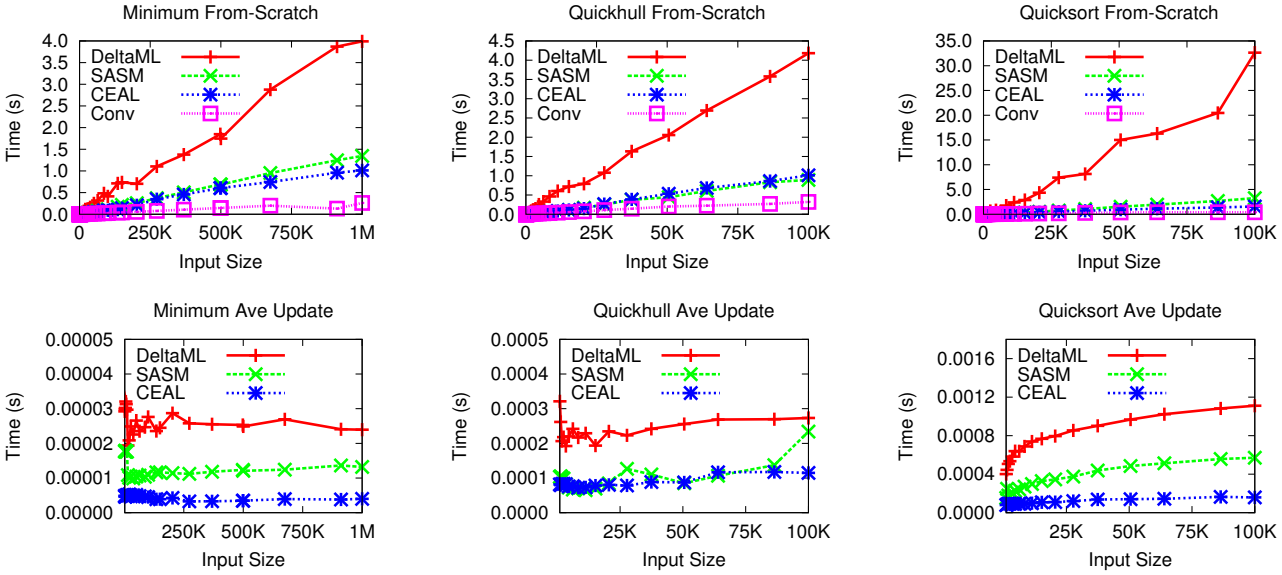

Figure 19. minimum, quickhull and quicksort performance in DeltaML, CEAL and our own opt versions (labeled SASM).

version (Overhead is the ratio FS/Conv), the average update time for the self-adjusting version (Ave. Update) and the *speed-up* gained by using change propagation to update the output versus rerunning the conventional version (Speed-up is the ratio Conv/Ave. Update). All reported times are in seconds. For the self-adjusting versions, we use the optimized (opt) target of each benchmark.

The preprocessing overheads of most benchmarks are less than a factor of ten; for simpler list primitives benchmarks, this overhead is about 18 or less. However, even at these only moderate input sizes (viz. 10^5 and 10^6), the self-adjusting versions deliver speed-ups of two, three or four orders of magnitude. Moreover, as we illustrate below (Section 8.4), these speedups increase with input size.

8.4 Comparison to Past Work

To illustrate how our implementation compares with past systems, Figure 19 gives representative examples. It compares the from-scratch and average update times for three self-adjusting benchmarks across three different implementations: one in DeltaML [27], one in CEAL [23] and the opt target of our implementation (labeled SASM, for *S*elf-*A*djusting *S*tack *M*achines). In the from-scratch graphs, we also compare with the conventional (non-self-adjusting) C implementations of each benchmark (labeled Conv).

The three benchmarks shown (viz. minimum, quickhull and quicksort) illustrate a general trend. First, in from-scratch runs, the SASM implementations are only slightly slower than that of CEAL, while the DeltaML implementations are considerably slower than both. For instance, in the

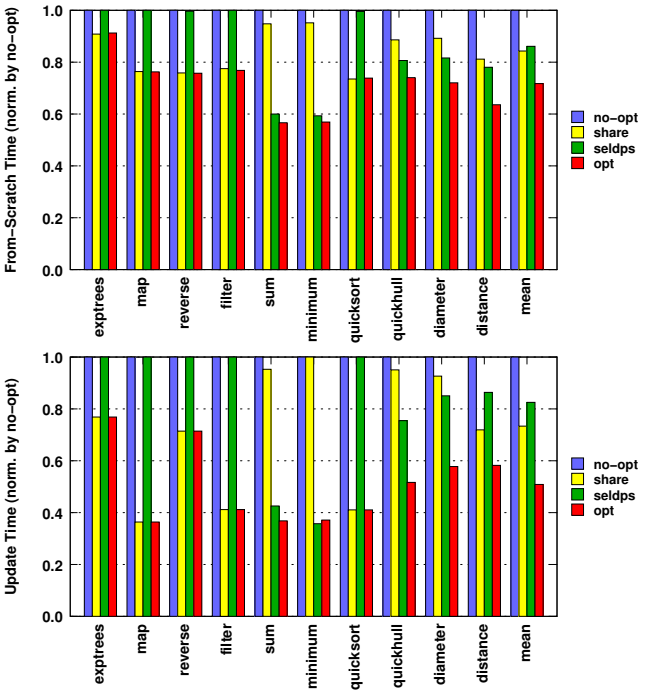

Figure 18. Comparison of benchmark targets.

case of quicksort, the DeltaML implementation is a factor of ten slower than our own. While updating the computation via change propagation, the performance of the SASM implementations lies somewhere between that of DeltaML and CEAL, with CEAL consistently being either faster than the others, or comparable to SASM. Although not reported here, we obtain similar results with other benchmarks.

9. Related Work

We discuss most closely related work in the previous sections of the paper, especially Section 1. Here, we briefly characterize earlier work on incremental computation and more recent work on generalizing self-adjusting-computation techniques to support parallel computation.

Of the many techniques proposed to support incremental computation (see the survey [33]), the most effective ones are dependence graphs, memoization, and partial evaluation. Dependence graphs record the dependencies between data in a computation and rely on a change-propagation algorithm to update the computation when the input is modified (e.g., [15, 25]). Dependence graphs are effective in some applications, e.g. syntax-directed computations, but are not general-purpose because change propagation does not update the dependencies. For example, the INC language [37], which uses dependence graphs, does not permit recursion. Memoization (also called function caching) (e.g., [1, 24, 32]) applies to any purely functional program and therefore is more broadly applicable than dependence graphs. This classic idea dating back to the late 1950's [11, 28, 29] can improve efficiency when executions of a program with similar inputs perform similar function calls. It turns out, however, that even a small input modification can prevent reuse via memoization, e.g., when they affect computations deep in the call tree [6]. Partial evaluation approaches [18, 36] require the user to fix a part of the input and specialize the program to speedup modifications to the remaining unfixed part. The main limitation of this approach is that it allows input modifications only within a predetermined partition.

In addition to the early systems discussed above, a more recent system, DITTO [34], offers support for incremental invariants-checking in Java. It requires no programmer annotations but only supports a purely-functional subset of Java. DITTO also places further restrictions on the programs; while these restrictions are reasonable for expressing invariant checks, they also narrow the scope of the approach.

More recent work generalized self-adjusting computation techniques to support parallel computations. Hammer et al [21] present an algorithm for parallel change propagation and propose some implementation techniques. More recent papers apply parallel self-adjusting computation to individual problems [8, 35] and the map-reduce framework [12] where large-scale computations may be performed in a parallel and distributed setting.

10. Conclusion

We described a sound abstract machine semantics for self-adjusting computation based on a low-level intermediate language. We implemented this language by presenting compilation and optimization techniques, including a C-like front end. Our experiments confirm that the self-adjusting programs produced with our approach often perform asymptotically faster than full reevaluation, resulting in orders of magnitude speedups in practice. We also confirmed that our approach is competitive with past approaches, which are either unsound or unsuited to low-level settings.

Acknowledgments We thank Joshua Dunfield and the anonymous reviewers for their feedback, which greatly improved our presentation.

References

[1] M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

[2] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. D for data: Differentiating data structures. *Fundam. Inf.*, 65(1-2): 1–28, 2004.

[3] U. A. Acar, A. Ihler, R. Mettu, and O. Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007.

[4] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, 2008.

[5] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the*

16th Annual European Symposium on Algorithms, September 2008.

[6] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–3:53, 2009.

[7] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*, 2010.

[8] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Parallelism in dynamic well-spaced point sets. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2011.

[9] P. K. Agarwal, L. J. Guibas, H. Edelsbrunner, J. Erickson, M. Isard, S. Har-Peled, J. Hershberger, C. Jensen, L. Kavraki, P. Koehl, M. Lin, D. Manocha, D. Metaxas, B. Mirtich, D. Mount, S. Muthukrishnan, D. Pai, E. Sacks, J. Snoeyink, S. Suri, and O. Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

[10] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[11] R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

[12] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symposium on Cloud Computing*, 2011.

[13] M. Carlsson. Monads for incremental computing. In *International Conference on Functional Programming*, pages 26–35, 2002.

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

[15] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Principles of Programming Languages*, pages 105–116, 1981.

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

[17] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

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

[19] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 237–247, 1993.

[20] M. Hammer and U. A. Acar. Memory management for self-adjusting computation. In *International Symposium on Memory Management*, pages 51–60, 2008.

[21] M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

[22] M. Hammer, G. Neis, Y. Chen, and U. A. Acar. Self-adjusting stack machines. 2011. `http://arxiv.org/abs/1108.3265`.

[23] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[24] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 311–320, 2000.

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

[26] G. Huet. The zipper. *Journal of Functional Programming*, 7 (5):549–554, 1997.

[27] R. Ley-Wild, M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming*, 2008.

[28] J. 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.

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

[30] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[31] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.

[32] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages*, pages 315–328, 1989.

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

[34] A. Shankar and R. Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Programming Language Design and Implementation*, 2007.

[35] O. Sumer, U. A. Acar, A. Ihler, and R. Mettu. Fast parallel and adaptive updates for dual-decomposition solvers. In *Conference on Artificial Intelligence (AAAI)*, 2011.

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

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