CEAL: A C-Based Language for Self-Adjusting Computation

Matthew A. Hammer Umut A. Acar* Yan Chen

Toyota Technological Institute at Chicago
{hammer,umut,chenyan}@tti-c.org

Abstract

Self-adjusting computation offers a language-centric approach to writing programs that can automatically respond to modifications to their data (e.g., inputs). Except for several domain-specific implementations, however, all previous implementations of self-adjusting computation assume mostly functional, higher-order languages such as Standard ML. Prior to this work, it was not known if self-adjusting computation can be made to work with low-level, imperative languages such as C without placing undue burden on the programmer.

We describe the design and implementation of CEAL: a C-based language for self-adjusting computation. The language is fully general and extends C with a small number of primitives to enable writing self-adjusting programs in a style similar to conventional C programs. We present efficient compilation techniques for translating CEAL programs into C that can be compiled with existing C compilers using primitives supplied by a run-time library for self-adjusting computation. We implement the proposed compiler and evaluate its effectiveness. Our experiments show that CEAL is effective in practice: compiled self-adjusting programs respond to small modifications to their data by orders of magnitude faster than recomputing from scratch while slowing down a from-scratch run by a moderate constant factor. Compared to previous work, we measure significant space and time improvements.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Performance, Algorithms.

Keywords Self-adjusting computation, compilation, control and data flow, dominators, tail calls, trampolines, performance.

1. Introduction

Researchers have long observed that in many applications, application data evolves slowly or incrementally over time, often requiring only small modifications to the output. This creates the potential for applications to adapt to changing data significantly faster than re-computing from scratch. To realize this potential, researchers in the algorithms community develop so called *dynamic* or *kinetic* algorithms or data structures that take advantage of the particular properties of the considered problem to update computations quickly. Such algorithms have been studied extensively over a range of hundreds of papers (e.g. [13, 17] for surveys). These advances show that computations can often respond to small modifications to their data nearly a linear factor faster than recomputing from scratch, in practice delivering speedups of orders of magnitude. As a frame of comparison, note that asymptotic improvements in performance far surpasses the goal of parallelism, where speedups are bound by the number of available processors. Designing, analyzing, and implementing dynamic/kinetic algorithms, however, can be complex even for problems that are relatively simple in the conventional setting, e.g., the problem of incremental planar convex hulls, whose conventional version is straightforward, has been studied over two decades (e.g., [32, 11]). Due to their complexity, implementing these algorithms is an error-prone task that is further complicated by their lack of composability.

Self-adjusting computation (e.g., [4, 3]) offers a language-centric approach to realizing the potential speedups offered by incremental modifications. The approach aims to make writing self-adjusting programs, which can automatically respond to modifications to their data, nearly as simple as writing conventional programs that operate on unchanging data, while delivering efficient performance by providing an automatic update mechanism. In self-adjusting computation, programs are stratified into two components: a meta-level *mutator* and a *core*. The mutator interacts with the user or the outside world and interprets and reflects the modifications in the data to the core. The core, written like a conventional program, takes some input and produces an output. The core is self-adjusting: it can respond to modifications to its data by employing a general-purpose, built-in change propagation mechanism. The mutator can execute the core with some input from scratch, which we call a *from-scratch* or an *initial run*, modify the data of the core, including the inputs and other computation data, and update the core by invoking change propagation. A typical mutator starts by performing a from-scratch run of the program (hence the name initial run), and then repeatedly modifies the data and updates the core via change propagation.

At a high level, *change propagation* updates the computation by re-executing the parts that are affected by the modifications, while leaving the unaffected parts intact. Change propagation is guaranteed to update the computation correctly: the output obtained via change propagation is the same as the output of a from-scratch execution with the modified data. Even in the worst case, change propagation falls back to a from-scratch execution—asymptotically, it is never slower (in an amortized sense)—but it is often significantly faster than re-computing from-scratch.

Previous research developed language techniques for self-adjusting computation and applied it to a number of application domains (e.g., for a brief overview [3]). The applications show that from-scratch executions of self-adjusting programs incur a

* Acar is partially supported by a gift from Intel.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

moderate overhead compared to conventional programs but can respond to small modifications orders-of-magnitude faster than recomputing from scratch. The experimental evaluations show that in some cases self-adjusting programs can be nearly as efficient as the "hand-designed" and optimized dynamic/kinetic algorithms (e.g., [6]). Recent results also show that the approach can help develop efficient solutions to challenging problems such as some three-dimensional motion simulation problems that have resisted algorithmic approaches [5].

Existing general-purpose implementations of self-adjusting computation, however, are all in high-level, mostly functional languages such as Standard ML (SML) or Haskell [27, 12]. Several exist in lower-level languages such as C [6] and Java [35] but they are domain-specific. In Shankar and Bodik's implementation [35], which targets invariant-checking applications, core programs must be purely functional and functions cannot return arbitrary values or use values returned by other functions in an unrestricted way. Acar et al's C implementation [6] targets a domain of tree applications. Neither approach offers a general-purpose programming model. The most general implementation is Hammer et al's C library [21], whose primary purpose is to support efficient memory management for self-adjusting computation. The C library requires core programs to be written in a style that makes dependencies between program data and functions explicit, limiting its effectiveness as a source-level language.

That there is no general-purpose support for self-adjusting computation in low level, imperative languages such as C is not accidental: self-adjusting computation critically relies on higher-order features of high-level languages. To perform updates efficiently, change propagation must be able to re-execute a previously-executed piece of code in the same state (modulo the modifications), and skip over parts of the computation that are unaffected by the modifications. Self-adjusting computation achieves this by representing the dependencies between the data and the program code as a trace that records specific components of the program code and their run-time environments. Since higher-order languages can natively represent closed functions, or *closures*, consisting of a function and its free variables, they are naturally suitable for implementing traces. Given a modification, change propagation finds the closures in the trace that depend on the modified data, and re-executes them to update the computation and the output. Change propagation utilizes recorded control dependencies between closures to identify the parts of the computation that need to be purged and uses memoization to recover the parts that remain the same. To ensure efficient change propagation, the trace is represented in the form of a dynamic dependence graph that supports fast random access to the parts of the computation to be updated.

In this paper, we describe the design, implementation, and evaluation of CEAL: a C-based language for self-adjusting computation. The language extends C with several primitives for self-adjusting computation (Section 2). Reflecting the structure of self-adjusting programs, CEAL consists of a meta language for writing mutators and a core language for writing core programs. The key linguistic notion in both the meta and the core languages is that of the *modifiable reference* or *modifiable* for short. A modifiable is a location in memory whose contents may be read and updated. CEAL offers primitives to create, read (access), and write (update) modifiables just like conventional pointers. The crucial difference is that CEAL programs can respond to modifications to modifiables automatically. Intuitively, modifiables mark the computation data that can change over time, making it possible to track dependencies selectively. At a high level, CEAL can be viewed as a dialect of C that replaces conventional pointers with modifiables.

By designing the CEAL language to be close to C, we make it possible to use familiar C syntax to write self-adjusting programs. This poses a compilation challenge: compiling CEAL programs to self-adjusting programs requires identifying the dependence information needed for change propagation. To address this challenge, we describe a two-phase compilation technique (Sections 5 and 6). The first phase *normalizes* the CEAL program to make the dependencies between data and parts of the program code explicit. The second phase translates the normalized CEAL code to C by using primitives supplied by a *run-time-system* (*RTS*) in place of CEAL's primitives. This requires creating closures for representing dependencies and efficiently supporting tail calls. We prove that the size of the compiled C code is no more than a multiplicative factor larger than the source CEAL program, where the multiplicative factor is determined by the maximum number of live variables over all program points. The time for compilation is bounded by the size of the compiled C code and the time for live variable analysis. Section 3.2 gives an overview of the compilation phases via an example.

We implement the proposed compilation technique and evaluate its effectiveness. Our compiler, `cealc`, provides an implementation of the two-level compilation strategy and relies on the RTS for supplying the self-adjusting-computation primitives. Our implementation of the RTS employs the recently-proposed memory management techniques [21], and uses asymptotically optimal algorithms and data structures to support traces and change propagation. For practical efficiency, the compiler uses intra-procedural compilation techniques that make it possible to use simpler, practically efficient algorithms. Our compiler `cealc` is between a factor of 3–8 slower and generates binaries that are 2–5 times larger than `gcc`.

We perform an experimental evaluation by considering a range of benchmarks, including several primitives on lists (e.g., map, filter), several sorting algorithms, and computational geometry algorithms for computing convex hulls, the distance between convex objects, and the diameter of a point set. As a more complex benchmark, we implement a self-adjusting version of the Miller-Reif tree-contraction algorithm, which is a general-purpose technique for computing various properties of trees (e.g., [28, 29]). Our timing measurements show that CEAL programs are about 6–19 times slower than the corresponding conventional C program when executed from scratch, but can respond to small changes to their data orders-of-magnitude faster than recomputing from scratch. Compared to the state-of-the-art implementation of self-adjusting computation in SML [27], CEAL uses about 3–5 times less memory. In terms of run time, CEAL performs significantly faster than the SML-based implementation. In particular, when the SML benchmarks are given significantly more memory than they need, we measure that they are about a factor of 9 slower. Moreover, this slowdown increases (without bound) as memory becomes more limited. We also compared our implementation to a hand-optimized implementation of self-adjusting computation for tree contraction [6]. Our experiments show that we are about 3–4 times slower.

In this paper, we present a C-based general-purpose language for self-adjusting computation. Our contributions include the language, the compiler, and the experimental evaluation. An extended version of this paper, including the proofs and more detailed experiments, can be found in the accompanying technical report [22].

2. The CEAL language

We present an overview of the CEAL language, whose core is formalized in Section 4. The key notion in CEAL is that of a *modifiable reference* (or *modifiable*, for short). A modifiable is a location in memory whose content may be read and updated. From an operational perspective, a modifiable is just like an ordinary pointer in memory. The major difference is that CEAL programs are sensitive to modifications of the contents of modifiables performed by the mutator, i.e., if the contents are modified, then the computation can respond to that change by updating its output automatically via change propagation.

Reflecting the two-level (core and meta) structure of the model, the CEAL language consists of two sub-languages: meta and core. The *meta language* offers primitives for performing an initial run, modifying computation data, and performing change propagation—the mutator is written in the meta language. CEAL's *core language* offers primitives for writing core programs. A CEAL program consists of a set of functions, divided into core and meta functions: the core functions (written in the core language), are marked with the keyword `ceal`, meta functions (written in the meta language) use conventional C syntax. We refer to the part of a program consisting of the core (meta) functions simply as the core (meta or mutator).

To provide scalable efficiency and improve usability, CEAL provides its own memory manager. The memory manager performs automatic garbage collection of allocations performed in the core (via CEAL's allocator) but not in the mutator. The language does not require the programmer to use the provided memory manager, the programmer can manage memory explicitly if so desired.

The Core Language. The core language extends C with modifiables, which are objects that consist of word-sized values (their *contents*) and supports the following primitive operations, which essentially allow the programmer to treat modifiables like ordinary pointers in memory.

`modref_t* modref():` creates a(n) (empty) modifiable
`void write(modref_t *`m`, void *`p`):` writes p into m
`void* read(modref_t *`m`):` returns the contents of m

In addition to operations on modifiables, CEAL provides the `alloc` primitive for memory allocation. For correctness of change propagation, core functions must modify the heap only through modifiables, i.e., memory accessed within the core, excluding modifiables and local variables, must be write-once. Also by definition, core functions return `ceal` (nothing). Since modifiables can be written arbitrarily, these restrictions cause no loss of generality or undue burden on the programmer.

The Meta Language. The meta language also provides primitives for operating on modifiables; `modref` allocates and returns a modifiable, and the following primitives can be used to access and update modifiables:

`void* deref(modref_t *`m`):` returns the contents of m.
`void modify(modref_t *`m`, void *`p`):` modifies the contents of the modifiable m to contain p.

As in the core, the `alloc` primitive can be used to allocate memory. The memory allocated at the meta level, however, needs to be explicitly freed. The CEAL language provides the `kill` primitive for this purpose.

In addition to these, the meta language offers primitives for starting a self-adjusting computation, `run_core`, and updating it via change propagation, `propagate`. The `run_core` primitive takes a pointer to a core function f and the arguments $\overline{a}$, and runs f with $\overline{a}$. The `propagate` primitive updates the core computation created by `run_core` to match the modifications performed by the mutator via `modify`.[1] Except for modifiables, the mutator may not modify memory accessed by the core program. The meta language makes no other restrictions: it is a strict superset of C.

[1] The actual language offers a richer set of operations for creating multiple self-adjusting cores simultaneously. Since the meta language is not our focus here we restrict ourselves to this simpler interface.

```
typedef enum { NODE, LEAF} kind_t;
typedef struct {
  kind_t kind;
  enum { PLUS, MINUS } op;
  modref_t *left, *right;
} node_t;
typedef struct {
  kind_t kind;
  int num;
} leaf_t;
```

Figure 1. Data type definitions for expression trees.

```
ceal eval (modref_t *root, modref_t *res) {
  node_t *t = read (root);
  if (t->kind == LEAF) {
    write (res,(void*)((leaf_t*) t)->num);
  } else {
    modref_t *m_a = modref ();
    modref_t *m_b = modref ();
    eval (t->left, m_a);
    eval (t->right, m_b);
    int a = (int) read (m_a);
    int b = (int) read (m_b);
    if (t->op == PLUS) {
      write (res, (void*)(a + b));
    } else {
      write (res, (void*)(a - b));
    }
  }
  return;
}
```

Figure 2. The `eval` function written in CEAL (core).

```
exp = "(3_d +_c 4_e) -_b (1_g -_f 2_h) +_a (5_j -_i 6_k)";
tree = buildTree (exp);
result = modref ();
run_core (eval, tree, result);
subtree = buildTree ("6_m +_l 7_n");
t = find ("k",subtree);
modify (t,subtree);
propagate ();
```

Figure 3. The mutator written in CEAL (meta).

3. Example and Overview

We give an example CEAL program and give an overview of the compilation process (Sections 5 and 6) via this example.

3.1 An Example: Expression Trees

We present an example CEAL program for evaluating and updating expression trees. Figure 1 shows the data type definitions for expressions trees. A tree consists of leaves and nodes each represented as a record with a `kind` field indicating their type. A `node` additionally has an operation field, and left & right children placed in modifiables. A leaf holds an integer as data. For illustrative purposes, we only consider plus and minus operations. [2] This representation differs from the conventional one only in that the children are stored in modifiables instead of pointers. By storing the children in modifiables, we enable the mutator to modify the expression and update the result by performing change propagation.

Figure 2 shows the code for `eval` function written in core CEAL. The function takes as arguments the root of a tree (in a modifiable) and a result modifiable where it writes the result of evaluating the tree. It starts by reading the root. If the root is a leaf, then its value is written into the result modifiable. If the

[2] In their most general form, expression trees can be used to compute a range of properties of trees and graphs. We discuss such a general model in our experimental evaluation.

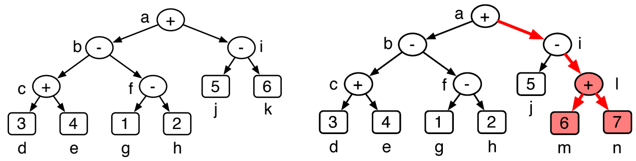

Figure 4. Example expression trees.

root is an internal node, then it creates two result modifiables (m_a and m_b) and evaluates the left and the right subexpressions. The function then reads the resulting values of the subexpressions, combines them with the operation specified by the root, and writes the value into the result. This approach to evaluating a tree is standard: replacing modifiables with pointers, reads with pointer dereference, and writes with assignment yields the conventional approach to evaluating trees. Unlike the conventional program, the CEAL program can respond to updates quickly.

Figure 3 shows the pseudo-code for a simple mutator example illustrated in Figure 4. The mutator starts by creating an expression tree from an expression where each subexpression is labeled with a unique key, which becomes the label of each node in the expression tree. It then creates a result modifiable and evaluates the tree with `eval`, which writes the value 7 into the result. The mutator then modifies the expression by substituting the subexpression $(6_m \ +_l \ 7_n)$ in place of the modifiable holding the leaf k and updates the computation via change propagation. Change propagation updates the result to 0, the new value of the expression. By representing the data and control dependences in the computation accurately, change propagation updates the computation in time proportional to the length of the path from k (changed leaf) to the root, instead of the total number of nodes as would be conventionally required. We evaluate a variation of this program (which uses floating point numbers in place of integers) in Section 8 and show that change propagation updates these trees efficiently.

3.2 Overview of Compilation

Our compilation technique translates CEAL programs into C programs that rely on a run-time-system *RTS* (Section 6.1) to provide self-adjusting-computation primitives. Compilation treats the mutator and the core separately. To compile the mutator, we simply replace meta-CEAL calls with the corresponding RTS calls—no major code restructuring is needed. In this paper, we therefore do not discuss compilation of the meta language in detail.

Compiling core CEAL programs is more challenging. At a high level, the primary difficulty is determining the *code dependence* for the modifiable being read, i.e., the piece of code that depends on the modifiable. More specifically, when we translate a read of modifiable to an RTS call, we need to supply a closure that encapsulates all the code that uses the value of the modifiable being read. Since CEAL treats references just like conventional pointers, it does not make explicit what that closure should be. In the context of functional languages such as SML, Ley-Wild et al used a continuation-passing-style transformation to solve this problem [27]. The idea is to use the continuation of the read as a conservative approximation of the code dependence. Supporting continuations in stack-based languages such as C is expensive and cumbersome. Another approach is to use the source function that contains the read as an approximation to the code dependence. This not only slows down change propagation by executing code unnecessarily but also can cause code to be executed multiple times, e.g., when a function (caller) reads a modified value from a callee, the caller has to be executed, causing the callee to be executed again.

To address this problem, we use a technique that we call normalization (Section 5). Normalization restructures the program such

```
1   ceal eval (modref_t *root, modref_t *res) {
2     node_t *t = read (root);
      tail read_r (t,res);
19  }

a   ceal read_r (node_t *t, modref_t *res) {
3     if (t->kind == LEAF) {
4       write (res,(void*)((leaf_t*) t)->num);
        tail eval_final ();
5     } else {
6       modref_t *m_a = modref ();
7       modref_t *m_b = modref ();
8       eval (t->left, m_a);
9       eval (t->right, m_b);
10      int a = (int) read (m_a);
        tail read_a (res,a,m_b);
17    }
    }

b   ceal read_a (modref_t *res, int a, modref_t *m_b) {
11    int b = (int) read (m_b);
      tail read_b (res,a,b);
    }

c   ceal read_b (modref_t *res, int a, int b) {
12    if (t->op == PLUS) {
13      write (res, (void*)(a + b));
        tail eval_final ();
14    } else {
15      write (res, (void*)(a - b));
        tail eval_final ();
16    }
    }

d   ceal eval_final () {
18    return;
    }
```

Figure 5. The normalized expression-tree evaluator.

that each read operation is followed by a tail call to a function that marks the start of the code that depends on the modifiable being read. The dynamic scope of the function tail call ends at the same point as that of the function that the read is contained in. To normalize a CEAL program, we first construct a specialized rooted control-flow graph that treats certain nodes—function nodes and nodes that immediately follow read operations—as *entry nodes* by connecting them to the root. The algorithm then computes the dominator tree for this graph to identify what we call *units*, and turns them into functions, making the necessary transformations for these functions to be tail-called. We prove that normalization runs efficiently and does not increase the program by more than a factor of two (in terms of its graph nodes).

As an example, Figure 5 shows the code for the normalized expression evaluator. The numbered lines are taken directly from the original program in Figure 2. The highlighted lines correspond to the new function nodes and the tail calls to these functions. Normalization creates the new functions, `read_r`, `read_a`, `read_b` and tail-calls them after the reads lines 2, 10, and 11 respectively. Intuitively, these functions mark the start of the code that depend on the `root`, and the result of the left and the right subtrees (m_a and m_b respectively). The normalization algorithm creates a trivial function, `eval_final` for the `return` statement to ensure that the `read_a` and `read_b` branch out of the conditional—otherwise the correspondence to the source program may be lost.[3]

We finalize compilation by translating the normalized program into C. To this end, we present a *basic translation* that creates closures as required by the RTS and employs trampolines[4] to support tail calls without growing the stack (Section 6.2). This basic

[3] In practice we eliminate such trivial calls by inlining the `return`.

[4] A trampoline is a dispatch loop that iteratively runs a sequence of closures.

Types	τ	::=	int | modref_t | $\tau*$
Values	v	::=	$\ell \mid n$
Prim. op's	o	::=	$\oplus \mid \ominus \mid \ldots$
Expressions	e	::=	$v \mid o(\overline{x}) \mid x[y]$
Commands	c	::=	nop | $x := e$ | $x[y] := e$
		|	$x :=$ modref() | $x :=$ read y | write x y
		|	$x :=$ alloc y f $\overline{z}$ | call f $(\overline{x})$
Jumps	j	::=	goto l | tail f $(\overline{x})$
Basic Blocks	b	::=	$\{l :$ done$\}$ | $\{l :$ cond x j_1 $j_2\}$ | $\{l : c\ ;\ j\}$
Fun. Defs	F	::=	$f\ (\overline{\tau_1\ x})\ \{\overline{\tau_2\ y}; \overline{b}\}$
Programs	P	::=	$\overline{F}$

Figure 6. The syntax of CL.

translation is theoretically satisfactory but it is practically expensive, both because it requires run-time type information and because trampolining requires creating a closure for each tail call. We therefore present two major refinements that apply trampolining selectively and that monomorphize the code by statically generating type-specialized instances of certain functions to eliminate the need for run-time type information (Section 6.3). It is this refined translation that we implement (Section 7).

We note that the quality of the self-adjusting program generated by our compilation strategy depends on the source code. In particular, if the programmer does not perform the reads close to where the values being read are used, then the generated code may not be effective. In many cases, it is easy to detect and statically eliminate such poor code by moving reads appropriately—our compiler performs a few such optimizations. Since such optimizations are orthogonal to our compilation strategy (they can be applied independently), we do not discuss them further in this paper.

4. The Core Language

We formalize core-CEAL as a simplified variant of C called CL (Core Language) and describe how CL programs are executed.

4.1 Abstract Syntax

Figure 6 shows the abstract syntax for CL. The meta variables x and y (and variants) range over an unspecified set of variables and the meta variables ℓ (and variants) range over a separate, unspecified set of (memory) locations. For simplicity, we only include integer, modifiable and pointers types; the meta variable τ (and variants) range over these types. The type system of CL mirrors that of C, and thus, it offers no strong typing guarantees. Since the typing rules are standard we do not discuss them here.

The language distinguishes between values, expressions, commands, jumps, and basic blocks. A value v is either a memory location ℓ or an integer n. Expressions include values, primitive operations (e.g. plus, minus) applied to a sequence of variables, and array dereferences $x[y]$. Commands include no-op, assignment into a local variable or memory location, modifiable creation, read from a modifiable, write into a modifiable, memory allocation, and function calls. Jumps include goto jumps and tail jumps. Programs consist of a set of functions defined by a name f, a sequence of formal arguments $\overline{\tau_1\ x}$, local variable declarations $\overline{\tau_2\ y}$ and a body of basic blocks. Each basic block, labeled uniquely, takes one of three forms: a *done block*, $\{l :$ done$\}$, a *conditional block*, $\{l :$ cond x j_1 $j_2\}$, and a *command-and-jump block*, $\{l : c\ ;\ j\}$. When referring to command-and-jump blocks, we sometimes use the type of the command, e.g., a read block, a write block, regardless of the jump. Symmetrically, we sometimes use the type of the jump, e.g., a goto block, a tail-call block, regardless of the command. Note that since there are no return instructions, a function cannot return a value (since they can write to modifiables arbitrarily, this causes no loss of generality).

4.2 Execution (Operational Semantics)

Execution of a CL program begins when the mutator uses run_core to invoke one of its functions (e.g, as in Figure 3). Most of the operational (dynamic) semantics of CL should be clear to the reader if s/he is familiar with C or similar languages. The interesting aspects include tail jumps, operations on modifiables and memory allocation. A tail jump executes like a conventional function call, except that it never returns. The modref command allocates a modifiable and returns its location. Given a modifiable location, the read command retrieves its contents, and the write command destructively updates its contents with a new value. A done block, $\{k :$ done$\}$, completes execution of the current function and returns. A conditional block, $\{k :$ cond x j_1 $j_2\}$, checks the value of local variable x and performs jump j_1 if the condition is true (non-zero) and jump j_2 otherwise. A command block, $\{l : c\ ;\ j\}$, executes the command c followed by jump j. The alloc command allocates an array at some location ℓ with a specified size in bytes (y) and uses the provided function f to initialize it by calling f with ℓ and the additional arguments ($\overline{z}$). After initialization, it returns ℓ. By requiring this stylized interface for allocation, CL makes it easier to check and conform to the correct-usage restrictions (defined below) by localizing initialization effects (the side effects used to initialize the array) to the provided initialization function.

To accurately track data-dependencies during execution, we require, but do not enforce that the programs conform to the following *correct-usage* restrictions: 1) each array is side-effected only during the initialization step and 2) that the initialization step does not read or write modifiables.

4.3 From CEAL to CL

We can translate CEAL programs into CL by 1) replacing sequences of statements with corresponding command blocks connected via goto jumps, 2) replacing so-called "structured control-flow" (e.g., if, do, while, etc.) with corresponding (conditional) blocks and goto jumps, and 3) replacing return with done.

5. Normalizing CL Programs

We say that a program is in *normal form* if and only if every read command is in a tail-jump block, i.e., followed immediately by a tail jump. In this section, we describe an algorithm for normalization that represents programs with control flow graphs and uses dominator trees to restructure them. Section 5.4 illustrates the techniques described in this section applied to our running example.

5.1 Program Graphs

We represent CL programs with a particular form of rooted control flow graphs, which we shortly refer to as a *program graph* or simply as a *graph* when it is clear from the context.

The graph for a program P consists of nodes and edges, where each node represents a function definition, a (basic) block, or a distinguished root node (Section 5.4 shows an example). We tag each non-root node with the label of the block or the name of the function that it represents. Additionally, we tag each node representing a block with the code for that block and each node representing a function, called a *function node*, with the prototype (name and arguments) of the function and the declaration of local variables. As a matter of notational convenience, we name the nodes with the label of the corresponding basic block or the name of the function, e.g., u_l or u_f.

The edges of the graph represent control transfers. For each goto jump belonging to a block $\{k : c\ ;$ goto $l\}$, we have an edge from node u_k to node u_l tagged with goto l. For each function node u_f whose first block is u_l, we have an edge from u_f to u_l labeled with goto l. For each tail-jump block $\{k : c\ ;$ tail $f(\overline{x})\}$, we have an edge from u_k to u_f tagged with `tail` $f(\overline{x})$. If a node u_k represents a `call`-instruction belonging to a block $\{k$: `call` $f(\overline{x})$; $j\}$, then we insert an edge from u_k to u_f and tag it with `call` $f(\overline{x})$. For each conditional block $\{k$: `cond` x j_1 $j_2\}$ where j_1 and j_2 are the jumps, we insert edges from k to targets of j_1 and j_2, tagged with `true` and `false`, respectively.

We call a node a *read-entry* node if it is the target of an edge whose source is a read node. More specifically, consider the nodes u_k belonging to a block of the form $\{k : x$:= `read` y ; $j\}$ and u_l which is the target of the edge representing the jump j; the node u_l is a read entry. We call a node an *entry node* if it is a read-entry or a function node. For each entry node u_l, we insert an edge from the root to u_l into the graph.

There is a (efficiently) computable isomorphism between a program and its graph that enables us to treat programs and graphs as a single object. In particular, by changing the graph of a program, our normalization algorithm effectively restructures the program itself.

Property 1 (Programs and Graphs)
The program graph of a CL *program with n blocks can be constructed in expected $O(n)$ time. Conversely, given a program graph with m nodes, we can construct its program in $O(m)$ time.*

5.2 Dominator Trees and Units

Let $G = (V, E)$ be a rooted program graph with root node u_r. Let $u_k, u_l \in V$ be two nodes of G. We say that u_k *dominates* u_l if every path from u_r to u_l in G passes through u_k. By definition, every node dominates itself. We say that u_k is an *immediate dominator* of u_l if $u_k \neq u_l$ and u_k is a dominator of u_l such that every other dominator of u_l also dominates u_k. It is easy to show that each node except for the root has a unique immediate dominator (e.g., [26]). The immediate-dominator relation defines a tree, called a *dominator tree* $T = (V, E_I)$ where by $E_I = \{(u_k, u_l) \mid u_k \text{ is an immediate dominator of } u_l\}$.

Let T be a dominator tree of a rooted program graph $G = (V, E)$ with root u_r. Note that the root of G and T are both the same. Let u_l be a child of u_r in T. We define the *unit of* u_l as the vertex set consisting of u_l and all the descendants of u_l in T; we call u_l the *defining node* of the unit. Normalization is made possible by an interesting property of units and cross-unit edges.

Lemma 2 (Cross-Unit Edges)
Let $G = (V, E)$ be a rooted program graph and T be its dominator tree. Let U_k and U_m be two distinct units of T defined by vertices u_k and u_m respectively . Let $u_l \in U_k$ and $u_n \in U_m$ be any two vertices from U_k and U_m. If $(u_l, u_n) \in E$, i.e., a cross-unit edge in the graph, then $u_n = u_m$.

Proof: Let u_r be the root of both T and G. For a contradiction, suppose that $(u_l, u_n) \in E$ and $u_n \neq u_m$. Since $(u_l, u_n) \in E$ there exists a path $p = u_r \rightsquigarrow u_k \rightsquigarrow u_l \rightarrow u_n$ in G. Since u_m is a dominator of u_n, this means that u_m is in p, and since $u_m \neq u_n$ it must be the case that either u_k proceeds u_m in p or vice versa. We consider the two cases separately and show that they each lead to a contradiction.

- If u_k proceeds u_m in p then $p = u_r \rightsquigarrow u_k \rightsquigarrow u_m \rightsquigarrow u_l \rightarrow u_n$. But this means that u_l can be reached from u_r without going through u_k (since $u_r \rightarrow u_m \in G$). This contradicts the assumption that u_k dominates u_l.
- If u_m proceeds u_k in p then $p = u_r \rightsquigarrow u_m \rightsquigarrow u_k \rightsquigarrow u_l \rightarrow u_n$. But this means that u_n can be reached from u_r without going through u_m (since $u_r \rightarrow u_k \in G$). This contradicts the assumption that u_m dominates u_n. ■

```
NORMALIZE ($P$) =
  Let $G = (V, E)$ be the graph of $P$ rooted at $u_r$
  Let $T$ be the dominator tree of $G$ (rooted at $u_r$)
  $G' = (V', E')$, where $V' = V$ and $E' = \emptyset$
  for each unit $U$ of $T$ do
    $u_l$ ← defining node of $U$
    if $u_l$ is a function node then
      (* $u_l$ is not critical *)
      $E_U \leftarrow \{(u_l, u) \in E \mid (u \in U)\}$
    else
      (* $u_l$ is critical *)
      pick a fresh function $f$, i.e., $f \notin \mathsf{funs}(P)$
      let $\overline{x}$ be the live variables at $l$ in $P$
      let $\overline{y}$ be the free variables in $U$
      $\overline{z} = \overline{y} \setminus \overline{x}$
      $V' \leftarrow V' \cup \{u_f\}$
      tag$(u_f) = f(\overline{x})\{\overline{z}\}$
      $E_U \leftarrow \{(u_r, u_f), (u_f, u_l)\}$ ∪
              $\{(u_1, u_2) \in E \mid u_1, u_2 \in U \wedge u_2 \neq u_l\}$
      for each critical edge $(u_k, u_l) \in E$ do
        if $u_k \notin U$ then
          (* cross-unit edge *)
          $E_U \leftarrow E_U \cup \{(u_k, u_f)\}$
          tag$((u, u_f))$ = tail $f(\overline{x})$
        else
          (* intra-unit edge *)
          if $u_k$ is a read node then
            $E_U \leftarrow E_U \cup \{(u_k, u_f)\}$
            tag$((u, u_f))$ = tail $f(\overline{x})$
          else
            $E_U \leftarrow E_U \cup \{(u_k, u_l)\}$
      $E' \leftarrow E' \cup E_U$
```

Figure 7. Pseudo-code for the normalization algorithm.

5.3 An Algorithm for Normalization

Figure 7 gives the pseudo-code for our normalization algorithm, `NORMALIZE`. At a high level, the algorithm restructures the original program by creating a function from each unit and by changing control transfers into these units to tail jumps as necessary.

Given a program P, we start by computing the graph G of P and the dominator tree T of G. Let u_r be the root of the dominator tree and the graph. The algorithm computes the normalized program graph $G' = (V', E')$ by starting with a vertex set equal to that of the graph $V' = V$ and an empty edge set $E' = \emptyset$. It proceeds by considering each unit U of T.

Let U be a unit of T and let the node u_l of T be the defining node of the unit. If u_l is a not a function node, then we call it and all the edges that target it *critical*. We consider two cases to construct a set of edges E_U that we insert into G' for U. If u_l is not critical, then E_U consists of all the edges whose target is in U. If u_l is critical, then we define a function for it by giving it a fresh function name f and computing the set of live variables $\overline{x}$ at block l in P, and free variables $\overline{y}$ in all the blocks represented by U. The set $\overline{x}$ becomes the formal arguments to the function f and the set $\overline{z}$ of variables defined as the remaining free variables, i.e., $\overline{y} \setminus \overline{x}$, become the locals. We then insert a new node u_f into the normalized graph and insert the edges (u_r, u_f) and (u_f, u_l) into E_U as well as all the edges internal to U that do not target u_l. This creates the new function $f(\overline{x})$ with locals $\overline{z}$ whose body is defined by the basic blocks of the unit U. Next, we consider critical edges of the form (u_k, u_l). If the edge is a cross-unit edge, i.e., $u_k \notin U$, then we replace it with an edge into u_f by inserting (u_k, u_f) into E_U and tagging the edge with a tail jump to the defined function f representing U. If the critical edge is an intra-unit edge, i.e., $u_k \in U$, then we have two cases to consider: If u_k is a read node, then we insert (u_k, u_f) into E_U and tag it with a tail jump to function f with the appropriate arguments, effectively redirecting the edge to u_f. If u_k is not a read node, then we insert the edge into E_U, effectively leaving it intact. Although the algorithm only

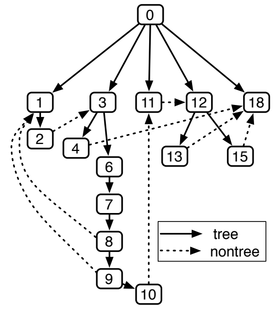

Figure 9. The dominator tree for the graph in Figure 8.

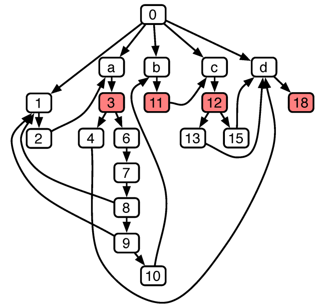

Figure 10. Normalized version of the graph in Figure 8.

redirects critical edges Lemma 2 shows this is sufficient: all other edges in the graph are contained within a single unit and hence do not need to be redirected.

5.4 Example

Figure 8 shows the rooted graph for the expression-tree evaluator shown in Figure 2 after translating it into CL (Section 4.3). The nodes are labeled with the line numbers that they correspond to; `goto` edges and control-branches are represented as straight edges, and (tail) calls are represented with curved edges. For example, node 0 is the root of the graph; node 1 is the entry node for the function `eval`; node 3 is the conditional on line 3; the nodes 8 and 9 are the recursive calls to `eval`. The nodes 3, 11, 12 are read-entry nodes.

Figure 9 shows the dominator tree for the graph (Figure 8). For illustrative purposes the edges of the graph that are not tree edges are shown as dotted edges.[5] The units are defined by the vertices 1, 3, 11, 12, 18. The cross-unit edges, which are non-tree edges, are (2,3), (4,18), (10,11), (11,12), (13,18), (15,18). Note that as implied by Lemma 2, all these edges target the defining nodes of the units.

Figure 10 shows the rooted program graph for the normalized program created by algorithm `NORMALIZE`. By inspection of Figures 9 and 2, we see that the critical nodes are 3, 11, 12, 18, because they define units but they are not function nodes. The algorithm creates the fresh nodes a, b, c, and d for these units and redirects all the cross-unit edges into the new function nodes and tagged with tail jumps (tags not shown). In this example, we have no intra-unit critical edges. Figure 5 shows the code for the normalized graph (described in Section 3.2).

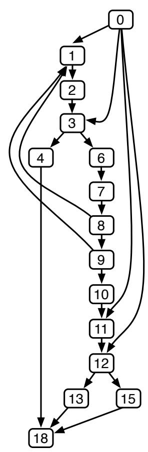

Figure 8.

5.5 Properties of Normalization

We state and prove some properties about the normalization algorithm and the (normal) programs it produces. The normalization algorithm uses a live variable analysis to determine the formal and actual arguments for each fresh function (see line 13 in Figure 7). We let $T_L(P)$ denote the time required for live variable analysis of a CL program P. The output of this analysis for program P is a function live$(\cdot)$, where live(l) is the set variables which are live at (the start of) block $l \in P$. We let $M_L(P)$ denote the maximum number of live variables for any block in program P, i.e., $\max_{l \in P} |\mathsf{live}(l)|$. We assume that each variable, function name, and block label require one word to represent. The following theorems relate the size of a CL program before and after normalization (Theorem 3) and bound the time required to perform normalization (Theorem 4).

[5] Note that the dominator tree can have edges that are not in the graph.

Theorem 3 (Size of Output Program)
If CL program P has n blocks and $P' = \mathtt{NORMALIZE}(P)$, then P' also has n blocks and at most n additional function definitions. Furthermore if it takes m words to represent P, then it takes $O(m + n \cdot M_L(P))$ words to represent P'.

Proof: Observe that the normalization algorithm creates no new blocks—just new function nodes. Furthermore, since at most one function is created for each critical node, which is a block, the algorithm creates at most one new function for each block of P. Thus, the first bound follows.

For the second bound, note that since we create at most one new function for each block, we can name each function using the block label followed by a marker (stored in a word), thus requiring no more than $2n$ words. Since each fresh function has at most $M_L(P)$ arguments, representing each function signature requires $O(M_L(P))$ additional words (note that we create no new variable names). Similarly, each call to a new function requires $O(M_L(P))$ words to represent. Since the number of new functions and new calls is bounded by n, the total number of additional words needed for the new function signatures and the new function calls is bounded by $O(m + n \cdot M_L(P))$. ■

Theorem 4 (Time for Normalization)
If CL program P has n blocks then running $\mathtt{NORMALIZE}(P)$ takes $O(n + n \cdot M_L(P) + T_L(P))$ time.

Proof: Computing the dominator tree takes linear time [19]. By definition, computing the set of live variables for each node takes $T_L(P)$ time. We show that the remaining work can be done in $O(n + n \cdot M_L(P))$ time. To process each unit, we check if its defining node is a function node. If so, we copy each incoming edge from the original program. If not, we create a fresh function node, copy non-critical edges, and process each incoming critical edge. Since each node has a constant out degree (at most two), the total number of edges considered per node is constant. Since each defining node has at most $M_L(P)$ live variables, it takes $O(M_L(P))$ time to create a fresh function. Replacing a critical edge with a tail jump edge requires creating a function call with at most $M_L(P)$ arguments, requiring $O(M_L(P))$ time. Thus, it takes $O(n + n \cdot M_L(P))$ time to process all the units. ■

6. Translation to C

The translation phase translates normalized CL code into C code that relies on a run-time system providing self-adjusting computation primitives. To support tail jumps in C without growing the stack we adapt a well-known technique called *trampolining* (e.g., [24, 36]). At a high level, a trampoline is a loop that runs a sequence of closures that, when executed, either return another closure or `NULL`, which causes the trampoline to terminate.

6.1 The Run-Time System

Figure 11 shows the interface to the run-time system (RTS). The RTS provides functions for creating and running closures. The `closure_make` function creates a closure given a function and a complete set of matching arguments. The `closure_run` function sets up a trampoline for running the given closure (and the closures it returns, if any) iteratively. The RTS provides functions for creating, reading, and writing modifiables (`modref_t`). The

```
typedef struct {...} closure_t;
closure_t* closure_make(closure_t* (*f)(τ x), τ x);
void closure_run(closure_t* c);

typedef struct {...} modref_t;
void modref_init(modref_t *m);
void modref_write(modref_t *m, void *v);
closure_t* modref_read(modref_t *m, closure_t *c);

void* allocate(size_t n, closure_t *c);
```

Figure 11. The interface for the run-time system.

Expressions:

$$[\![e]\!] \;=\; e$$

Commands:

$$
\begin{array}{rcl}
[\![\texttt{nop}]\!] & = & \texttt{;} \\
[\![x := e]\!] & = & x\ \texttt{:=}\ [\![e]\!]\texttt{;} \\
[\![x[y] := e]\!] & = & x\texttt{[}y\texttt{]}\ \texttt{:=}\ [\![e]\!]\texttt{;} \\
[\![\texttt{call}\ f(\overline{x})]\!] & = & \texttt{closure\_run(}f(\overline{x})\texttt{);} \\
[\![x := \texttt{alloc}\ y\ f\ \overline{z}]\!] & = & \texttt{closure\_t *}c\texttt{;} \\
 & & c\ \texttt{:=}\ \texttt{closure\_make(}f\texttt{,NULL::}\overline{z}\texttt{);} \\
 & & x\ \texttt{:=}\ \texttt{allocate(}y\texttt{,}c\texttt{);} \\
[\![x := \texttt{modref}()]\!] & = & [\![x := \texttt{alloc (sizeof(modref\_t))} \\
 & & \qquad\qquad \texttt{modref\_init}\ \langle\rangle]\!] \\
[\![\texttt{write}\ x\ y]\!] & = & \texttt{modref\_write(}x\texttt{, }y\texttt{);}
\end{array}
$$

Jumps:

$$
\begin{array}{rcl}
[\![\texttt{goto}\ l]\!] & = & \texttt{goto}\ l\texttt{;} \\
[\![\texttt{tail}\ f(\overline{x})]\!] & = & \texttt{return (closure\_make(}f\texttt{,}\overline{x}\texttt{));}
\end{array}
$$

Basic Blocks:

$$
\begin{array}{rcl}
[\![\{l : \texttt{done}\}]\!] & = & \{l\texttt{: return NULL;}\} \\
[\![\{l : \texttt{cond}\ x\ j_1\ j_2\}]\!] & = & \{l\texttt{: if (}x\texttt{)}\ \{[\![j_1]\!]\}\ \texttt{else}\ \{[\![j_2]\!]\}\} \\
[\![\{l : c\ ;\ j\}]\!] & = & \{l\texttt{: } [\![c]\!]\texttt{; } [\![j]\!]\} \\
[\![\{l : x := \texttt{read}\ y\ ; & = & \{l\texttt{: closure\_t *}c\texttt{;} \\
\quad \texttt{tail}\ f(x,\overline{z})\}]\!] & & \quad c\ \texttt{:=}\ \texttt{closure\_make(}f\texttt{,NULL::}\overline{z}\texttt{);} \\
 & & \quad \texttt{return (modref\_read(}y\texttt{,}c\texttt{));}\}
\end{array}
$$

Functions:

$$[\![f\ (\overline{\tau_x\ x})\ \{\overline{\tau_y\ y}\ ;\ \overline{b}\}]\!] \;=\; \texttt{closure\_t* }f\texttt{(}\overline{\tau_x\ x}\texttt{)}\{\overline{\tau_y\ y\texttt{;}}\ [\![\overline{b}]\!]\}$$

Figure 12. Translation of CL into C.

`modref_init` function initializes memory for a modifiable; together with `allocate` this function can be used to allocate modifiables. The `modref_write` function updates the contents of a modifiable with the given value. The `modref_read` function reads a modifiable, substitutes its contents as the first argument of the given closure, and returns the updated closure. The `allocate` function allocates a memory block with the specified size (in bytes), runs the given closure after substituting the address of the block for the first argument, and returns the address of the block. The closure acts as the initializer for the allocated memory.

6.2 The Basic Translation

Figure 12 illustrates the rules for translating normalized CL programs into C. For clarity, we deviate from C syntax slightly by using `:=` to denote the C assignment operator.

The most interesting cases concern function calls, tail jumps, and modifiables. To support trampolining, we translate functions to return closures. A function call is translated into a direct call whose return value (a closure) is passed to a trampoline, `closure_run`. A tail jump simply creates a corresponding closure and returns it to the active trampoline. Since each tail jump takes place in the context of a function, there is always an active trampoline. The translation of `alloc` first creates a closure from the given initialization function and arguments prepended with `NULL`, which acts as a place-holder for the allocated location. It then supplies this closure and the specified size to `allocate`.

We translate `modref` as a special case of allocation by supplying the size of a modifiable and using `modref_init` as the initialization function. Translation of write commands is straightforward. We translate a command-and-jump block by separately translating the command and the jump. We translate reads to create a closure for the tail jump following the read command and call `modref_read` with the closure & the modifiable being read. As with tail jumps, the translated code returns the resulting closure to the active trampoline. When creating the closure, we assume, without loss of generality, that the result of the read appears as the first argument to the tail jump and use `NULL` as a placeholder for the value read. Note that, since translation is applied after normalization, all read commands are followed by a tail jump, as we assume here.

6.3 Refinements for Performance

One disadvantage of the basic translation described above is that it creates closures to support tail jumps; this is known to be slow (e.g., [24]). To address this problem, we trampoline only the tail jumps that follow reads, which we call *read trampolining*, by refining the translation as follows: $[\![\texttt{tail}\ f\ (\overline{x})]\!] = \texttt{return}\ f(\overline{x})$. This refinement treats all tail calls other than those that follow a read as conventional function calls. Since the translation of a read already creates a closure, this eliminates the need to create extra closures. Since in practice self-adjusting programs perform reads periodically (popping the stack frames down to the active trampoline), we observe that the stack grows only temporarily.

Another disadvantage is that the translated code uses the RTS function `closure_make` with different function and argument types, i.e., polymorphically. Implementing this requires run-time type information. To address this problem, we apply monomorphisation [37] to generate a set of `closure_make` functions for each distinctly-typed use in the translated program. Our use of monomorphisation is similar to that of the MLton compiler, which uses it for compiling SML, where polymorphism is abundant [1].

6.4 Bounds for Translation and Compilation

By inspecting Figure 12, we see that the basic translation requires traversing the program once. Since the translation is purely syntactic and in each case it expands code by a constant factor, it takes linear time in the size of the normalized program. As for the refinements, note that read trampolining does not affect the bound since it only performs a simple syntactic code replacement. Monomorphisation requires generating specialized instances of the `closure_make` function. Each instance with k arguments can be represented with $O(k)$ words and can be generated in $O(k)$ time. Since k is bounded by the number of arguments of the `tail` jump (or `alloc` command) being translated, monomorphisation requires linear time in the size of the normalized code. Thus, we conclude that we can translate a normalized program in linear time while generating C code whose size is within a constant factor of the normalized program. Putting together this bound and the bounds from normalization (Theorems 3 and 4), we can bound the time for compilation and the size of the compiled programs.

Theorem 5 (Compilation)
Let P be a CL program with n blocks that requires m words to represent. Let $M_L(P)$ be the maximum number of live variables over all blocks of P and let $T_L(P)$ be the time for live-variable analysis. It takes $O(n + n \cdot M_L(P) + T_L(P))$ time to compile the program into C. The generated C code requires $O(m + n \cdot M_L(P))$ words to represent.

7. Implementation

The implementation of our compiler, `cealc`, consists of a front-end and a runtime library. The front-end transforms CEAL code into C code. We use an off-the-shelf compiler (`gcc`) to compile the translated code and link it with the runtime library.

Our front-end uses an intra-procedural variant of our normalization algorithm that processes each core function independently from the rest of the program, rather than processing the core program as a whole (as presented in Section 5). This works because inter-procedural edges (i.e., tail jump and call edges) in a rooted graph don't impact its dominator tree—the immediate dominator of each function node is always the root. Hence, the subgraph for each function can be independently analyzed and transformed. Since each function's subgraph is often small compared to the total program size, we use a simple, iterative algorithm for computing dominators [30, 14] that is efficient on smaller graphs, rather than an asymptotically optimal algorithm with larger constant factors [19]. During normalization, we create new functions by computing their formal arguments as the live variables that flow into their entry nodes in the original program graph. We use a standard (iterative) liveness analysis for this step, run on a per-function basis (e.g., [30, 7]). Since control flow can be arbitrary (i.e., non-reducible) and the number of local variables can only be bounded by the number of nodes in the function's subgraph, n, this iterative algorithm has $O(n^3)$ worst case time for pathological cases. However, since n is often small compared to the total program size and since functions are usually not pathological, we observe that this approach works sufficiently well in practice.

Our front-end is implemented as an extension to CIL (C Intermediate Language), a library of tools used to parse, analyze and transform C code [31]. We provide the CEAL core and mutator primitives as ordinary C function prototypes. We implement the `ceal` keyword as a `typedef` for `void`. Since these syntactic extensions are each embedded within the syntax of C, we define them with a conventional C header file and do not modify CIL's C parser. To perform normalization, we identify the core functions (which are marked by the `ceal` return type) and apply the intra-procedural variant of the normalization algorithm. The translation from CL to C directly follows the discussions in Section 6. We translate mutator primitives using simple (local) code substitution.

Our runtime library provides an implementation of the interface discussed in Section 6. The implementation is built on previous work which focused on efficiently supporting automatic memory management for self-adjusting programs. We extend the previous implementation with support for tail jumps via trampolining, and support for imperative (multiple-write) modifiable references [4].

Our front-end extends CIL with about 5,000 lines of additional Objective Caml (OCaml) code, and our runtime library consists of about 5,000 lines of C code. Our implementation is available online at `http://ttic.uchicago.edu/~ceal`.

8. Experimental Results

We present an empirical evaluation of our implementation.

8.1 Experimental Setup and Measurements

We run our experiments on a 2Ghz Intel Xeon machine with 32 gigabytes of memory running Linux (kernel version 2.6). All our programs are sequential (no parallelism). We use `gcc` version 4.2.3 with "`-O3`" to compile the translated C code. All reported times are wall-clock times measured in milliseconds or seconds, averaged over five independent runs.

For each benchmark, we consider a *self-adjusting* and a *conventional* version. Both versions are derived from a single CEAL program. We generate the conventional version by replacing modifiable references with conventional references, represented as a word-sized location in memory that can be read and written. Unlike modifiable references, the operations on conventional references are not traced and thus conventional programs are not normalized. The resulting conventional versions are essentially the same as the static C code that a programmer would write for that benchmark.

For each self-adjusting benchmark we measure the time required for propagating a small modification by using a special *test mutator*. Invoked after an initial run of the self-adjusting version, the test mutator performs two modifications for each element of the input: it deletes the element and performs change propagation, it inserts the element back and performs change propagation. We report the average time for a modification as the total running time of the test mutator divided by the number of updates performed (usually two times the input size).

For each benchmark we measure the from-scratch running time of the conventional and the self-adjusting versions; we define the *overhead* as the ratio of the latter to the former. The overhead measures the slowdown caused by the dependence tracking techniques employed by self-adjusting computation. We measure the *speedup* for a benchmark as the ratio of the from-scratch running time of the conventional version divided by the average modification time computed by running the test mutator.

8.2 Benchmark Suite

Our benchmarks include list primitives, and more sophisticated algorithms for computational geometry and tree computations.

List Primitives. Our list benchmarks include the list primitives `filter`, `map`, and `reverse`, `minimum`, and `sum`, which we expect to be self-explanatory, and the sorting algorithms `quicksort` and `mergesort`. Our `map` benchmark maps each number x of the input list to $f(x) = \lfloor x/3 \rfloor + \lfloor x/7 \rfloor + \lfloor x/9 \rfloor$ in the output list. Our `filter` benchmark filters out an input element x if and only if $f(x)$ is odd. Our `reverse` benchmark reverses the input list. The list reductions `minimum` and `sum` find the minimum and the maximum elements in the given lists. 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.

Computational Geometry. Our computational-geometry benchmarks are `quickhull`, `diameter`, and `distance`; `quickhull` computes the convex-hull of a point set using the classic algorithm with the same name; `diameter` computes the diameter (the maximum distance between any two points) of a point set; `distance` computes the minimum distance between two sets of points. The implementations of `diameter` and `distance` use `quickhull` as a subroutine. For `quickhull` and `distance`, input points are drawn from a uniform distribution over the unit square in $\mathbb{R}^2$. For `distance`, two non-overlapping unit squares in $\mathbb{R}^2$ are used, and from each square we draw half the input points.

Tree-based Computations. The `exptrees` benchmark is a variation of our simple running example, and it uses floating-point numbers in place of integers. We generate random, balanced expression trees as input and perform modifications by changing their leaves. The `tcon` benchmark is an implementation of the tree-contraction technique by Miller and Reif [28]. This this a technique rather than a single benchmark because it can be customized to compute various properties of trees, e.g., [28, 29, 6]. For our experiments, we generate binary input trees randomly and perform a generalized contraction with no application-specific data or information. We measure the average time for an insertion/deletion of edge by iterating over all the edges as with other applications.

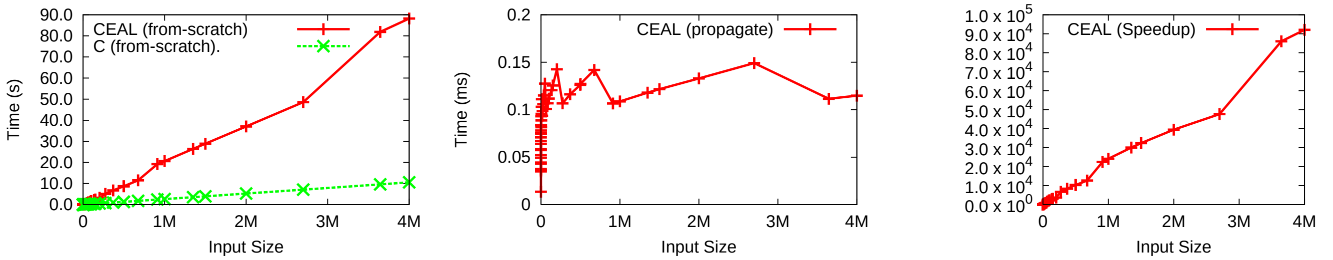

Figure 13. Experimental results for `tcon` (self-adjusting tree-contraction).

		From-Scratch			Propagation		
Application	n	Cnv.	Self.	O.H.	Ave. Update	Speedup	Max Live
`filter`	10.0M	0.5	7.4	14.2	2.1×10^{-6}	2.4×10^{5}	3017.2M
`map`	10.0M	0.7	11.9	17.2	1.6×10^{-6}	4.2×10^{5}	3494.6M
`reverse`	10.0M	0.6	11.9	18.8	1.6×10^{-6}	3.9×10^{5}	3494.6M
`minimum`	10.0M	0.8	10.9	13.8	4.8×10^{-6}	1.6×10^{5}	3819.4M
`sum`	10.0M	0.8	10.9	13.9	7.0×10^{-5}	1.1×10^{4}	3819.8M
`quicksort`	1.0M	3.5	22.4	6.4	2.4×10^{-4}	1.4×10^{4}	8956.7M
`quickhull`	1.0M	1.1	12.3	11.5	2.3×10^{-4}	4.6×10^{3}	6622.9M
`diameter`	1.0M	1.0	12.1	12.0	1.2×10^{-4}	8.3×10^{3}	6426.9M
`exptrees`	10.0M	1.0	7.2	7.2	1.4×10^{-6}	7.1×10^{5}	4821.1M
`mergesort`	1.0M	6.1	37.6	6.1	1.2×10^{-4}	5.1×10^{4}	15876.3M
`distance`	1.0M	1.0	11.0	11.0	1.3×10^{-3}	7.5×10^{2}	5043.6M
`rctree-opt`	1.0M	2.6	20.6	7.9	1.0×10^{-4}	2.5×10^{4}	5843.7M

Table 1. Summary of measurements with CEAL; all times in seconds and space in bytes.

		From-Scratch (Self.)			Propagation Time			Propagation Max Live		
Application	n	CEAL	SaSML	SaSML / CEAL	CEAL	SaSML	SaSML / CEAL	CEAL	SaSML	SaSML / CEAL
`filter`	1.0M	0.7	6.9	9.3	1.4×10^{-6}	8.7×10^{-6}	6.2	306.5M	1333.5M	4.4
`map`	1.0M	0.8	7.8	9.3	1.6×10^{-6}	1.1×10^{-5}	7.1	344.7M	1519.1M	4.4
`reverse`	1.0M	0.8	6.7	8.0	1.6×10^{-6}	9.2×10^{-6}	5.8	344.7M	1446.7M	4.2
`minimum`	1.0M	1.1	5.1	4.6	3.4×10^{-6}	3.0×10^{-5}	8.8	388.4M	1113.8M	2.9
`sum`	1.0M	1.1	5.1	4.6	4.8×10^{-5}	1.7×10^{-4}	3.5	388.5M	1132.4M	2.9
`quicksort`	100.0K	1.6	43.8	26.9	1.6×10^{-4}	2.6×10^{-3}	15.6	775.4M	3719.9M	4.8
`quickhull`	100.0K	1.0	5.1	5.1	1.0×10^{-4}	3.3×10^{-4}	3.3	657.9M	737.7M	1.1
`diameter`	100.0K	0.9	5.2	5.8	8.6×10^{-5}	3.7×10^{-4}	4.3	609.0M	899.5M	1.5

Table 2. Times and space for CEAL versus SaSML for a common set of benchmarks.

8.3 Results

For brevity, we illustrate detailed results for one of our benchmarks, tree contraction (`tcon`), and summarize the others. Figure 13 shows the results with `tcon`. The leftmost figure compares times for a from-scratch run of the conventional and self-adjusting versions; the middle graph shows the time for an average update; and the rightmost graph shows the speedup. The results show that self-adjusting version is slower by a constant factor (of about 8) than the conventional version. Change propagation time increases slowly (logarithmically) with the input size. This linear-time gap between recomputing from scratch and change propagation yields speedups that exceed four orders of magnitude even for moderately sized inputs. We also compare our implementation to that of an hand-optimized implementation, which is between 3–4 times faster. We interpret this is as a very encouraging results for the effectiveness of our compiler, which does not perform significant optimizations. The accompanying technical report gives more details on this comparison [22].

Table 1 summarizes our results for CEAL benchmarks at fixed input sizes of 1 million (written “1.0M”) and 10 million (written “10.0M”). From left to right, for each benchmark, we report the input size considered, the time for conventional and self-adjusting runs, the overhead, the average time for an update, the speedup, and the maximum live memory required for running the experiments (both from-scratch and test mutator runs). The individual graphs for these benchmarks resemble that of the tree contraction (Figure 13). For the benchmarks run with input size 10M, the average overhead is 14.2 and the average speedup is 1.4×10^5; for those run with input size 1M, the average overhead is 9.2 and the average speedup is 3.6×10^4. We note that for all benchmarks the speedups are scalable and continue to increase with larger input sizes.

8.4 Comparison to SaSML

To measure the effectiveness of the CEAL approach to previously proposed approaches, we compare our implementation to SaSML, the state-of-art implementation of self-adjusting computation in SML [27]. Table 2 shows the running times for the common bench-

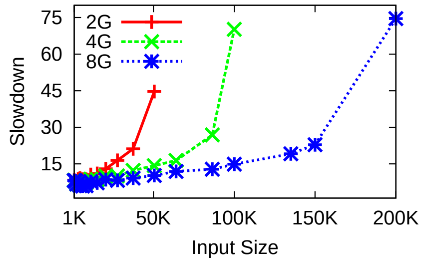

Figure 14. SaSML slowdown compared to CEAL for `quicksort`.

marks, taken on the same computer as the CEAL measurements, with inputs generated in the same way. For both CEAL and SaSML we report the from-scratch run time, the average update time and the maximum live memory required for the experiment. A column labeled $\frac{\text{SaSML}}{\text{CEAL}}$ follows each pair of CEAL and SaSML measurements and it reports the ratio of the latter to the former. Since the SaSML benchmarks do not scale well to the input sizes considered in Table 1, we make this comparison at smaller input sizes—1 million and 100 thousand (written "100.0K").

Comparing the CEAL and SaSML figures shows that CEAL is about 5–27 times faster for from-scratch runs (about 9 times faster on average) and 3–16 times faster for change propagation (about 7 times faster on average). In addition, CEAL consumes up to 5 times less space (about 3 times less space on average).

An important problem with the SaSML implementation is that it relies on traditional tracing garbage collectors (i.e., the collectors used by most SML runtime systems, including the runtime used by SaSML) which previous work has shown to be inherently incompatible with self-adjusting computation, preventing it from scaling to larger inputs [21]. Indeed, we observe that, while our implementation scales to larger inputs, SaSML benchmarks don't. To illustrate this, we limit the heap size and measure the change-propagation slowdown computed as the time for a small modification (measured by the test mutator) with SaSML divided by that with CEAL for the `quicksort` benchmark (Figure 14). Each line ends roughly when the heap size is insufficient to hold the live memory required for that input size. As the figure shows, the slowdown is not constant and increases super linearly with the input size to quickly exceed an order of magnitude (can be as high as 75).

8.5 Performance of the Compiler

We evaluate the performance of our compiler, `cealc`, using test programs from our benchmark suite. Each of the programs we consider consists of a core, which includes all the core-CEAL code needed to run the benchmark, and a corresponding test mutator for testing this core. We also consider a *test driver* program which consists of all the test mutators (one for each benchmark) and their corresponding core components.

We compile each program with `cealc` and record both the compilation time and the size of the output binary. For comparison purposes, we also compile each program directly with `gcc` (i.e., without `cealc`) by treating CEAL primitives as ordinary functions with external definitions. Table 3 shows the results of the comparison. As can be seen, `cealc` is 3–8 times slower than `gcc` and creates binaries that are 2–5 times larger.

In practice, we observe that the size of core functions can be bounded by a moderate constant. Thus the maximum number of live variables, which is an intra-procedural property, is also bounded by a constant. Based on Theorem 5, we therefore expect the compiled binaries to be no more than a constant factor larger than the source programs. Our experiments show that this constant to be between 2 and 5 for the considered programs.

Theorem 5 implies that the compilation time can bounded by the size of the program plus the time for live-variable analysis. Since

		cealc		gcc	
Program	Lines	Time	Size	Time	Size
Expression trees	422	0.84	74K	0.34	58K
List primitives	553	1.87	109K	0.49	61K
Mergesort	621	2.25	123K	0.54	62K
Quicksort	622	2.22	123K	0.54	62K
Quickhull	988	3.81	176K	0.72	66K
Tree contraction	1918	8.16	338K	1.03	76K
Test Driver	4229	13.69	493K	2.61	110K

Table 3. Compilation times (in seconds) and binary sizes (in bytes) for some CEAL programs. All compiled with `-O0`.

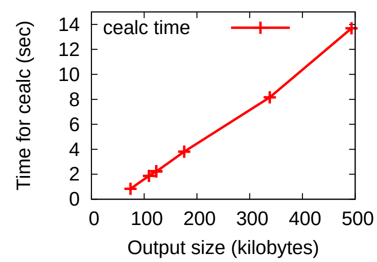

Figure 15. Time for `cealc` versus size of binary output.

our implementation performs live variables analysis and constructs dominator trees on a per-function basis (Section 7), and since the sizes of core functions are typically bounded by a moderate constant, these require linear time in the size of the program. We therefore expect to see the compilation times to be linear in the size of the generated code. Indeed Figure 15 shows that the `cealc` compilation times increase nearly linearly with size of the compiled binaries.

9. Related Work

We discuss the most closely related work in the rest of the paper. In this section, we mention some other work that is related more peripherally.

Incremental and Self-Adjusting Computation. The problem of developing techniques to enable computations respond to incremental changes to their output have been studied since the early 80's. We refer the reader to the survey by Ramalingam and Reps [34] and a recent paper [27] for a more detailed set of references. Effective early approaches to incremental computation either use dependence graphs [16] or memoization (e.g., [33, 2]). Self-adjusting computation generalizes dependence graphs techniques by introducing dynamic dependence graphs, which enables a change propagation algorithm update the structure of the computation based on data modifications, and combining them with a form of computation memoization that permits imperative updates to memory [3].

Dominators. The dominator relation has common use in compilers that perform program analysis and optimization (e.g., [7, 15, 18]). There are a number of asymptotically efficient algorithms for computing dominators (e.g., [26, 19]). In practice simple but asymptotically inefficient algorithms also perform reasonably well [14]. Our implementation uses the simple algorithm described in many compiler books, e.g., [30].

Tail Calls. We use a selective trampolining to support tail calls efficiently (Section 6). Several other proposals to supporting tail calls in C exists (e.g., [36, 23, 9, 20]). Peyton Jones summarizes some of these techniques [24] and discusses the tradeoffs. The primary advantage of trampolining is that it is fully portable; the disadvantage is its cost, which our compiler reduces by piggy-backing closure creation with those of the reads.

10. Discussions

We discuss some limitations of our approach and propose future research directions.

Syntax and Types for Modifiables. The modifiable primitives `read` and `write` assume that the type of a modifiable's contents is `void*`. As a result, their uses sometimes require explicit type coercions (e.g., as in Figure 2). Furthermore, the primitives have a function-like syntax, rather than the familiar C-like syntax for dereferencing (reading) and assignment (writing).

It may be possible to support more conventional C syntax and avoid type coercions by generalizing the notion of a modifiable reference to that of a "modifiable field". In this approach, the programmer would annotate the fields of `struct`s that are subject to change across core updates with a new keyword (e.g., `mod`) that indicates that the field is modifiable. Accessing (reading) and assigning to (writing) these fields would use the same syntax as conventional C `struct`s, which would make the use of modifiable primitives implicit. Just as conventional fields carry type information, each modifiable field would also carry a type that could be used to ensure that its uses are well-typed. This approach would generalize modifiable references since they could be easily encoded as `struct` with a single modifiable field.

Automatic Minimization of Read Bodies. Conceptually, each read operation in CEAL has an implicit "read body" which consists of the code that uses the read value. Our normalization algorithm approximates each read body conservatively by assuming that it extends to the end of the function containing the associated read operation. In general, however, a read value may be used in only a small portion of the body found by normalization. In these cases it's often advantageous (though not always necessary) to refactor the core program so that each read body identified by normalization is minimal, i.e., it contains no code that is independent from the read value. In the proposed approach the programmer can perform this refactoring by hand, but we expect that a compiler mechanism could handle some cases for refactoring automatically by employing further data- and control-flow analysis.

Support for Return Values. Currently, core functions in CEAL cannot return values as functions can in C, but instead use *destination-passing style* (DPS) to communicate results through modifiables. In DPS, the caller provides one or more destinations (modifiables) to the callee who writes these destinations with its results before returning control to the caller; to access the results, the caller reads the value of each destination (e.g., in Figure 2). By restricting core programs to DPS, we ensure that the caller/callee data-dependencies can be correctly tracked via modifiable references and their associated operations.

Although DPS simplifies tracking data dependencies in CEAL, it also forces the programmer to essentially perform a DPS conversion by hand. Moreover, the DPS restriction can be viewed as overly conservative: if the return value of a core function is not affected by changes to modifiables then the use of this value need not be tracked via modifiable references, i.e., it can be returned directly to the caller. We expect that future work on CEAL can add support conventional C-style return values by adding an automatic DPS conversion to the CEAL compiler that acts selectively: when a return value is subject to change across modifiable updates the compiler should automatically DPS convert this function (and its call sites), otherwise, the function can return a value as usual, without any special treatment.

Optimizations. Although the proposed approach is faster and requires less memory than previous approaches, we expect that future optimizations will offer additional time and space savings.

As an example, it may be possible to reduce the overhead of tracing the core primitives (e.g., `read`) when they are used in consecutive sequences (e.g., Figure 2 contains a pair of consecutive reads). The compiler could detect such uses and allow the runtime to trace them as a group rather than individually, thereby reducing their individual costs. In particular, since consecutive reads usually store similar closures in the trace (i.e., closures with one or more common values), tracing such reads using a single closure would offer a significant time and space savings.

In addition to reducing the tracing overhead, future work can explore more efficient ways of supporting tail calls in normalized programs. The proposed approach uses trampolines since they are simple to implement in a portal manner. However, each "bounce" on a trampoline requires at least a function-return (passing control to the trampoline) and an indirect call (to the function contained in the trampolined closure). By contrast, a "real" tail call is little more than an unconditional jump [36, 24].

Intermediate Representation. Our core language CL can be thought of as a simplified source language for self-adjusting computation. As we show, this language is sufficient to implement and reason about the proposed compilation techniques. However, we expect that both implementing and reasoning about future analyses, transformations and optimizations (e.g., those proposed in this section) will be simplified by translating into an intermediate language that is either based on static single assignment (SSA) form [15], or a suitable alternative [10]. Furthermore, since SSA form shares a close relationship to functional programming [8, 25], we expect that using it as an intermediate language for CEAL will allow future work on CEAL to be more directly applicable to the self-adjusting languages that extend existing functional languages (e.g., SaSML [27]).

11. Conclusion

We describe a C-based language for writing self-adjusting programs in a style similar to conventional C programs, present compilation strategies for the language, and describe & evaluate our implementation. This is the first result in making self-adjusting computation work in its full generality with a (low-level) language that does not support higher-order features or offer automatic memory management. Our experiments show that the proposed approach is efficient in practice and significantly improves performance over previous approaches.

Acknowledgements

We thank Matthew Fluet for many helpful discussions and suggestions, and the anonymous reviewers for their valuable feedback.

References

[1] MLton. http://mlton.org/.

[2] Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and Caching of Dependencies. In *Proceedings of the International Conference on Functional Programming*, pages 83–91, 1996.

[3] Umut A. Acar. Self-adjusting computation (an overview). In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2009.

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

[5] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, September 2008.

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

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998.

[9] Henry G. Baker. Cons should not cons its arguments, part II: Cheney on the MTA. *SIGPLAN Not.*, 30(9):17–20, 1995.

[10] J. A. Bergstra, T. B. Dinesh, and J. Heering. A complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems*, 19:639–684, 1996.

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

[12] Magnus Carlsson. Monads for Incremental Computing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming*, pages 26–35. ACM Press, 2002.

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

[14] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm.

[15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[16] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental Evaluation of Attribute Grammars with Application to Syntax-directed Editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

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

[18] Matthew Fluet and Stephen Weeks. Contification using dominators. In *Proceedings of the International Conference on Functional Programming*, pages 2–13, 2001.

[19] Loukas Georgiadis and Robert E. Tarjan. Finding dominators revisited: extended abstract. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 869–878, 2004.

[20] Jr. Guy L. Steele. Rabbit: A compiler for scheme. Technical report, Cambridge, MA, USA, 1978.

[21] Matthew A. Hammer and Umut A. Acar. Memory management for self-adjusting computation. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 51–60, 2008.

[22] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: A C-based language for self-adjusting computation. Technical Report TTIC-TR-2009-2, Toyota Technological Institute, 2009.

[23] Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2:127–202, 1992.

[24] Simon Peyton Jones. C- -: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*. Springer Verlag, 1998.

[25] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pages 13–22, New York, NY, USA, 1995. ACM.

[26] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

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

[28] Gary L. Miller and John H. Reif. Parallel tree contraction, part I: Fundamentals. *Advances in Computing Research*, 5:47–72, 1989.

[29] Gary L. Miller and John H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[30] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[31] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley 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] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[33] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.

[34] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 502–510, 1993.

[35] Ajeet Shankar and Rastislav Bodik. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming language Design and Implementation*, 2007.

[36] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, 1992.

[37] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.