Compiling Self-Adjusting Programs with Continuations

Ruy Ley-Wild *
Carnegie Mellon University
rleywild@cs.cmu.edu

Matthew Fluet Umut A. Acar †
Toyota Technological Institute at Chicago
{fluet,acar}@tti-c.org

Abstract

Self-adjusting programs respond automatically and efficiently to input changes by tracking the dynamic data dependences of the computation and incrementally updating the output as needed. In order to identify data dependences, previously proposed approaches require the user to make use of a set of monadic primitives. Rewriting an ordinary program into a self-adjusting program with these primitives, however, can be difficult and error-prone due to various monadic and proper-usage restrictions, some of which cannot be enforced statically. Previous work therefore suggests that self-adjusting computation would benefit from direct language and compiler support.

In this paper, we propose a language-based technique for writing and compiling self-adjusting programs from ordinary programs. To compile self-adjusting programs, we use a continuation-passing style (cps) transformation to automatically infer a conservative approximation of the dynamic data dependences. To prevent the inferred, approximate dependences from degrading the performance of change propagation, we generate memoized versions of cps functions that can reuse previous work even when they are invoked with different continuations. The approach offers a natural programming style that requires minimal changes to existing code, while statically enforcing the invariants required by self-adjusting computation.

We validate the feasibility of our proposal by extending Standard ML and by integrating the transformation into MLton, a whole-program optimizing compiler for Standard ML. Our experiments indicate that the proposed compilation technique can produce self-adjusting programs whose performance is consistent with the asymptotic bounds and experimental results obtained via manual rewriting (up to a constant factor).

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

General Terms Languages

Keywords self-adjusting computation, continuation-passing style, memoization

* This author was partially supported by a Bell Labs Graduate Fellowship and NSF Grant 0429505.

† This author was partially supported by a gift from Intel.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

1. Introduction

Self-adjusting programs respond automatically and efficiently to input changes by tracking dependencies between data and code in the computation and incrementally updating the output (Acar et al. 2006b). After a from-scratch run, the input can be changed and the output can be updated via *change propagation*, a mechanism for re-executing the portions of the computation affected by the new values while reusing the unaffected portions. Previous work developed techniques for performing change propagation in time proportional to the affected portions and showed that the approach is effective in a number of application domains, including invariant checking (Shankar and Bodik 2007), motion simulation (Acar et al. 2006c, 2008b), and machine learning (Acar et al. 2007, 2008c).

An ordinary program can be converted into a self-adjusting version by manually integrating the change-propagation mechanism into the program. Since this can be very difficult, previous work proposed systematic techniques for rewriting ordinary programs into self-adjusting versions. These rewriting techniques rely on a library of monadic primitives that enable creating, reading, and writing *changeable data*, i.e., data that can change across runs. The libraries expect the programmer to obey certain monadic restrictions, delimit the scope of reads, program in a destination-passing style, and apply memoization by manually declaring all free variables of memoized functions. Consequently, rewriting a program into a self-adjusting program can require substantial restructuring of the existing code (Figure 1 (right) shows a function for partitioning a list with a predicate written using these monadic primitives). Furthermore, safety of memoization primitives is not checked statically. In fact, after some attempts at specifying a simple, safe, systematic interface through library support, previous work points out that direct language and compiler support is essential for writing self-adjusting programs, but leaves the nature of such language and compiler support unspecified (Acar et al. 2006a).

In this paper, we propose a technique for annotating an ordinary program with a small number of primitives and compiling them into equivalent self-adjusting versions. The annotations serve to identify changeable data and memoizing functions without code restructuring. Changeable data is indicated by the type α `box` and can be created and accessed by simple `put` and `get` primitives. Memoizing functions are declared with the `mfun` keyword. We compile an annotated source program by applying an *adaptive continuation-passing-style translation* that infers the dependencies between changeable data.

By performing a cps translation, we enable the programmer to annotate an existing direct-style program with no restructuring. There is a cost to this simplicity: the translation uses continuations to approximate the programmer-supplied, fine-grained dependence information made explicit in the monadic approach. Since a continuation represents the entire rest of a computation, the approach can cause change propagation to re-execute code unnecessarily—continuations are coarse approximations of actual dependencies. To

```
datatype 'a list =
 nil | cons of 'a * 'a list
fun partition p l = let
 fun part l =
  case l of
    nil => (nil,nil)
  | cons (h,t) => let
    val (a,b) = part t
    in
     if p h then
      (cons(h,a),b)
     else
      (a,cons(h,b))
    end
 in part l end
```

```
datatype 'a list =
 nil | cons of 'a * 'a list box
afun partition p l = let
 mfun part l =
  case get $ l of
    nil => (put $ nil,put $ nil)
  | cons(h,t) => let
    val (a,b) = part $ t
    in
     if p h then
      (put $ (cons(h,a)),b)
     else
      (a,put $ (cons(h,b)))
    end
 in part $ l end
```

```
datatype 'a list =
 nil | cons of 'a * 'a list mod
fun partition p l = let
 fun part l = read(l, fn l =>
  case l of
    nil => write(mod(write(nil),mod(write(nil))
  | cons(h,t) => memo (h,t) (fn () => let
    val ab = mod(part t)
    in read(ab, fn (a,b) =>
     if p h then
      write(mod(write(cons(h,a)),b)
     else
      write(a,mod(write(cons(h,b))))
    end))
 in part l end
```

Figure 1. The partition function: ordinary (left), self-adjusting (center), and with previously proposed monadic interface (right).

regain efficiency, the translation produces memoized cps functions as well as memoized continuations. Memoizing a cps function directly on its data and continuation arguments does not suffice because it prevents the result of a function call from being reused when the continuation differs (even if the data arguments are the same). We solve this problem by treating continuations themselves as changeable data. When a memoized cps function encounters previously-seen data but a different continuation, it can immediately pass the memoized result to the (new) continuation without having to re-execute the body of the function. We formalize the compilation as a translation (Section 5) from an annotated source language ASrc (Section 3) with direct-style primitives to a self-adjusting target language SATgt (Section 4) with cps primitives.

We show that the proposed approach is realistic by extending the Standard ML language and modifying the MLton compiler (MLt) to compile self-adjusting programs (Section 6). Our implementation includes a library for self-adjusting computation providing the features of SATgt. We perform an experimental analysis by compiling self-adjusting versions of a number of (annotated) benchmarks. Our experiments (Section 7) indicate that the compiled self-adjusting programs can be slower by a constant factor than their non-self-adjusting counterparts when run from scratch. When responding to input changes, however, self-adjusting programs can be orders of magnitude faster than recomputing from scratch (as compared to the non-adaptive version). The experimental indicate that the approach performs consistently with the previous evaluation of self-adjusting computation based on monadic, user-level libraries (Acar et al. 2006b,a).

2. Overview

We give an overview of our approach by considering an example.

2.1 Self-adjusting programs

A self-adjusting program is a pure program that manipulates *changeable* data, i.e., data that can be changed by external factors. In typical usage, a host *mutator* program contains a self-adjusting subprogram. The host mutator creates the initial changeable input data, runs a self-adjusting program, and observes the output. Then, it can change the input data (via side-effecting operations) and force *change propagation* to update the output of the self-adjusting program. To efficiently update the output, change propagation combines *adaptivity*, a mechanism for re-executing the portions of the computation affected by input changes, and *memoization*, a mechanism for reusing the unaffected portions of the computation.

To express self-adjusting programs, we use an extended SML syntax. A self-adjusting program consists of normal (pure, non-adaptive) functions and *adaptive functions* declared with the `afun` and `mfun` keywords; the latter declares a *memoizing* adaptive function. Adaptive functions have the adaptive function type τ `-$>` `$\tau$. The infix `$` keyword is used for adaptive application; an adaptive application may only appear in the body of an adaptive function (and may not appear in the body of a normal function).

The *box type* τ `box` serves as a container for changeable data. The `put:` α `-$>` α `box` primitive places a value into a box, while the `get:` α `box -$>` α primitive returns the contents of a box. Since the primitives have adaptive function types, they may only be used within a self-adjusting computation. The host mutator may create, modify, and inspect changeable data via a collection of meta-level primitives, which we treat informally in this section.

The distinction between adaptive functions and normal functions serves both language design and implementation purposes. From the design perspective, the distinction prevents self-adjusting-computation primitives from being used outside of a self-adjusting computation. From the implementation perspective, the distinction improves the efficiency of our compilation strategy and the resulting self-adjusting programs. In particular, only the adaptive functions need to be compiled into continuation-passing style.

2.2 Writing self-adjusting programs

Figure 1 shows the ordinary (left) and self-adjusting (center) versions of a `partition` function for lists. In the ordinary version, lists are defined by the usual recursive datatype and the function traverses the list and constructs the output from tail to head, applying the predicate to each element of the list.

We obtain the self-adjusting version in two steps. First, we change the list type so that a list tail is boxed. This allows the mutator to modify lists by inserting/deleting elements. Second, we change the `partition` function to operate on boxed lists by inserting a `get` operation when destructing a list and inserting a `put` operation when constructing a list. Since the auxiliary function `part` is recursive, we memoize it by declaring it with `mfun`. Note that the self-adjusting syntax and primitives (underlined) do not require significant changes to the code: simply deleting them yields the ordinary implementation of `partition`. For the purposes of comparison, the right of Figure 1 shows the code for `partition` with the previously provided monadic primitives (Acar et al. 2006b,a). As can be seen, the monadic primitives require significant changes, even for this simple function. (The significance of the changes is

```
1  datatype 'a list = nil | cons of 'a * 'a list mod

2  fun partition p ml k = let
3    fun part (ml, k) = read ml (fn l =>
4      case l of
5        nil => write nil (fn ma =>
6               write nil (fn mb => k (ma, mb)))
7      | cons(h,mt) => let
8        val k' = fn (a,b) =>
9          if p h then
10           write (cons(h,a)) (fn ma => k (ma,b))
11         else
12           write (cons(h,b)) (fn mb => k (a,mb))
13       in part_memo mt k' end)
14   and part_memo ml k = let
15     val k_memo = fn r => memo k r
16     in
17       write k_memo (fn mk => let
18             val k' = fn r => read mk (fn k => k r)
19             in memo part (ml, k') end)
20     end
21   in part_memo ml k end
```

Figure 2. The partition function compiled.

best measured by considering the differences in the abstract syntax trees, not the differences in the lexical tokens.)

2.3 Compiling self-adjusting programs

Compilation translates a source self-adjusting program into an intermediate language that generalizes the previously proposed monadic primitives. In this section we use a simplified intermediate language that provides two cps primitives on *modifiable references*: `write: α -> (α mod -> res) -> res` and `read: α mod -> (α -> res) -> res`, where `res` is an abstract result type. The `write` primitive initializes a new modifiable with a value and passes the reference to the continuation.[1] The `read` primitive dereferences a modifiable and passes the contents to the continuation.

To compile a source self-adjusting program, we translate adaptive functions into equivalent cps functions and memoize them if so indicated (via keyword `mfun`). We compile boxes into *modifiable references* by translating each `box` type to a `mod` type, each `get` primitive to a `read` primitive, and each `put` primitive to a `write` primitive. For `read` and `write`, we supply the current continuation.

Memoizing functions in cps requires some care. To see this, recall that to use a function call via memoization, the current arguments must match the arguments of a previous call. Since the arguments of a cps function include its continuation, memoization would require the continuations to match. This decreases the effectiveness of memoization because we may not match a function call when the continuations differ.[2] We address this problem by translating memoized adaptive functions to cps functions that treat their continuations as changeable data. This allows the memoized function to match when the modifiable (containing the continuation) matches a previous call, ignoring the contents of the modifiable. Since the continuation is changeable data, if it differs in the current run from the previous run, then change propagation will re-execute any invocation of the continuation, but without having to re-execute the body of the matched function. We memoize functions and continuations with a primitive `memo: (α -> res) -> α -> res`.

Figure 2 shows the compiled code for `partition`. To obtain this code, we translate the functions `partition` and `part` and adaptive applications into cps, replace `put/get` with `write/read`, and memoize `part` as `part_memo`. To do so, `part_memo` memoizes its continuation and writes it into a modifiable. It then calls `part` with a continuation that, when invoked, reads and invokes the original continuation. Since the application of `part` is memoized, it will match when it is called with the same modifiable list and the continuation `k` is written into the same modifiable. (This can be ensured by "remembering" the continuation modifiable chosen for each argument modifiable list.) We describe how the compiled program achieves efficient change propagation in Section 5.1.

3. Adaptive Source Language (ASrc)

The ASrc language is a simply-typed λ-calculus with natural numbers and recursive functions[3], extended with a distinguished class of *adaptive* functions and *boxing* primitives. The language does not directly yield self-adjusting programs: its semantics is analogous to that of a call-by-value λ-calculus, but the additional forms are used by the compilation scheme of Section 5 to yield an equivalent self-adjusting program.

The syntax of ASrc is given by the following grammar, which defines types τ, expressions e, and values v, using identifier metavariables f and x.

$$
\begin{array}{rcl}
\tau & ::= & \mathbf{nat} \mid \tau_x \to \tau \mid \tau_x \xrightarrow{\$} \tau \mid \tau\ \mathbf{box} \\
e & ::= & \mathbf{zero} \mid \mathbf{succ}\, e \mid \mathbf{caseN}\, e_n\, e_z\, (x.e_s) \mid \\
 & & x \mid \mathbf{fun}\, f.x.e \mid e_f\, e_x \mid \\
 & & \mathbf{afun}\, f.x.e \mid \mathbf{mfun}\, f.x.e \mid e_f\, \$\, e_x \mid \\
 & & \mathbf{put}\, \$\, e \mid \mathbf{get}\, \$\, e \mid \ell \\
v & ::= & \mathbf{zero} \mid \mathbf{succ}\, v \mid x \mid \mathbf{fun}\, f.x.e \mid \\
 & & \mathbf{afun}\, f.x.e \mid \mathbf{mfun}\, f.x.e \mid \ell \\
\lambda x.e & \stackrel{\text{def}}{=} & \mathbf{fun}\, f.x.e \qquad \text{with } f \notin \mathrm{FV}(e)
\end{array}
$$

As noted above, we distinguish normal (non-adaptive) functions $\tau_x \to \tau$ from adaptive functions $\tau_x \xrightarrow{\$} \tau$. Correspondingly, there are separate introduction and elimination forms for the two classes of functions; an **mfun** adaptive function indicates that the function should use memoization when compiled. Boxed types $\tau\ \mathbf{box}$ act like immutable references, with the primitives **put** and **get** acting like allocation and dereference. Hence, we take a store σ to be a finite map from locations ℓ to values; the notation $\sigma[\ell \mapsto v]$ denotes the store σ updated with ℓ mapped to v. Since stores are immutable, they are also acyclic. Contexts Γ and Σ are maps from variables and locations to types, respectively.

Figure 3 gives the static and dynamic semantics of ASrc: the typing judgement $\Sigma; \Gamma \vdash^{\delta} e : \tau$ ascribes the type τ to the expression e at the mode δ (either normal $\bullet$ or adaptive \$) and in the contexts Γ and Σ, while the large-step evaluation relation $\sigma; e \Downarrow \sigma'; v'$ reduces the expression e in the store σ to the value v' and the updated store σ'.

The mode component of the typing judgement precludes adaptive applications and the boxing primitives from the body of normal functions, but allows normal applications, adaptive applications, and the boxing primitives in the body of adaptive functions. As noted earlier, in the context of our implementation, these requirements prevent self-adjusting primitives from being used outside of a self-adjusting computation. While these requirements could be

[1] The write primitive can reuse modifiables written in the previous run of the program. This is essential for efficient change propagation.

[2] Note that, for self-adjusting programs, memoization during change propagation attempts to match function calls from the previous run of the program. There is no attempt to match calls within a run of the program.

[3] The ASrc language (as well as the SATgt language of Section 4 and the translation from the former to the latter) may easily be extended with products, sums, recursive types, etc.; we have omitted such constructs as they provide no additional insight, but are supported by the implementation.

$$\frac{}{\Sigma;\Gamma \vdash^{\delta} \mathbf{zero} : \mathbf{nat}} \qquad \frac{\Sigma;\Gamma \vdash^{\delta} e : \mathbf{nat}}{\Sigma;\Gamma \vdash^{\delta} \mathbf{succ}\, e : \mathbf{nat}} \qquad \frac{\Sigma;\Gamma \vdash^{\delta} e_n : \mathbf{nat} \quad \Sigma;\Gamma \vdash^{\delta} e_z : \tau \quad \Sigma;\Gamma, x{:}\mathbf{nat} \vdash^{\delta} e_s : \tau}{\Sigma;\Gamma \vdash^{\delta} \mathbf{caseN}\, e_n\, e_z\, (x.e_s) : \tau}$$

$$\frac{x{:}\tau \in \Gamma}{\Sigma;\Gamma \vdash^{\delta} x : \tau} \qquad \frac{\Sigma;\Gamma, f{:}\tau_x \to \tau, x{:}\tau_x \vdash^{\bullet} e : \tau}{\Sigma;\Gamma \vdash^{\delta} \mathbf{fun}\, f.x.e : \tau_x \to \tau} \qquad \frac{\Sigma;\Gamma \vdash^{\delta} e_f : \tau_x \to \tau \quad \Sigma;\Gamma \vdash^{\delta} e_x : \tau_x}{\Sigma;\Gamma \vdash^{\delta} e_f\, e_x : \tau}$$

$$\frac{\Sigma;\Gamma, f{:}\tau_x \xrightarrow{\$} \tau, x{:}\tau_x \vdash^{\$} e : \tau}{\Sigma;\Gamma \vdash^{\delta} \mathbf{afun}\, f.x.e : \tau_x \xrightarrow{\$} \tau} \qquad \frac{\Sigma;\Gamma, f{:}\tau_x \xrightarrow{\$} \tau, x{:}\tau_x \vdash^{\$} e : \tau}{\Sigma;\Gamma \vdash^{\delta} \mathbf{mfun}\, f.x.e : \tau_x \xrightarrow{\$} \tau} \qquad \frac{\Sigma;\Gamma \vdash^{\$} e_x : \tau_x \xrightarrow{\$} \tau \quad \Sigma;\Gamma \vdash^{\$} e_f : \tau_x}{\Sigma;\Gamma \vdash^{\$} e_f\, \$\, e_x : \tau}$$

$$\frac{\Sigma;\Gamma \vdash^{\$} e : \tau}{\Sigma;\Gamma \vdash^{\$} \mathbf{put}\, \$\, e : \tau\ \mathbf{box}} \qquad \frac{\Sigma;\Gamma \vdash^{\$} e : \tau\ \mathbf{box}}{\Sigma;\Gamma \vdash^{\$} \mathbf{get}\, \$\, e : \tau} \qquad \frac{l{:}\tau \in \Sigma}{\Sigma;\Gamma \vdash^{\delta} l : \tau\ \mathbf{box}}$$

$$\frac{}{\sigma; v \Downarrow \sigma; v} \qquad \frac{\sigma; e \Downarrow \sigma'; v'}{\sigma; \mathbf{succ}\, e \Downarrow \sigma'; \mathbf{succ}\, v'} \qquad \frac{\sigma; e_n \Downarrow \sigma''; \mathbf{zero} \quad \sigma''; e_z \Downarrow \sigma'; v'}{\sigma; \mathbf{caseN}\, e_n\, e_z\, (x.e_s) \Downarrow \sigma'; v'} \qquad \frac{\sigma; e_n \Downarrow \sigma''; \mathbf{succ}\, v'' \quad \sigma''; [v''/x]\, e_s \Downarrow \sigma'; v'}{\sigma; \mathbf{caseN}\, e_n\, e_z\, (x.e_s) \Downarrow \sigma'; v'}$$

$$\frac{\sigma; e_f \Downarrow \sigma_1; \mathbf{fun}\, f.x.e \quad \sigma_1; e_x \Downarrow \sigma_2; v_x \quad \sigma_2; [\mathbf{fun}\, f.x.e/f, v_x/x]\, e \Downarrow \sigma'; v'}{\sigma; e_f\, e_x \Downarrow \sigma'; v'} \qquad \frac{\sigma; e_f \Downarrow \sigma_1; \mathbf{afun}\, f.x.e \quad \sigma_1; e_x \Downarrow \sigma_2; v_x \quad \sigma_2; [\mathbf{afun}\, f.x.e/f, v_x/x]\, e \Downarrow \sigma'; v'}{\sigma; e_f\, \$\, e_x \Downarrow \sigma'; v'}$$

$$\frac{\sigma; e_f \Downarrow \sigma_1; \mathbf{mfun}\, f.x.e \quad \sigma_1; e_x \Downarrow \sigma_2; v_x \quad \sigma_2; [\mathbf{mfun}\, f.x.e/f, v_x/x]\, e \Downarrow \sigma'; v'}{\sigma; e_f\, \$\, e_x \Downarrow \sigma'; v'}$$

$$\frac{\sigma; e \Downarrow \sigma'; v' \quad \ell' \notin \mathrm{dom}\, \sigma'}{\sigma; \mathbf{put}\, \$\, e \Downarrow \sigma'[\ell \mapsto v']; \ell'} \qquad \frac{\sigma; e \Downarrow \sigma'; \ell \quad \sigma'(\ell) = v'}{\sigma; \mathbf{get}\, \$\, e \Downarrow \sigma'; v'}$$

Figure 3. ASrc typing judgement $\Sigma;\Gamma \vdash^{\delta} e : \tau$ (top) and evaluation judgement $\sigma; e \Downarrow \sigma'; v'$ (bottom).

expressed by a more complicated set of expression subgrammars, expressing them using a mode component of the typing judgement scales easily to additional language features and is more consistent with our implementation. The normal vs. adaptive distinction may also be interpreted as a simple effect system (Henglein et al. 2005) that syntactically distinguishes effectful and non-effectful functions and applications.

Evaluation presupposes that neither the initial expression nor store have free variables, but the initial expression *may* have free locations that are present in the initial store; these locations represent the program's (changeable) input.

Finally, note that boxed types $\tau\ \mathbf{box}$ and the primitives **put** and **get** contribute no computational power to the language. However, boxing an expression indicates that the corresponding translated expression should write the result into a modifiable reference; any subsequent uses of the result must read from the modifiable, making data dependencies explicit.

Although the ASrc language does not provide any facilities for creating and modifying the inputs to a self-adjusting computation, we can sketch the actions and queries made by a host program that (re-)evaluates a self-adjusting computation. (These *meta operations* are discussed in Section 6.) Suppose we have a ASrc program e such that $\Sigma;\cdot \vdash^{\$} e : \tau$ and an initial store σ_0 such that $\vdash^{\bullet} \sigma_0 : \Sigma \uplus \Sigma_0$ (for the obvious store typing judgement). Thus, we may (initially) evaluate e under the store σ_0, yielding the (initial) result v_0': $\sigma_0; e \Downarrow \sigma_0'; v_0'$. Now, suppose we have a modified store σ_1 such that $\vdash^{\bullet} \sigma_1 : \Sigma \uplus \Sigma_1$. (This modified store may have been obtained from the initial store by changing contents of some locations, inserting and deleting locations, *etc*.) We are interested in the result v_1' yielded by (re-)evaluating e under σ_1. The next section describes a language with a change-propagation relation that reuses portions of the computation that evaluated e under σ_0 to yield v_1' more efficiently than using the evaluation relation $\sigma_1; e \Downarrow \sigma_1'; v_1'$.

4. Self-Adjusting Target Language (SATgt)

The SATgt language is a simply-typed λ-calculus with natural numbers and recursive functions, extended with *modifiable references* and a *memoization* primitive. The language directly yields self-adjusting programs: its semantics includes both an evaluation relation and a change propagation relation. Section 5 shows how ASrc programs are compiled into SATgt programs by a cps transformation that uses ASrc annotations to insert primitives for self-adjusting computation.

The syntax of SATgt is given by the following grammar, which defines types τ, expressions e, values v, and adaptive commands κ, using identifier metavariables f and x.

$$\begin{array}{rcll} \tau & ::= & \mathbf{nat} \mid \tau_x \to \tau \mid \tau\ \mathbf{mod} \mid \mathbf{res} \\ e & ::= & \mathbf{zero} \mid \mathbf{succ}\, e \mid \mathbf{caseN}\, e_n\, e_z\, (x.e_s) \mid \\ & & x \mid \mathbf{fun}\, f.x.e \mid e_f\, e_x \mid \\ & & \mathbf{write}\, v\, v_k \mid \mathbf{read}\, v\, v_k \mid \ell \mid \mathbf{memo}\, e \mid \mathbf{halt}\, v \\ v & ::= & \mathbf{zero} \mid \mathbf{succ}\, v \mid x \mid \mathbf{fun}\, f.x.e \mid \ell \mid \kappa \\ \kappa & ::= & \mathbf{write}\, v\, v_k \mid \mathbf{read}\, v\, v_k \mid \mathbf{memo}\, e \mid \mathbf{halt}\, v \\ \lambda x.e & \stackrel{\mathrm{def}}{=} & \mathbf{fun}\, f.x.e \qquad \text{with } f \notin \mathrm{FV}(e) \end{array}$$

Modifiables $\tau\ \mathbf{mod}$ act like immutable references, with the primitives **write** and **read** acting like allocation and dereference. Note that both primitives are formulated in continuation-passing style, with v_k serving as the continuation of the operation. The type **res** is an opaque answer type for continuations, while **halt** is a continuation that injects a final value into the **res** type. As before, we take a store σ to be a finite map of locations to values. Contexts Γ and Σ are maps from variables and locations to types, respectively.

Figure 4 gives the static and dynamic semantics of SATgt. The typing judgement $\Sigma;\Gamma \vdash e : \tau$ ascribes the type τ to the expression e in the contexts Γ and Σ. The large-step evaluation relation $\dot{T}; \sigma; e \Downarrow_E T'; \sigma'; v'$ (resp. $\dot{T}; \sigma; \kappa \Downarrow_K T'; \sigma'; v'$) reduces the expression e (resp. the adaptive command κ) under the store σ to the value v' and the updated store σ'. For the present time, we suggest that the reader ignore the $\dot{T}$ and T' components; we discuss them in detail in Section 4.1. The auxiliary evaluation relation $e \Downarrow v'$ reduces an expression e to a value v'; such evaluation is pure and

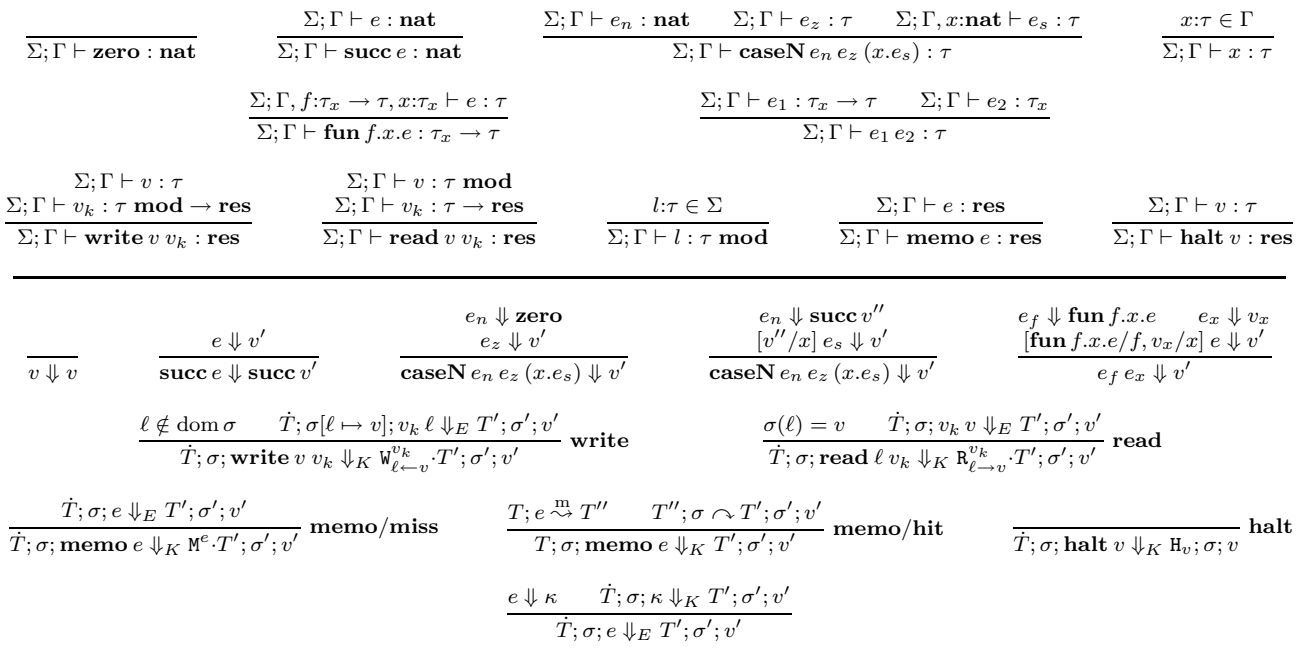


Figure 4. SATgt typing rule $\Sigma; \Gamma \vdash e : \tau$ (top) and evaluation relations $e \Downarrow v'$ and $\dot{T}; \sigma; \kappa \Downarrow_K T'; \sigma'; v'$ and $\dot{T}; \sigma; e \Downarrow_E T'; \sigma'; v'$ (bottom).

independent of the store. The three evaluation relations model the execution of a self-adjusting program as the interleaving of pure computations and adaptive commands. Note that evaluation presupposes that neither the initial expression nor store have free variables, but the initial expression *may* have free locations which are present in the initial store; these locations represent the program's (changeable) input.

A $\mathbf{write}\ v\ v_k$ command yields a fresh location ℓ of type $\tau\ \mathbf{mod}$ that is delivered to the continuation v_k and updates the store σ at ℓ with v. A $\mathbf{read}\ \ell\ v_k$ command yields a value v of type τ (fetched from the store σ at ℓ) that is delivered to the continuation v_k.

A memoized expression $\mathbf{memo}\ e$ in SATgt has no special behavior when evaluated from scratch ($\mathbf{memo/miss}$). However, memoization enables the reuse of computations *across runs* during change propagation. This differs from other uses of memoization that permit sharing subcomputations within a single run of a program. The $\mathbf{halt}\ v$ command yields a computation's final result value.

4.1 Change Propagation and Memoization

In order to update a program's output in response to changes in its input, a *change propagation* mechanism is employed to re-execute the portions of the computation affected by the changes and to reuse the unaffected portions. The evaluation relation records information necessary for change propagation in a *trace*, a sequence of write, read, and memo *actions* terminated by a halt action:

$$A ::= \mathtt{W}^{v_k}_{\ell \leftarrow v} \mid \mathtt{R}^{v_k}_{\ell \rightarrow v} \mid \mathtt{M}^{e} \qquad T ::= \mathtt{H}_v \mid A \cdot T \qquad \dot{T} ::= \Box \mid T$$

The evaluation relation $\dot{T}; \sigma; e \Downarrow_E T'; \sigma'; v'$ (resp. $\dot{T}; \sigma; \kappa \Downarrow_K T'; \sigma'; v'$) may now be interpreted as reducing the expression e (resp. the adaptive command κ) under the store σ and the (optional) reuse trace $\dot{T}$, yielding the value v', the updated store σ', and the computation trace T'. A present reuse trace T is itself a computation trace from a previous evaluation and is supplied to change propagation to guide the update; in particular, evaluation may reuse computations memoized in the previous evaluation (the memoization judgement of Figure 5 used in the $\mathbf{memo/hit}$ evaluation rule).

The change propagation relation $T; \sigma \curvearrowright T'; \sigma'; v'$ (given in Figure 5) may be interpreted as replaying the computation trace T under the store σ, yielding the value v', the updated store σ', and the updated computation trace T'.

Returning to the evaluation relation, a read operation $\mathbf{read}\ \ell\ v_k$ dereferences ℓ and extends the computation trace with a read action $\mathtt{R}^{v_k}_{\ell \rightarrow \sigma(\ell)}$ that records the location dereferenced, the value fetched, and the continuation of the read operation; the read action in a computation trace identifies computations that must be re-executed by change propagation. A write operation $\mathbf{write}\ v\ v_k$ allocates a location ℓ and extends the computation trace with a write action $\mathtt{W}^{v_k}_{\ell \leftarrow v}$ that records the location allocated, the value written, and the continuation of the write operation. Note that the choice of location ℓ is independent of the reuse trace $\dot{T}$. It is acceptable (and, indeed, often desirable) for the location ℓ to appear in a write action $\mathtt{W}^{v_k}_{\ell \leftarrow v}$ in the reuse trace; we say that such a location is (implicitly) *stolen* from the reuse trace.

The $\mathbf{memo/miss}$ rule evaluates a memoization expression $\mathbf{memo}\ e$ and yields a trace $\mathtt{M}^{e} \cdot T'$, where T' is the trace of the evaluation of e. The $\mathbf{memo/hit}$ rule uses the memoization judgement $T; e \overset{\mathrm{m}}{\rightsquigarrow} T''$ (Figure 5) to search the reuse trace T for a suffix reuse trace T'' that begins with the memoization action $\mathtt{M}^{e}$. Note that while the expression e may have free locations, the memoization judgement is independent of the store. Hence, the rule switches to change propagating T'' under the current store in order to correct any invalid reads or writes in the reuse trace T''.

Note that a memoization hit ($\mathbf{memo/hit}$ and $\mathbf{m/hit}$) requires the expression being evaluated and the expression in the memo action to be α-equivalent. This equivalence requires the location names appearing in the expressions to be syntactically equal. This, in turn, motivates the implicit stealing of locations by the $\mathbf{write}$ rule: (re-)executing a write, using a location that appears in the reuse trace, may allow a subsequent memoization action to match in the reuse trace.

$$\frac{e_\circ \equiv_\alpha e}{\mathtt{M}^{e_\circ}\cdot T;\, e \overset{\mathrm{m}}{\rightsquigarrow} \mathtt{M}^{e}\cdot T}\ \mathbf{m/hit} \qquad \frac{T;\, e \overset{\mathrm{m}}{\rightsquigarrow} T''}{A\cdot T;\, e \overset{\mathrm{m}}{\rightsquigarrow} T'}\ \mathbf{m/miss}$$

$$\frac{\ell_\circ \notin \mathrm{dom}\,\sigma \qquad T;\sigma[\ell_\circ \mapsto v] \curvearrowright T';\sigma';v'}{\mathtt{W}^{v_k}_{\ell_\circ \leftarrow v}\cdot T;\sigma \curvearrowright \mathtt{W}^{v_k}_{\ell_\circ \leftarrow v}\cdot T';\sigma';v'}\ \mathbf{write/reuse} \qquad \frac{\ell \notin \mathrm{dom}\,\sigma \qquad T;\sigma[\ell \mapsto v];v_k\ \ell \Downarrow T';\sigma';v'}{\mathtt{W}^{v_k}_{\ell_\circ \leftarrow v}\cdot T;\sigma \curvearrowright \mathtt{W}^{v_k}_{\ell \leftarrow v}\cdot T';\sigma';v'}\ \mathbf{write/change}$$

$$\frac{\sigma(\ell) = v \qquad v_\circ \equiv_\alpha v \qquad T;\sigma \curvearrowright T';\sigma';v'}{\mathtt{R}^{v_k}_{\ell \rightarrow v_\circ}\cdot T;\sigma \curvearrowright \mathtt{R}^{v_k}_{\ell \rightarrow v}\cdot T';\sigma';v'}\ \mathbf{read/reuse} \qquad \frac{\sigma(\ell) = v \qquad T;\sigma;v_k\ v \Downarrow T';\sigma';v'}{\mathtt{R}^{v_k}_{\ell \rightarrow v_\circ}\cdot T;\sigma \curvearrowright \mathtt{R}^{v_k}_{\ell \rightarrow v}\cdot T';\sigma';v'}\ \mathbf{read/change}$$

$$\frac{T;\sigma \curvearrowright T';\sigma';v'}{\mathtt{M}^{e}\cdot T;\sigma \curvearrowright \mathtt{M}^{e}\cdot T';\sigma';v'}\ \mathbf{memo} \qquad \frac{}{\mathtt{H}_v;\sigma \curvearrowright \mathtt{H}_v;\sigma;v}\ \mathbf{halt}$$

Figure 5. Memoization $T; e \overset{\mathrm{m}}{\rightsquigarrow} T''$ (top) and change propagation $T;\sigma \curvearrowright T';\sigma';v'$ (bottom).

Turning to the change propagation relation (Figure 5), recall that we interpret $T;\sigma \curvearrowright T';\sigma';v'$ as replaying the computation trace T under the store σ, yielding the value v', the updated store σ', and the updated computation trace T'. A write $\mathtt{W}^{v_k}_{\ell_\circ \leftarrow v}$ that is consistent with the current store (**write/reuse**, requiring that $\ell_\circ \notin \mathrm{dom}\,\sigma$) extends the store with $\ell_\circ$ bound to v and recursively change propagates the tail of the trace. A write that is inconsistent with the current store (if $\ell_\circ \in \mathrm{dom}\,\sigma$) or is nondeterministically chosen to be re-executed (**write/change**) forces the allocation of a fresh location ℓ and re-evaluates the continuation v_k with the location ℓ. A read $\mathtt{R}^{v_k}_{\ell \rightarrow v_\circ}$ that is consistent with the current store (**read/reuse**, requiring $\sigma(\ell) = v \equiv_\alpha v_\circ$) recursively change propagates the tail of the trace. A read that is inconsistent with the current store (if $\sigma(\ell) = v \not\equiv_\alpha v_\circ$) or is nondeterministically chosen to be re-executed (**read/change**) re-evaluates the continuation v_k with the current contents of ℓ. Replaying a memoization action recursively change propagates the tail of the trace. Replaying a halt action yields the (unchanged) computation result. Whenever change propagation is recursively applied, the updated computation trace is extended with an appropriate action.

Note that change propagation copies the prefix of the computation trace up to the first read or write that triggers re-execution. If there were no **memo/hit** evaluation rule, then re-execution would never return to change propagation and the entire tail of the computation would be re-executed by the evaluation judgement, which may be no better (asymptotically) than evaluating from scratch. Hence, memoization is crucial for efficient change propagation.

Although the **write** rule may allocate locations in the reuse trace and the memoization judgement may match computations in the reuse trace, the rules are intentionally nondeterministic to avoid committing to particular allocation and memoization policies. It is possible to consider the rules as being guided by an oracle that decides when to steal locations and when to match memoizations. Since making such choices optimally is undecidable in general, the adaptive library described in Section 6.1.2 provides mechanisms that restrict when locations may be stolen and when memoization may match.

We can now sketch the use of change propagation by a host program that (re-)evaluates a self-adjusting computation. Suppose we have a SATgt program e such that $\Sigma; \cdot \vdash e : \mathbf{res}$ and an initial store σ_0 such that $\vdash \sigma_0 : \Sigma \uplus \Sigma_0$. Thus, we may (initially) evaluate e under the store σ_0 and an empty reuse trace, yielding the (initial) result v_0' and a computation trace T_0': $\square;\sigma_0;e \Downarrow_E T_0';\sigma_0';v_0'$. Now, suppose we have a modified store σ_1 such that $\vdash \sigma_1 : \Sigma \uplus \Sigma_1$. We are interested in the result v_1' yielded by (re-)evaluating e under σ_1. To obtain v_1', we may change propagate the trace T_0' under the store σ_1: $T_0';\sigma_1 \curvearrowright T_1';\sigma_1';v_1'$. The correctness of change propagation asserts that the v_1', σ_1', and T_1' obtained via the change-propagation relation could also have been obtained via the evaluation relation: $\square;\sigma_1;e \Downarrow_E T_1';\sigma_1';v_1'$. Hence, change propagation suffices to determine the output of a program on changed inputs.

5. Compiling ASrc to SATgt

The annotations of ASrc programs are used to guide an *adaptive continuation-passing style* transformation into an equivalent SATgt program. The transformation is a standard cps conversion, with the following notable differences: only adaptive ASrc functions are converted to continuation-passing style, while normal ASrc functions remain in direct style (thus, our transformation is an instance of a selective cps transformation (Danvy and Hatcliff 1993a,b; Nielsen 2001; Kim and Yi 2001)); the ASrc boxing primitives **put** and **get** are converted into explicit **write** and **read** operations; and memoizing adaptive ASrc functions are converted using explicit **memo** operations. Figure 6 shows the type and expression translations. The correctness and efficiency of the translation is captured by the fact that well-typed ASrc programs are compiled into equivalent well-typed SATgt programs with the same asymptotic complexity for initial runs (*i.e.* SATgt evaluation with an empty reuse trace).

The type translation $[\![\tau^{\mathsf{asrc}}]\!] = \tau^{\mathsf{satgt}}$ preserves the **nat** type, recursively translates the normal function type (without introducing continuations), converts the adaptive function type to take a continuation argument, and converts the boxed type to a modifiable type. There are two term translations: $[\![e^{\mathsf{asrc}}]\!]^\bullet = e^{\mathsf{satgt}}$ translates expressions appearing as the body of normal ASrc functions and $[\![e^{\mathsf{asrc}}]\!]^\$\ v_k^{\mathsf{satgt}} = e^{\mathsf{satgt}}$ translates expressions appearing as the body of adaptive ASrc functions (using the SATgt-value v_k^{satgt} as the explicit continuation term); the metavariables y and k are used to distinguish identifiers introduced by the translation. The $[\![e]\!]^\bullet$ translation recursively translates sub-expressions and appropriately translates the body of normal and adaptive functions, introducing continuation arguments for adaptive functions; we explain the translation of **mfun** in more detail below. Note that $[\![e]\!]^\bullet$ is not defined for adaptive applications or the boxing primitives, as these expressions may not appear in the body of a well-typed normal function.

The $[\![e]\!]^\$\ v_k$ translation is a standard cps conversion for constants (**zero** and ℓ), variables (x), adaptive functions ($\mathbf{afun}\ f.x.e$), and adaptive applications ($e_1\ \$\ e_2$). For ASrc expressions translated to SATgt expressions that are evaluated in direct style ($\mathbf{succ}\, e$, $\mathbf{caseN}\ e_n\ e_z\ (x.e_s)$, and $e_1\ e_2$), the translation delivers the result to the continuation v_k. Finally, it translates the boxing primitives

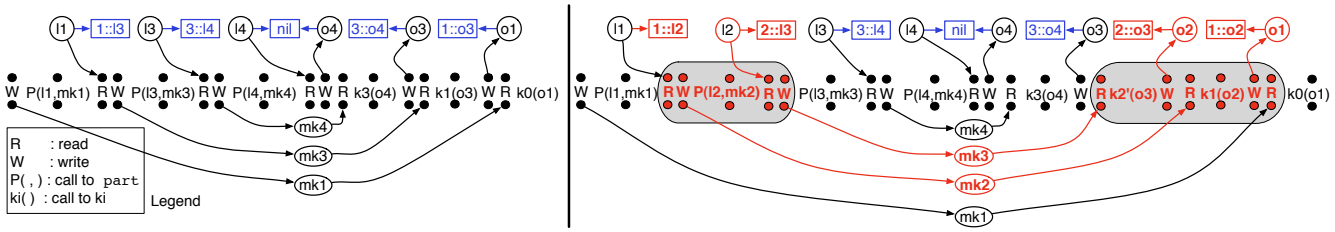


Figure 7. Execution of `partition` on lists `[1,3]` (left) and `[1,2,3]` (right).

$$
\begin{aligned}
[\![\mathbf{nat}]\!] &= \mathbf{nat} \\
[\![\tau_x \to \tau]\!] &= [\![\tau_x]\!] \to [\![\tau]\!] \\
\left[\!\!\left[\tau_x \xrightarrow{\$} \tau\right]\!\!\right] &= [\![\tau_x]\!] \to (([\![\tau]\!] \to \mathbf{res}) \to \mathbf{res}) \\
[\![\tau\ \mathbf{box}]\!] &= [\![\tau]\!]\ \mathbf{mod}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{zero}]\!]^\bullet &= \mathbf{zero} \\
[\![\mathbf{succ}\, e]\!]^\bullet &= \mathbf{succ}\ [\![e]\!]^\bullet \\
[\![\mathbf{caseN}\, e_n\, e_z\, (x.e_s)]\!]^\bullet &= \mathbf{caseN}\ [\![e_n]\!]^\bullet\ [\![e_z]\!]^\bullet\ (x.\,[\![e_z]\!]^\bullet) \\
[\![x]\!]^\bullet &= x \\
[\![\mathbf{fun}\, f.x.e]\!]^\bullet &= \mathbf{fun}\, f.x.\,[\![e]\!]^\bullet \\
[\![e_f\ e_x]\!]^\bullet &= [\![e_f]\!]^\bullet\ [\![e_x]\!]^\bullet \\
[\![\mathbf{afun}\, f.x.e]\!]^\bullet &= \mathbf{fun}\, f.x.\lambda k.\,[\![e]\!]^\$\ k \\
[\![\mathbf{mfun}\, f.x.e]\!]^\bullet &= \\
&\quad \mathbf{fun}\, f.x.\lambda k_1. && 1 \\
&\quad\ \ \mathbf{let}\ k_2 = \lambda y_r.\mathbf{memo}\ (k_1\ y_r)\ \mathbf{in} && 2 \\
&\quad\ \ \mathbf{write}\ k_2\ (\lambda y_k. && 3 \\
&\quad\ \ \ \ \mathbf{let}\ k' = \lambda y_r.\mathbf{read}\ y_k\ (\lambda k_3.k_3\ y_r)\ \mathbf{in} && 4 \\
&\quad\ \ \ \ \mathbf{memo}\ ([\![e]\!]^\$\ k')) && 5 \\
[\![\ell]\!]^\bullet &= \ell
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{zero}]\!]^\$\ v_k &= v_k\ \mathbf{zero} \\
[\![\mathbf{succ}\, e]\!]^\$\ v_k &= [\![e]\!]^\$\ (\lambda y.v_k\ (\mathbf{succ}\ y)) \\
[\![\mathbf{caseN}\, e_n\, e_t\, (x.e_s)]\!]^\$\ v_k &= \\
&\quad [\![e_n]\!]^\$\ (\lambda y.\mathbf{caseN}\ y\ ([\![e_z]\!]^\$\ v_k)\ (x.([\![e_s]\!]^\$\ v_k))) \\
[\![x]\!]^\$\ v_k &= v_k\ x \\
[\![\mathbf{fun}\, f.x.e]\!]^\$\ v_k &= v_k\ [\![\mathbf{fun}\, f.x.e]\!]^\bullet \\
[\![e_f\ e_x]\!]^\$\ v_k &= [\![e_f]\!]^\$\ (\lambda y_f.\,[\![e_x]\!]^\$\ (\lambda y_x.v_k\ (y_f\ y_x))) \\
[\![\mathbf{afun}\, f.x.e]\!]^\$\ v_k &= v_k\ [\![\mathbf{afun}\, f.x.e]\!]^\bullet \\
[\![\mathbf{mfun}\, f.x.e]\!]^\$\ v_k &= v_k\ [\![\mathbf{mfun}\, f.x.e]\!]^\bullet \\
[\![e_1\,\$\,e_2]\!]^\$\ v_k &= [\![e_1]\!]^\$\ (\lambda y_f.\,[\![e_2]\!]^\$\ (\lambda y_x.(y_f\ y_x)\ v_k)) \\
[\![\mathbf{put}\,\$\,e]\!]^\$\ v_k &= [\![e]\!]^\$\ (\lambda y.(\mathbf{write}\ y\ v_k)) \\
[\![\mathbf{get}\,\$\,e]\!]^\$\ v_k &= [\![e]\!]^\$\ (\lambda y.(\mathbf{read}\ y\ v_k)) \\
[\![\ell]\!]^\$\ v_k &= v_k\ \ell
\end{aligned}
$$

Figure 6. Type translation $[\![\tau^{\mathsf{asrc}}]\!] = \tau^{\mathsf{satgt}}$ (top) and expression translations $[\![e^{\mathsf{asrc}}]\!]^\bullet = e^{\mathsf{satgt}}$ and $[\![e^{\mathsf{asrc}}]\!]^\$\ v_k^{\mathsf{satgt}} = e^{\mathsf{satgt}}$ (bottom).

($\mathbf{put}\,\$\,e$ and $\mathbf{get}\,\$\,e$) by extending the continuation v_k with a **write** or **read** operation. Although the **halt** expression is not in the image of the translation, it may be used as an initial continuation for evaluating a cps-converted program.

The translation of memoizing functions (**mfun**) is central to producing effective self-adjusting programs through compilation. Note that a naïve translation:

$$[\![\mathbf{mfun}\, f.x.e]\!]^\bullet \quad = \quad \mathbf{fun}\, f.x.\lambda k.\mathbf{memo}\ ([\![e]\!]^\$\ k)$$

is ineffective, because $\mathbf{memo}\ ([\![e]\!]^\$\ k)$ will only result in a memoization hit when both the argument x and continuation k are the same as in the previous run, despite the fact that the computation of e depends only on x and not on the calling context (now explicit in the continuation k). Ideally, a memoizing function (in cps) should maintain a map from arguments to results; if the function is called with a previously-seen argument, then the (possibly different) continuation is invoked with the memoized result immediately. The compilation of a memoizing adaptive ASrc function into a SATgt function consists of two parts, which we explain by referring to the line numbers in the translation of **mfun**. Note that, unlike the translation of a non-memoizing adaptive function (**afun**), a memoizing adaptive function modifies its continuation before executing the function body.

First, memoizing on the argument alone is achieved by treating the continuation as changeable data and placing it in a modifiable reference (lines 3-5). Therefore, if change propagation starts re-executing due to an inconsistent read or write *before* a call to a memoizing function, then re-executing the function call with the same argument can result in a memoization hit even if the continuation differs. By placing the continuation in a modifiable bound to y_k (line 3), the function is memoized on the argument x and the reference y_k (line 5). Provided write allocation stores the new continuation at the same location, the function is effectively memoized on the argument alone. When the function body invokes its continuation k' with the result, the new continuation must be passed the old result. This is achieved by reading the continuation back from the modifiable into k_3 and invoking it with the result y_r (line 4).

Second, if change propagation encounters an inconsistent read or write *during* the execution of a memoizing function, then it is necessary to re-execute the function body but it may be possible to avoid invoking the function's continuation. When an inconsistent read or write occurs during the execution of the body of a memoizing function, the function's continuation k_1 will be the same as on the previous run. If the function body yields the same result value during re-execution as during the previous run, then it is desirable to reuse the previous computation, rather than invoking (the same) k_1 with (the same) result. This can be achieved by wrapping continuation k_1 with the **memo** operation (line 2).

Although SATgt memoization can match whenever identical expressions are evaluated, we do not implement this semantics because it would require comparing arbitrary expressions and maintaining a global memoization table. Memoizing a function enables using local memo tables indexed by the function's argument and is sufficient for making change propagation work well in practice.

5.1 An Example Execution

To illustrate how translated programs execute, recall the `partition` function (Figure 1) and its translation (Figure 2). Consider executing `partition` with the constant `true` predicate on the modifiable list `[1,3]` and then updating the input list to `[1,2,3]` by inserting an element. Figure 7 shows a pictorial representation of the traces from the two executions. A trace consists of read and write actions, and memoized calls to `part` and continuations. The differences between the two runs are highlighted on the right.

The continuation passed to `partition` (and thus to the first call to `part_memo`) is named `k0`. Each recursive call to `part_memo` memoizes and writes its continuation into a modifiable (lines 15 and 17) and makes a memoized call to `part` (line 19), which reads the next modifiable in the list (line 3), makes a continuation that ultimately writes an element to the output (lines 8-12), and calls `part_memo`. Modifiables containing the continuations are labeled `mki`. To insert 2 into the input and update the output, we allocate a new modifiable `l2`, change the modifiable `l1`, and run change propagation.

Change propagation starts re-executing the read of `l1`, which writes the same continuation `k1` as before to `mk2` and calls `part` with `l2` and `mk2`. After `l2` is read, a new continuation `k2'` is written to `mk3`[4] and `part` is called with `l3` and `mk3`. This call is in the reuse trace, so there is a memoization hit, which completes the execution of the first read. Since the continuation written to `mk3` is new, change propagation starts re-executing the read of `mk3`, which calls the continuation `k2'` with `o3`. The continuation `k2'` writes to `o2`, reads `mk2`, and calls the continuation `k1`. The continuation `k1` writes to `o1`[5], reads `mk1`, and calls `k0` with `o1`. This call is in the reuse trace, so there is a memoization hit, which completes the execution of the second read and, as there are no more inconsistent reads, change propagation completes. Change propagation performs the work for the element before the insertion point and at the insertion point only; regardless of the input size, the result is updated by performing a small, constant amount of work.

6. Implementing Self-Adjusting SML

To validate this approach to self-adjusting programming, we have extended the Standard ML programming language with syntactic, static typing, and library support for self-adjusting programs and we have modified the MLton compiler to support the extended language.[6] This integration yields direct linguistic support for self-adjusting programming, which has been suggested to be necessary for scaling to large programs and for enforcing the numerous invariants required for self-adjusting computation to work correctly (Acar et al. 2006a).

6.1 Self-Adjusting SML

Thorough support for self-adjusting programming in Standard ML is provided by the combination of language extensions and a library of adaptive and meta-level operations. A user interested in writing self-adjusting programs need only familiarize herself with these two components.

6.1.1 Language extensions

Following ASrc, we introduce an adaptive function type, written "*ty* `-$>` *ty*". An adaptive function is introduced via either the expression form "`afn` *match*" or the declaration forms "`afun` *tyvarseq fvalbind*" and "`mfun` *tyvarseq fvalbind*". All of these forms simply change the introductory keyword of existing Standard ML forms for function introduction; hence, adaptive functions enjoy the same syntactic support for mutually recursive function definitions, clausal function definitions, and pattern matching as (normal) functions. An adaptive function is eliminated via the expression form "*exp* `$` *exp*"; as in ASrc, adaptive applications may only appear within the body of an adaptive function.

[4] Stealing allows the write to reuse `mk3`.

[5] Again, stealing allows the write to reuse `o1`.

[6] The implementation may be obtained at `http://ttic.uchicago.edu/~pl/sa-sml`.

```
signature ADAPTIVE = sig
 type 'a box
 val put : 'a -$> 'a box
 val get : 'a box -$> 'a

 (** Stealing, Reuse, and Memoization **)
 type 'a eq = 'a * 'a -> bool
 val eqDflt : 'a eq
 type 'a hash = 'a -> word
 val hashDflt : 'a hash
 type 'a key = 'a eq * 'a hash

 val mkPut' : 'k key * 'a eq -$> ('k * 'a -$> 'a box)
 val mkPut : unit -$> ('k * 'a -$> 'a box)
 val putTh : (unit -$> 'a) -$> 'a box

 val memoFix : ('a key * 'r eq *
                ((('a -$> 'r) -> ('a -$> 'r)))
               -> ('a -$> 'r)
 val memoCont : ('r eq * ('a -$> 'r))
                -> ('a -$> 'r)

 (** Meta operations **)
 val new : 'a eq * 'a -> 'a box
 val change : 'a box * 'a -> unit
 val deref : 'a box -> 'a

 datatype 'a res = Value of 'a | Exn of exn
 val call : ('a -$> 'r) * 'a -> 'r res ref
 val propagate : unit -> unit
end
structure Adaptive :> ADAPTIVE = struct ... end
```

Figure 8. Signature for the `Adaptive` library.

Note that while these language extensions introduce a new type, they do not (significantly) change the type system of Standard ML.[7] Hence, all of the familiar features of Standard ML (parametric polymorphism, type inference, *etc*.) are immediately available to adaptive functions. Furthermore, these extensions are easily integrated into a Standard ML compiler, since their treatment by the front-end is entirely analogous to the treatment of normal functions.

6.1.2 Library interface

Figure 8 gives the interface of the `Adaptive` library. The library provides the `box` type and the `get` and `put` adaptive functions from ASrc. The next group of types and values are mechanisms to control the nondeterministic stealing, reuse, and memoization that appears in the dynamic semantics of Section 4. The last group of types and values are *meta operations* used by a host mutator program to control a self-adjusting computation.

The functions for controlling nondeterminism require equality predicates and hash functions; the `key` type is an abbreviation for a tuple consisting of an equality predicate and a hash function. The default equality predicate and hash function use MLton extensions that provide a polymorphic structural equality predicate and a polymorphic structural hash function, both of which may be instantiated

[7] Technically, we must introduce new typing rules for adaptive functions and adaptive applications, but they are identical to the typing rules for normal functions and normal applications except for the use of the adaptive function type. Similarly, as in ASrc, a mode component in the typing rules is used to preclude adaptive applications from the body of normal functions.

at any type[8]; on values of function type, they operate on the structure of the function closure. The defaults make it easy to migrate an ordinary pure functional program to a self-adjusting program and avoid the need for awkward tagged values used in previous work (Acar et al. 2006a).

Controlling the nondeterministic stealing, reuse, and memoization during change propagation is the most complex and subtle aspect of using the adaptive library. However, these mechanisms are only necessary to improve the efficiency of change propagation, not to enforce its correctness.

The `mkPut'` operation takes a key and an equality predicate and returns an *allocator*, a function for allocating boxes. The allocator records the locations that were allocated for individual writes during an execution. When those writes are re-executed during change propagation, the library attempts to reuse the locations allocated in the previous execution by matching the supplied key element. A hash table is used to map key elements to locations, which motivates the components of the key. The mechanism is robust in the presence of repeated key elements: collisions may degrade the efficiency of change propagation, but not its correctness. When the location of a box is reused, the equality predicate determines whether the contents of the box have changed.[9]

The `mkPut` operation is a special case of `mkPut'` that uses the default equality predicate and hash function:

```
afun mkPut () =
  mkPut' $ ((eqDflt, hashDflt), eqDflt)
```

The `putTh` operations corresponds to a common scenario, where the allocating function returned by `mkPut` is used exactly once:

```
afun putTh th =
  let val putM = mkPut $ () in
  putM $ ((), th $ ())     end
```

If change propagation begins re-executing within the body of the adaptive thunk, then the result will be stored at the same location that was allocated during the previous execution.

The `memoFix` operation is used to create memoized functions. The operation takes a key tuple on the argument, an equality predicate on the result, and the function to memoize; since the `memoFix` operation is used to memoize recursive functions, it takes the function to memoize in the form suitable for a fixed-point combinator. Recall the translation of **mfun** from Figure 5. In order to reuse the location returned by $\mathbf{write}\ k_2\ (\lambda y_k.\cdots)$ at line 3, the equality predicate and hash function on the function argument are used to match the corresponding $\mathbf{write}\ k_2\ (\lambda y_k.\cdots)$ in the previous execution. The equality predicate and hash function on the function argument are also used to effect the $\mathbf{memo}\ (\llbracket e \rrbracket^{\$}\ k')$ at line 5.[10] The equality predicate on the result is used to implement the $\mathbf{memo}\ (k_1\ y_r)$ at line 2.

A subtlety of `memoFix` is that it memoizes *only* the recursive calls: re-executing the call of a function memoized by `memoFix` will only attempt to match calls of the function in the previous execution that are nested within the same root call of the function as is the re-executing call. Despite this apparent limitation, `memoFix` suffices for most self-adjusting computations, since change propagation typically begins re-execution within a nested call of a recursive function.

In practice, though, the adaptive library may provide additional operations with different memoization properties. For example, the `memoCont` operation implements only the $\mathbf{memo}\ (k_1\ y_r)$ at line 2 and only for non-recursive calls of the memoized function.

The `new`, `change`, and `deref` operations are used by the host mutator program to create and modify inputs for and inspect outputs of a self-adjusting computation. The `call` operation is used to perform the initial execution of a self-adjusting computation. Note, that the `call` operation is the *only* means of "applying" an adaptive function outside the body of another adaptive function. The result of the `call` operation is a mutable reference cell containing the output (distinguishing between normal and exceptional termination) of the self-adjusting computation. After changing the inputs, the `propagate` operation is used to perform change propagation; the new output may be observed as the new contents of the mutable cell. Since the meta operations are impure, they should not be used within adaptive functions — correctness demands that self-adjusting computations not make calls to impure normal functions.

6.2 Implementation

To implement Self-Adjusting SML, we modified the MLton compiler (version 20070826) to support the language extensions for adaptive functions and to perform a cps-transformation pass that converts adaptive functions into continuation-passing style and we developed libraries to support self-adjusting computation, on top of which the `Adaptive` library is implemented. Both the language extensions and the compiler modifications that we describe below are agnostic to the fact that they have been introduced to support self-adjusting computations. Indeed, the compiler provides no direct support for tracking the dynamic data dependences of self-adjusting computations or for re-executing computations during change propagation. That support comes from the self-adjusting-computation libraries. This approach minimized the necessary compiler modifications.

In total, we added or modified 1600 lines of code in the MLton compiler, of which 760 correspond to the cps-transformation pass, and wrote 2800 lines of code for the libraries.

6.2.1 Compiler modifications

Front-end. As suggested above, the language extensions are easily integrated into MLton, since their treatment by the front-end is entirely analogous to the treatment of normal functions. Most changes simply generalize the existing function introduction and function application forms to adaptive functions in a small number of the compiler intermediate languages.

To support the `mfun` keyword, the front-end desugars `mfun` declarations to adaptive functions whose bodies are memoized with (generalizations of) the `memoFix` operation. The desugaring uses the default equality predicate and hash function and supports mutually recursive `mfun` declarations.

Adaptive cps transformation. The cps-transformation pass is implemented as an SXML-program to SXML-program transformation; SXML is the name of a simply-typed, higher-order, A-normal-form intermediate language in the compiler. This is the most significant change to the compiler. Each of the ILs prior to and including the SXML IL were extended with adaptive function and adaptive application forms and the optimizations on and transformations between these ILs were extended to handle the new forms. The cps-transformation pass eliminates all adaptive functions and adaptive applications in the input SXML program. The output SXML program (having no adaptive forms) corresponds to an SXML program in the unmodified compiler; hence, no subsequent ILs, optimizations, or transformations in the compiler require any changes.

[8] This differs from Standard ML's polymorphic equality, which may only be instantiated at equality types.

[9] This is the implementation realization of the $v_\circ \equiv_\alpha v$ predicate in the $\mathbf{read/reuse}$ rule of Figure 5. Since there is a degree of nondeterminism between the $\mathbf{read/reuse}$ and $\mathbf{read/change}$ rules, the equality predicate need only be conservative.

[10] As before, a hash table is used to map arguments to continuation locations.

The actual cps transformation is entirely straightforward; indeed, the fact that the input program is in A-normal form makes the transformation even simpler than the one presented in Section 5. The only additional notable features of the transformation is the treatment of exceptions. Since the SXML IL has explicit `raise` and `handle` constructs for exceptions, we use a double-barrelled continuation-passing style transformation (Kim et al. 1998; Thielecke 2002), where each adaptive function is translated to take two continuations: a return continuation and an exception-handler continuation. When transforming the body of an adaptive function, `raise`s are translated to invocations of the exception continuation and `handle`s are translated to pass a local handler continuation to the body. This treatment of exceptions allows adaptive functions to freely raise and handle exceptions, just like normal functions. Indeed, an adaptive function may handle an exception raised by a normal application appearing in its body.

Primitives. Bridging the gap between the compiler and the self-adjusting-computation libraries are two primitives that witness the implementation of adaptive functions in continuation-passing style:

```
type 'a cont = 'a -> unit
type ('a, 'b) xfn = ('b cont * exn cont * 'a) cont
val afn_to_xfn : ('a -$> 'b) -> ('a, 'b) xfn
val xfn_to_afn : ('a, 'b) xfn -> ('a -$> 'b)
```

These primitives are not exposed to the user, as they could be used to violate invariants expected by the self-adjusting computation libraries described below. The primitives are eliminated by the cps-transformation pass, where they are implemented as the identity function.

6.2.2 Self-adjusting computation libraries

A low-level self-adjusting-computation library effectively implements the semantics of SATgt, providing operations for writing and reading modifiables, for memoizing functions, and for performing change propagation. It is an implementation of the previously proposed monadic libraries (Acar et al. 2006b,a) specialized for cps. To provide efficient change propagation, the implementation maintains a priority queue of continuations from inconsistent reads of modifiable references. The implementation represents trace elements (e.g., writes, reads) with a time-stamp data structure and uses hash tables as described above for stealing and memoization. By taking advantage of the fact that continuations are explicit, we are able to simplify the representation of reads to use only one time stamp (previous work required two) and eliminate the mutual recursion between change propagation and memoization.

The high-level `Adaptive` library described in Section 6.1.2 is implemented as a wrapper around the low-level library. Since the high-level library uses the adaptive functions from the language extensions while the low-level library uses explicit continuation-passing style, we use the the `afn_to_xfn` and `xfn_to_afn` primitives to convert between the two representations.

7. Experiments

We describe a preliminary experimental evaluation of our compilation technique. In summary, the experiments indicate that our approach to compiling self-adjusting programs is consistent with the previously reported asymptotic bounds and experimental evaluations (Acar et al. 2006b) of the monadic libraries (Acar et al. 2006a) for self-adjusting computation.

7.1 Synthetic applications

Benchmarks. We implemented self-adjusting versions of the following algorithms, which are used in previous evaluations.

- **filter**, **map**, **reverse**, **fold**: The standard list functions.
- **merge-sort, quick-sort**: The merge-sort and quick-sort algorithms for list sorting.
- **diameter**: An algorithm (Preparata and Shamos 1985) for computing diameter (extent) of a planar point set.
- **quick-hull**: The quick-hull (Barber et al. 1996) algorithm for the convex-hull of a planar point set.

These algorithms utilize a number of computing paradigms including simple iteration (`filter`, `map`), accumulator passing (`reverse`, `quick-sort`), random sampling (`fold`), and divide-and-conquer (`merge-sort`, `quick-sort`, `quick-hull`).

Our benchmarks are specific instances of the algorithms. All list benchmarks operate on integer lists. The `filter` benchmark keeps the even elements in an integer list. The `map` benchmark adds a fixed value to each element in an integer list. Both `minimum` and `sum` are instances of the `fold` algorithm. The sorting benchmarks (`qsort`, `msort`) sort integer lists. The computational-geometry benchmarks (`diameter, quick-hull`) operate on lists of points in two dimensions.

The input to our experiments is randomly generated. To generate a list of n integers, we choose a random permutation of the integers from 1 to n. To generate points in two dimensions, we choose random points uniformly from within a disc of radius $10n$.

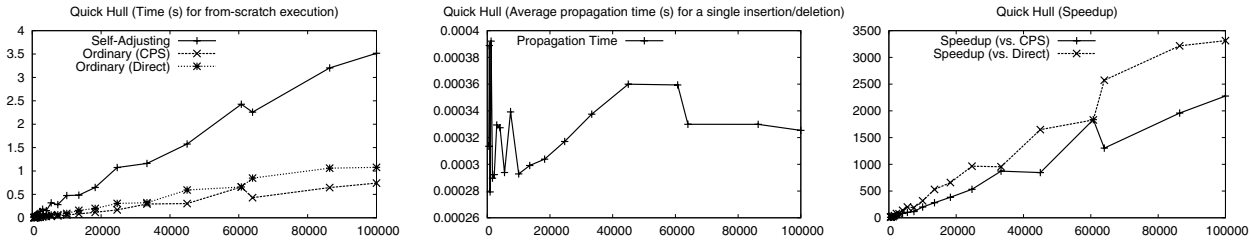
Measurements. Our experiments were performed on a desktop computer (two single-core 2GHz AMD Opteron processors; 8GB physical memory; Linux 2.6.23 operating system (Fedora 7)). Benchmarks were executed with the "`gc-summary`" runtime option, which reports GC statistics (*e.g.*, GC time). In this evaluation, we do not report GC times.

To understand the effects of the cps-transformation on the ordinary (non-self-adjusting) version of our benchmarks, we consider two instances of each ordinary benchmark. The cps instance of an ordinary benchmark is written using adaptive functions and adaptive applications, but does not use the adaptive library; the compiler transforms these adaptive functions and adaptive applications into cps. The direct-style instance is written using normal functions and normal applications (and does not use the adaptive library); these functions and applications are unchanged by the cps-transformation pass. Our results show that there is a slight performance difference between directly-style instance and the cps instances: direct style is often slightly faster, but not always. This confirms that our selective cps translation is effective in reducing the overheads of continuations. We therefore use the direct-style instance of the ordinary version for comparing ordinary and self-adjusting versions. We measure the following quantities:

- **Time for from-scratch execution:** The time for the from-scratch execution of the ordinary or self-adjusting version.
- **Average propagation time for a single insertion/deletion:** For each element in the input list, we delete the element, run change propagation, insert the element (at the same point), and run change propagation. The average is taken over all propagations.
- **Overhead:** This is the ratio of the time for the from-scratch execution of the self-adjusting version to the time for the from-scratch execution of the ordinary version with the same input.
- **Speedup:** This is the ratio of the time for the from-scratch run of the ordinary version to the average time for propagating a single insertion/deletion.

In our measurements, we isolate the quantity of interest; that is, we exclude the time to create the initial input and, in change-propagation timings, we exclude the time to perform the initial run.

Results. Table 1 gives summaries for the benchmarks at fixed input sizes. Each columns show the measurements for the quantities



(a) Time (s) for from-scratch execution (b) Average propagation time (s) (c) Speedup

Figure 9. Selected measurements for `quick-hull`.

Application (Input size)	Ord. Exec. (s)	Self-Adj. Exec. (s)	Self-Adj. Avg. Propagate (s)	Overhead	Speedup
filter (10^6)	0.22	4.00	0.000007	18.2	33170
map (10^6)	0.32	9.53	0.000018	30.0	17427
reverse (10^6)	0.20	4.83	0.000014	24.3	14410
minimum (10^6)	0.12	2.80	0.000020	23.5	5877
sum (10^6)	0.12	2.94	0.000155	25.6	740
msort (10^5)	0.56	17.96	0.001442	32.3	386
qsort (10^5)	0.35	11.16	0.000949	32.3	365
diameter (10^5)	1.25	3.54	0.000343	2.8	3628
quick-hull (10^5)	1.08	3.52	0.000325	3.3	3313

Table 1. Summary of benchmark timings.

described above. The overheads range between 3 and 33. The integer benchmarks have higher overheads because the perform trivial work between self-adjusting computation primitives. Computational geometry benchmarks have less overheads because the geometry operations are more expensive (relative to self-adjusting computation primitives). Change propagation leads to orders of magnitude speedups over from-scratch executions. This is because there is often a near-linear time asymptotic gap between executing from scratch and performing change propagation.

Figure 9(a) compares the from-scratch executions of the ordinary and self-adjusting versions of `quick-hull`. The figure shows that there is a small difference between the cps and the direct-style instance of the ordinary version. Figure 9(b) shows the average change-propagation time for a single insertion/deletion. As the figure shows, the time remains nearly constant. Intuitively, this is because many of the input changes do not change the output, which change propagation can take advantage of to update the output quickly. Figure 9(c) shows the average speedup, which increases linearly with the input size to exceed three orders of magnitude.

7.2 Raytracer application

For a less synthetic benchmark, we implemented a self-adjusting raytracer. This application would have been quite cumbersome to write using the previous monadic libraries (Acar et al. 2006a), but was straightforward using this work. The raytracer supports point and directional lights, sphere and plane objects, and diffuse, specular, transparent, and reflective surface properties. The surface properties of objects are changeable data; thus, for a fixed input scene (lights and objects) and output image size, we can render multiple images (via change propagation) that vary the surface properties of objects in the scene. Note that this application is not always well suited to self-adjusting computation because making a small change to the input can affect a large portion of the output.

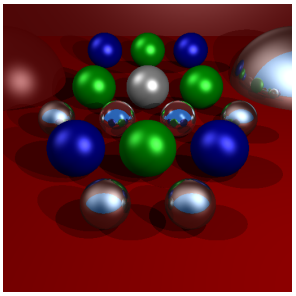


Figure 10. Ray-tracer output.

For experiments, we render an input scene (shown on the right) of 3 light sources and 19 objects with an output image size of 512×512 and then repeatedly change the surface properties of a single surface (which may be shared by multiple objects in the scene). A $\cdot^D$ change indicates that the surface was toggled with a diffuse (non-reflective) surface, while an $\cdot^M$ change indicates that the surface was toggled with a mirror surface. We measure the time for from-scratch execution for both the ordinary & self-adjusting versions, and the average propagation time for a single toggle of the surface. For each change to the input, we also measure the change in the output image as a fraction of pixes.

Image Size 512×512		Self-Adj. Exec. (s) 7.643	Ord. Exec. (s) 2.563
Surface Changed	Image Diff. (% pixels)	Self-Adj. Avg. Propagate (s)	Ord. Avg. From-Scratch (s)
A^D	57.22%	3.430	2.805
A^M	57.22%	11.277	3.637
B^D	8.43%	0.731	2.817
B^M	8.43%	1.471	2.781
C^D	9.20%	0.855	2.810
C^M	9.20%	1.616	2.785
D^D	1.85%	0.142	2.599
D^M	1.85%	0.217	2.731
E^D	19.47%	2.242	2.154
E^M	19.47%	4.484	2.237

Table 2. Summary of raytracer timings.

Table 2 shows that self-adjusting raytracer is about three times slower than the ordinary version. Change propagation yields speedups of 1.6 to 18.0 when less than 10% of the output image changes. If the output changes more significantly (surfaces A and E), then the change-propagation can be slower than the ordinary version. This is expected because the amount of work change propagation performs is roughly proportional to the fraction of change in the output (e.g., updating half the output requires half the work of a from-scratch execution). Changes that makes a surface reflective

(the $\cdot^M$ changes), are more expensive, because they require casting new rays in addition to updating existing rays.

8. Related Work

We review related work on incremental computation and some recent interactions between functional reactive programming and incremental computation. For a more complete list of references and other approaches to incremental computation, we refer the reader to the bibliography of Ramalingam and Reps (1993).

Dependence-graph techniques record the dependencies between data in a computation, so that a change-propagation algorithm can update the computation when the input is changed. Demers, Reps, and Teitelbaum (1981) and Reps (1982) introduced the idea of *static dependence graphs* and presented a change-propagation algorithm for them. The main limitation of static dependence graphs is that they do not permit the change-propagation algorithm to update the dependence structure. This significantly restricts the types of computations to which static-dependence graphs can be applied. For example, the INC language (Yellin and Strom 1991), which uses static dependence graphs for incremental updates, does not permit recursion. Another approach to incremental computation is based on memoization (Bellman 1957; McCarthy 1963; Michie 1968), where function calls are remembered and re-used when possible. Pugh and Teitelbaum (1989) were the first to apply memoization (also called function caching) to incremental computation. Since their work, others have investigated applications of various forms of memoization to incremental computation (Abadi et al. 1996; Liu et al. 1998; Heydon et al. 2000; Acar et al. 2003).

The first work on self-adjusting computation, called Adaptive Functional Programming (AFP) generalized dependence-graph approaches by introducing dynamic dependence graphs (DDGs) and proposing language facilities for writing adaptive programs (Acar et al. 2002). As an adaptive programs executes, a run-time system construct its DDG. When data changes takes place, a change-propagation algorithm updates both the output and the DDG by re-executing the parts of the computation affected by the changes as necessary. In AFP, the change-propagation algorithm conservatively deletes the parts of the DDG that might have a control dependence on changed data and constructs replacements by executing code as necessary. This can cause change-propagation to perform more work than optimal. Subsequent work identified a duality between change propagation and memoization to improve the effectiveness of change propagation by enabling the re-use of subgraphs of deleted DDGs via a form of memoization (Acar et al. 2006b). Recent work showed that self-adjusting computation may be generalized to support updateable (imperative) modifiable references (Acar et al. 2008a).

Self-adjusting computation has been implemented by extending several existing languages. Carlsson (2002) presented an implementation of the original AFP library (Acar et al. 2002) in Haskell. By using monads, the Haskell library ensures some correct-usage properties that the AFP library did not enforce. A later version of our SML library applied a techniques similar to Carlsson's to ensure safe usage of certain primitives (Acar et al. 2006a). Safe usage of memoization primitives, however, could not be enforced statically in the library setting. Shankar and Bodik (Shankar and Bodik 2007) gave a specialized implementation of self-adjusting computation in the Java language. The implementation is targeted to invariant-checking and is not sound in general. It restricts the kinds of programs that can be written (e.g., return values from functions calls can only be used in certain ways). Recent work presented an implementation of self-adjusting computation in the C language and extended change-propagation to support efficient garbage collection (Hammer and Acar 2008).

Functional Reactive Programming (FRP) (e.g., Elliott and Hudak 1997; Elliott 1998; Nilsson et al. 2002; Courtney 2001) offers techniques for programming reactive system. FRP provides primitives for programming with behaviors and events, which are continuous and discrete functions of time respectively. Although FRP research remained largely orthogonal to incremental computation, it may benefit from incremental computation, because computations performed at consecutive time steps can be similar. In particular, self-adjusting computation may be applied to FRP by representing time-varying values (e.g., behaviors, events, signals) using modifiable references and by performing change-propagation to update the computation when necessary. Cooper and Krishnamurti (2004; 2006) give a Scheme implementation of Adaptive Functional Programming (AFP) (Acar et al. 2002) for this purpose. The approach is also adapted to the Java language (fla). There are some differences between their implementation and AFP. AFP provides safe language facilities for controlling the granularity of dependence tracking, while pointing out that all dependences can also be tracked. The Scheme implementation tracks all dependences by placing all time-varying values (called signals) in modifiables. Subsequent work develops static optimization techniques for reducing the cost of tracking all dependences (Burchett et al. 2007) but offers no comparison to the AFP approach. AFP provides techniques for correct and efficient implementation of DDGs; a key component of the implementation is a representation of DDGs using topological orders and order-maintenance data structures. The Scheme implementation uses depth/height information instead of a topological order; this complicates the handling of cyclic dependences and the dynamic maintenance of the DDG. It can also be inefficient, because deleting a DDG node can change the height of all the remaining nodes, requiring linear time in the size of the DDG. The authors do not discuss how their implementation relates to or differs from that of AFP. As suggested elsewhere (Cooper and Krishnamurthi 2004), AFP (and more generally self-adjusting computation), may be used to support FRP directly.

9. Conclusion

In this paper, we develop a safe interface for self-adjusting computation and describe techniques for compiling self-adjusting programs written in the interface. The interface consists of simple `get` and `put` operations and an `mfun` keyword for declaring memoizing functions. These primitives can be inserted anywhere in the code subject to some simple type constraints. Programs written with these primitives can be statically checked using simple extensions to conventional type systems. Type safe programs are guaranteed to respond to changes correctly via change propagation.

We show that self-adjusting programs written in the proposed interface can be compiled by an adaptive-cps translation. The translation recovers sufficient information about the program to employ the previously proposed primitives. In particular the translation uses continuations as a coarse approximation to programmer-supplied fine-grain dependences. To ensure that change propagation remains efficient with the compiler-inferred dependences, the translation generates memoized cps functions that may be re-used even when their continuations differ. This is achieved by memozing continuations and by treating continuations as changeable data by writing them into pre-allocated modifiables.

We show that the proposal can be realistically incorporated into a language by extending the SML language and an existing optimizing compiler (MLton). We present a preliminary evaluation of the implementation by considering a number of applications. The experiments indicate that the approach is consistent with the previous proposal based on manual re-writing in terms of practical performance and asymptotic complexity. Whether this correspondence can be proved (or disproved) remains to be an open question.

References

MLton. http://mlton.org/.

Flapjax programming language. www.flapjax-lang.org.

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

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive Functional Programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

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

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A Library for Self-Adjusting Computation. *Electronic Notes in Theoretical Computer Science*, 148(2), 2006a.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006b.

Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic Algorithms via Self-Adjusting Computation. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pages 636–647, September 2006c.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian Inference. In *Neural Information Processing Systems (NIPS)*, 2007.

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

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 (ESA)*, September 2008b.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Inference on General Graphical Models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008c.

C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

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

Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A Static Optimization Technique for Transparent Functional Reactivity. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–80. ACM, 2007.

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

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

Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*, 2006.

Antony Courtney. Frappé: Functional Reactive Programming in Java. In *PADL '01: Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, pages 29–44. Springer-Verlag, 2001.

Olivier Danvy and John Hatcliff. CPS Transformation after Strictness Analysis. *Letters on Programming Languages and Systems (LOPLS)*, 1(3):195–212, 1993a.

Olivier Danvy and John Hatcliff. On the Transformation between Direct and Continuation Semantics. In *Proceedings of the Ninth Conference on Mathematical Foundations of Programming Semantics (MFPS)*, pages 627–648, 1993b.

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.

Conal Elliott. Functional Implementations of Continuous Modeled Animation. *Lecture Notes in Computer Science*, 1490:284–299, 1998.

Conal Elliott and Paul Hudak. Functional Reactive Animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273. ACM, 1997.

Matthew Hammer and Umut A. Acar. Memory Management for Self-Adjusting Computation. In *The 2008 International Symposium on Memory Management*, 2008.

Fritz Henglein, Henning Makholm, and Henning Niss. Effect Types and Region-based Memory Management. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 3, pages 87–135. MIT Press, Cambridge, MA, 2005.

Allan Heydon, Roy Levin, and Yuan Yu. Caching Function Calls Using Precise Dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 311–320, 2000.

Jung-taek Kim and Kwangkeun Yi. Interconnecting Between CPS Terms and Non-CPS Terms. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW)*, pages 7–16, 2001.

Jung-taek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the Overhead of ML Exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML*, pages 112–119, 1998.

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

John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, 1968.

Lasse Nielsen. A Selective CPS Transformation. In *Proceedings of the Seventeenth Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, volume 45 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 311–331. Elsevier, November 2001.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag Inc., 1985.

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.

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

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

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 (PLDI)*, 2007.

Hayo Thielecke. Comparing Control Constructs by Double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):367–412, 2002.

D. M. Yellin and R. E. Strom. INC: A Language for Incremental Computations. *ACM Transactions on Programming Languages and Systems*, 13 (2):211–236, April 1991.