# Programmable Self-Adjusting Computation

Ruy Ley-Wild

CMU-CS-10-146
October 11 2010

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Guy Blelloch, Chair
Stephen Brookes
Robert Harper
Umut Acar, Max-Planck Institute for Software Systems

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy.*



This research was sponsored by a Bell Labs Graduate Research Fellowship from Alcatel-Lucent Technologies, by the National Science Foundation under grant number CCF-0429505, by an IBM Open Collaborative Faculty award, and by donations from Intel Corporation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

**Keywords:** self-adjusting computation, adaptivity, memoization, change-propagation, continuation-passing style, typed compilation, cost semantics, trace distance, traceable data type

*Self-Adjusting Computation is the center of the mandala.*

# Abstract

Self-adjusting computation is a paradigm for programming incremental computations that efficiently respond to input changes by updating the output in time proportional to the changes in the structure of the computation. This dissertation defends the thesis that high-level programming abstractions improve the experience of reading, writing, and reasoning about and the efficiency of self-adjusting programs.

We show that high-level language constructs are suitable for writing readable self-adjusting programs and can be compiled into low-level primitives. In particular, language constructs such as ML-style *modifiable references* and *memoizing functions* provide orthogonal mechanisms for identifying stale computation to re-execute and opportunities for computation reuse. An *adaptive continuation-passing style* (ACPS) transformation compiles the high-level primitives into a continuation-passing language with explicit support for incrementality.

We show that a high-level *cost semantics* captures the performance of a self-adjusting program and a theory of *trace distance* suffices for formal reasoning about the efficiency of self-adjusting programs. The formal approach enables generalizing results from concrete runs to asymptotic bounds and compositional reasoning when combining self-adjusting programs.

We raise the level of abstraction for dependence-tracking from modifiable references to *traceable data types*, which can exploit problem-specific structure to identify stale computation. We consider *in-order* memoization that efficiently reuses work from previous runs in the same execution order and *out-of-order* memoization that allows previous work to be reordered at an additional expense.

The compilation approach is realized in the $\Delta$ML language, an extension to SML, and implemented as an extension to MLton with compiler and runtime support. Experimental evaluation of $\Delta$ML shows that applications with modifiable references are competitive with previous approaches. Moreover, traceable data types enable an asymptotic improvement in time and space usage relative to modifiable references.

# Acknowledgments

This dissertation culminates my research apprenticeship under my "official" advisor Guy Blelloch and my "unofficial" advisor Umut Acar. I'm grateful to Umut for introducing me to self-adjusting computation and for his persistent involvement in my academic development. I'm grateful to Guy for taking me on as a graduate student and guiding me to complete this work. Without Guy and Umut's continuous support and mentorship I would not have been able to complete this work nor learned so much about research.

I thank Steve Brookes and Bob Harper for reading this dissertation and providing valuable feedback that improved the presentation.

I'm indebted to Matthew Fluet, Kanat Tangwongsan, and Duru Turkoglu for their valuable collaboration. Matthew's involvement was crucial for the formal setup and metatheoretical development, as well as the original $\Delta$ML implementation and evaluation. Kanat's expertise in self-adjusting computation was essential for the implementation and evaluation of traceable data types. Duru's work on motion simulation provided important feedback on $\Delta$ML to improve its usefulness.

I'm enormously grateful to Steve Brookes, John Reynolds, Frank Pfenning for their patient research advising when I was still getting my bearings. Steve was especially kind in supervising my early graduate studies and gracious when I turned my focus towards self-adjusting computation. My education was enriched by the POP group through the excellent courses taught by Steve Brookes, Karl Crary, Bob Harper, Frank Pfenning, and John Reynolds, as well as the countless extramural conversations with POP students.

I thank Alan Jeffrey for being my Alcatel-Lucent Bell Labs Graduate Fellowship mentor.

I thank Sharon Burks, Deb Cavlovich, Catherine Copetas, and Denny Marous for expediently handling many official matters.

Last but not least, I thank my family and friends for their enduring moral support. I would never have gotten here without my parents' support and unconditional love.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Thesis

Self-adjusting computation is a paradigm for programming incremental computations that efficiently respond to input changes by updating the output in time proportional to the changes in the structure of the computation. Running a self-adjusting program constructs a *trace* of the execution that captures the data and control dependencies of the computation and identifies opportunities for computation reuse. To adjust the program to different inputs, a *change-propagation* mechanism edits the trace with a combination of *adaptivity* and *computation memoization*. Adaptivity uses the data dependencies of the trace to find and re-execute subcomputations that are inconsistent relative to the input changes. Dually, computation memoization suspends re-execution upon encountering a reuse point and change-propagation can fast forward to the next inconsistency in the trace, thus reusing the unaffected portions of the computation from the previous run. Therefore, updating the computation takes time proportional to the changes in the structure of the trace.

Previous work on self-adjusting computation identified adaptivity [Acar et al., 2006c] and computation memoization [Acar et al., 2003] as complementary mechanisms for change-propagation in a general-purpose language, developed efficient implementation techniques for change-propagation based on traces (*a.k.a. memoized dynamic dependence graphs*) [Acar et al., 2006b], proposed modal and monadic language constructs for writing self-adjusting programs, devised a notion of *trace stability* to identify the applicability of self-adjusting computation [Acar, 2005]. Self-adjusting computation has been empirically validated in a number of application domains, including hardware verification [Santambrogio et al., 2007], invariant checking [Shankar and Bodik, 2007], motion simulation [Acar et al.,

2006d, 2008b], and machine learning [Acar et al., 2007c, 2008c], and other algorithmic problems [Acar et al., 2004, 2005, 2009]. This dissertation builds upon the preceding foundations to defend the following:

> **Thesis Statement.** High-level programming abstractions improve the experience of reading and writing, enable reasoning about the performance of, and optimize the efficiency of self-adjusting programs.

We substantiate this claim with the following work.

**Compilation.** We show that high-level language constructs are suitable for writing readable self-adjusting programs and can be compiled into low-level primitives. In particular, language constructs such as ML-style *modifiable references* and *memoizing functions* provide orthogonal, direct style mechanisms for identifying adaptivity and computation memoization. An *adaptive continuation-passing style* (ACPS) transformation compiles the direct style language constructs into a continuation-passing language with explicit support for change-propagation. In contrast to previous modal and monadic languages [Acar et al., 2006c, 2003, 2006b] for self-adjusting computation, high-level direct style constructs enable a natural programming style and compilation avoid the need to manually rewrite programs. This compilation approach is realized in the $\Delta$ML language—an extension to SML—and implemented as an extension to MLton with compiler and run-time support. $\Delta$ML is experimentally shown to be competitive with previous approaches to self-adjusting computation.

**Formal Reasoning.** We show that a high-level *cost semantics* captures the performance of a self-adjusting program and a theory of *trace distance* suffices for formal reasoning about the efficiency of change-propagation. The formal approach enables generalizing results from concrete runs to asymptotic bounds and compositional reasoning when combining self-adjusting programs. Formal reasoning complements empirical studies that are limited to particular input sizes and the analytical approach of trace stability that requires whole-program analysis.

**Extensibility.** Finally, we show that parameterizing the language by *traceable data types* (TDTs) and different forms of memoization broadens the class of programs amenable to self-adjusting computation.

Traceable data types generalize dependence-tracking from the level of references to a user-controlled level of granularity that can exploit problem-specific structure and thus improve the performance of change-propagation. In addition to single- and multi-write

references, we consider traceable data types for accumulating values from a commutative group, partitioning a totally-ordered (possibly-continuous) domain into intervals, queues, priority queues, and dictionaries.

We also consider *in-order* and *out-of-order* computation memoization as alternative mechanisms for reusing computation during change-propagation. In-order computation memoization only allows change-propagation to reuse subcomputations that occur in the same relative order across runs, whereas out-of-order computation memoization avoids this restriction. Out-of-order memoization broadens and simplifies the applicability of self-adjusting computation to various classes of programs, including lazy (call-by-need) computation with its unpredictable order of evaluation. Previous work on self-adjusting computation considered the efficient implementation of in-order computation memoization for modal and monadic languages [Acar et al., 2006c, 2003, 2006b], here we consider both in-order and out-of-order computation memoization for a continuation-passing style language and its interaction with traceable data types.

## 1.2 Structure of the Dissertation

In Chapter 2, we motivate the need for high-level programming abstractions for self-adjusting computation. We consider previous language proposals for self-adjusting computation and the need for compilation support. We give a high-level description of the cost semantics and trace distance as a means to formally reason about self-adjusting programs, in complement to empirical and analytic approaches. We describe the limitations of data dependence-tracking at the level of memory cells and argue for the usefulness of traceable data types. Finally, we compare in-order and out-of-order computation memoization as alternative means of obtaining efficient self-adjusting programs.

In Chapter 3, we present a family of $\lambda$-calculus source languages $\mathsf{Src}*$ with memoizing functions and parameterized by any number of traceable data types with direct style operations. The dynamic and cost semantics of $\mathsf{Src}*$ yields from-scratch evaluation of programs, and produces a tree-shaped trace of a program's execution with an associated cost of evaluation. A formal theory of trace distance quantifies the difference between two runs, which is asymptotically equivalent to the cost of change-propagation. A *local* trace distance, which corresponds to in-order computation memoization, yields the *edit distance* between traces. A *global* trace distance, which corresponds to out-of-order computation memoization, decomposes each run into *trace slices* (traces with holes) representing subcomputations which can be reordered and compared with local trace distance. Trace distance can be used to derive asymptotic bounds on the performance of change-propagation

and to reason compositionally about the combination of self-adjusting programs.

In Chapter 4, we present a family of $\lambda$-calculus target languages $\mathsf{Tgt}*$ with a low-level memoization construct and parameterized by any number of traceable data types with continuation-passing operations. The dynamic and cost semantics of $\mathsf{Tgt}*$ includes from-scratch evaluation as well as change-propagation for adjusting a previous run to different inputs, and produces list-shaped traces due to the continuation-passing discipline with an associated cost of evaluation. The $\mathsf{Tgt}*$ languages are parameterized by either in-order or out-of-order computation memoization, the former reuses tail segments of a previous run thus limiting reuse to the same execution order, while the latter reuses arbitrary trace segments of a previous run thus allowing reordering. Analogous to the $\mathsf{Src}*$ languages, a formal theory of trace distance quantifies the difference between two runs, with local and global trace distance corresponding to in-order and out-of-order memoization, respectively. We show that change-propagating a $\mathsf{Tgt}$ execution has a cost proportional to trace distance and produces a result consistent with a from-scratch execution.

In Chapter 5, we give an adaptive continuation-passing style translation from $\mathsf{Src}*$ to $\mathsf{Tgt}*$ parameterized by any number of TDTs which preserves the static semantics, and asymptotically preserves the dynamic and cost semantics and trace distance.

In Chapter 6, we discuss the $\Delta\mathsf{ML}$ language as an implementation of the compilation-based approach with compiler and run-time support. In Chapter 7, we present experimental evaluation of the compilation with modifiable references and other traceable data types, and revisit the examples of Chapter 2 with applications of the cost semantics and trace distance with in-order and out-of-order memoization.

In Chapter 8, we conclude with related and future work.

# Chapter 2

# Overview

In this chapter, we discuss previous work and motivate the need for high-level programming abstractions for self-adjusting computation. The overarching goal is to improve the programmability of self-adjusting computation, which we achieve through (1) the design a high-level source language for writing self-adjusting programs that respond efficiently to input changes and with syntax and semantics similar to a conventional language, (2) a formal cost semantics for the user to reason about the responsiveness of the program under input changes, and (3) an extensible language with support for dependence-tracking of different data types and alternative forms of memoization.

## 2.1 Compilation

### 2.1.1 Background

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 languages with a general-purpose change-propagation mechanism and specialized language constructs to identify data dependencies and opportunities for computation reuse. The proposed language constructs follow modal [Acar et al., 2006c, 2003] or monadic [Acar et al., 2006b,a] typing disciplines that stratify the program to distinguish data as stable or changeable—i.e., whether it can change across runs—and make the distinction explicit to the control structure. The languages were implemented as libraries in SML [Acar et al., 2006b] and Haskell [Carlsson, 2002], which required the programmer to obey a monadic discipline, explicitly delimit the scope of reads, program in

a destination-passing style, and apply memoization by manually declaring and hashing all free variables of memoized expressions. Thus an ordinary program can be systematically rewritten into a self-adjusting version using these languages, but the process is difficult and error-prone due to code restructuring imposed by the typing discipline and proper-usage restrictions of the implementation, some of which cannot be enforced statically. 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].

### 2.1.2 Foreground

The aforementioned library-based languages use modal and monadic constructs that make data dependencies explicit in the structure of the self-adjusting program. We propose a natural, high-level source language with annotations, which eliminates the burden of restructuring a program for self-adjusting computation, and employ a compilation-based approach that forgoes the manually-identified dependencies and instead automatically infers an approximation from the annotations. We formalize the compilation as a translation (Chapter 5) from a source language Src (Chapter 3) with direct style primitives to a self-adjusting target language Tgt (Chapter 4) with continuation-passing style (CPS) Appel [1991] primitives.

The Src language is a direct style $\lambda$-calculus with ML-style references (and other traceable data types described in Section 2.1) to identify changeable data and memoizing functions to identify opportunities for computation reuse. These constructs suffice to annotate an existing direct style program without code restructuring, which can be compiled into an equivalent self-adjusting Tgt version. The language is general-purpose (Turing-complete) and expressive: it allows writing both structured programs (e.g., iterative divide-and-conquer list algorithms) as well as unstructured programs (e.g., graph algorithms).

The target of compilation is the Tgt language, a $\lambda$-calculus with continuation-passing ML-style references and a memoization primitive for computation reuse across runs. The language is self-adjusting: its semantics includes evaluation and change-propagation that can be used to reduce expressions to values and adapt computations to input changes.

The compilation scheme consists of an *adaptive continuation-passing style* (ACPS) translation that infers the dependencies between changeable data and inserts memoization points at function call and return sites. The ACPS translation uses continuations to approximate the scope of use of changeable data, which, in the modal and monadic ap-

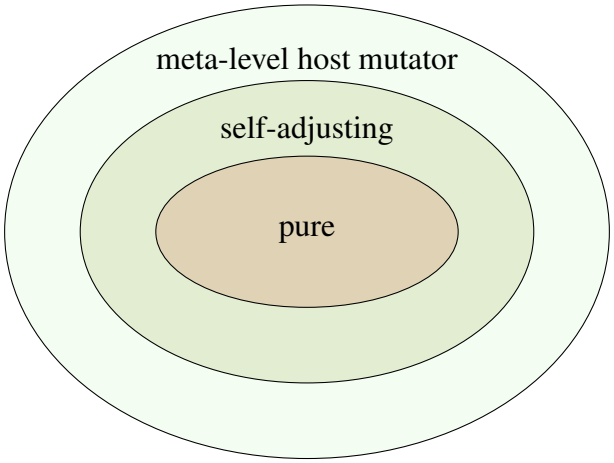



Figure 2.1: The relation between pure, self-adjusting, and impure host mutator programs.

proaches, was made explicit through programmer-supplied, fine-grained dependence information. 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 avoid unnecessary recomputation, the translation produces memoizing CPS functions as well as memoizing 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 memoizing 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.

### Writing Self-Adjusting Programs

In typical usage, a host *mutator* program contains a self-adjusting subprogram that manipulates changeable data, i.e., data that can be changed by external factors across runs. 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.

We present examples in ΔML, an SML extension for self-adjusting programs based on the compilation approach. A self-adjusting program consists of normal (pure, non-adaptive) functions and *adaptive functions* of type $\tau$ `-$>` $\tau$ declared with the **`afun`** and

**mfun** keywords; the latter declares a *memoizing* adaptive function. 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). Each self-adjusting program has a single entry point which itself is an adaptive function. Note that dropping the underlined code yields a non-self-adjusting SML version.

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.

Changeable data is manipulated through the *modifiable references* (abbreviated *modref*), which are ML-style references of type $\tau$ **modref** with support for adaptivity. Modifiables are manipulated with the direct style primitives:

```
put: α -$> α modref,
get: α modref -$> α,
set: α modref * α -$> unit.
```

The **put** primitive places a value into a modref, the **get** primitive returns the contents of a modref, and the **set** primitive updates the contents of a modref. Since the primitives have adaptive function types, they may only be used within a self-adjusting computation. In Section 2.3, we present other *traceable data types* for representing changeable data.

The host mutator may create, modify, and inspect changeable data via a collection of meta-level primitives, which we treat informally in this section. Figure 2.1 shows the relation between pure programs, self-adjusting programs, and meta-level host mutator programs.

**Example 1**
Figure 2.2 shows the ordinary (left), compilation-based self-adjusting (center), and monadic-based self-adjusting (right) [Acar et al., 2006b,a] versions of a function for partitioning a list with a predicate. The function takes a predicate `p` and a list `l` and returns two lists consisting of the elements of `l` for which `p` returns true and false, respectively.

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.

The compilation-based self-adjusting version is obtained by the following modifications. First, we make the list type changeable by placing the tail element in a modref,

```
datatype 'a list =
 nil | ::  of 'a * 'a list

fun partition p l = let
 fun loop l =
  case l of
    nil =>
    (nil,nil)
  | h::t => let
    val (a,b) = loop t
    in
     if p h then
      (h::a,b)
     else
      (a,h::b)
    end
 in loop l end
```

```
datatype 'a cell =
 nil | ::  of 'a * 'a list
withtype 'a list =
 'a cell modref

afun partition p l = let
 mfun loop l =
  case get $ l of
    nil =>
    (put $ nil,put $ nil)
  | h::t => let
    val (a,b) = loop $ t
    in
     if p h then
      (put $ (h::a),b)
     else
      (a,put $ (h::b))
    end
 in loop $ l end
```

```
datatype 'a cell =
 nil | ::  of 'a * 'a list
withtype 'a list =
 'a cell modref

fun partition p l = let
 fun loop l = read(l, fn l =>
  case l of
    nil =>
    write(mod(write(nil),mod(write(nil))
  | h::t => memo (h,t) (fn () => let
    val ab = mod(loop t)
    in read(ab, fn (a,b) =>
     if p h then
      write(mod(write(h::a),b)
     else
      write(a,mod(write(h::b)))
    end ))
 in loop l end
```

Figure 2.2: The partition function: ordinary (left), compilation-based self-adjusting (center), and monadic-based self-adjusting (right) versions.

which allows the mutator to modify lists by inserting/deleting elements. Second, we change the `partition` function to operate on modref-based lists by inserting a `get` operation when destructing a list and inserting a `put` operation when constructing a list. Third, 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`.

The monadic-based self-adjusting version is obtained by using the following primitives. Due to a destination-passing discipline, modrefs are allocated with the `mod` primitive before they are initialized with `write`. Moreover, all changeable data should be dereferenced with the monadic `read`, which explicitly delimits the scope of the operation and must end in a `write`. The `memo` construct requires all free variables of memoized expressions to be manually identified and hashed. Note that the monadic type discipline requires significantly more code restructuring, even for this simple function. The significance of the changes is best measured by considering the differences in the abstract syntax trees, not the differences in the lexical tokens.

**Compiling Self-Adjusting Programs**

Compilation translates a Src self-adjusting program into an intermediate language that generalizes the previously proposed monadic primitives. The adaptive continuation-passing style (ACPS) transformation serves to automatically infer a conservative approximation of the dynamic data dependencies. To prevent the inferred, approximate dependencies from degrading the performance of change-propagation, we generate memoizing 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.

To compile a Src self-adjusting program, we translate adaptive functions into equivalent CPS functions and memoize them if so indicated by the **mfun** keyword. 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. Since the arguments of a CPS function include its continuation and memoizing a function requires that the current arguments must match the arguments of a previous call, memoizing functions in CPS requires some care. Memoizing on the continuation decreases the effectiveness of memoization because a function call cannot match when the continuations differ. We address this problem by translating memoizing adaptive functions to CPS functions that treat their continuations as changeable data. This allows the memoizing 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 target-level primitive **memo** : $(\alpha$ -> **res**$)$ -> $\alpha$ -> **res**. Effectively, **memo** `(f x)` checks whether `f x` (for the same `f` and `x`) was executed in the previous run. If so, there is a memoization hit; if not, `f` is invoked with `x`. In either case, the fact that **memo** `(f x)` was executed is recorded for the next run.

The intermediate target language provides modifiable references $\tau$ **modref** with continuation-passing primitives:

```
putk: α -> α  modref cont,
getk: α  modref -> α cont,
setk: α  modref * α -> unit cont,
```

where $\tau$ **cont** is $(\tau$ -> **res**$)$ -> **res** for some abstract result type **res**. The **putk**

```
datatype 'a cell = nil | ::  of 'a * 'a list
withtype 'a list = 'a cell modref
fun partition p ml k = let
   fun loop (ml, k) = getk ml (fn l =>
     case l of
       nil => putk nil (fn ma =>
              putk nil (fn mb => k (ma, mb)))
     | h::mt => let
       val k' = fn (a,b) =>
         if p h then
           putk (h::a) (fn ma => k (ma,b))
         else
           putk (cons(h,b)) (fn mb => k (a,mb))
       in loop_memo mt k' end)
   and loop_memo ml k = let
     val k_memo = fn r => memo k r
     in
       putk k_memo (fn mk => let
             val k' = fn r => getk mk (fn k => k r)
             in memo loop (ml, k') end)
     end
   in loop_memo ml k end
```

Figure 2.3: The partition function compiled.

primitive initializes a new modifiable with a value and passes the reference to the continuation; this primitive can reuse modifiables written in the previous run of the program, which is essential for efficient change-propagation. The **getk** primitive dereferences a modifiable and passes the contents to the continuation. The **setk** primitive updates the modifiable and passes a unit value to the continuation. Src-level modrefs are compiled into Tgt-level modrefs by structurally translating the **modref** type and each **put**, **get**, and **set** primitive to the corresponding continuation-passing version.

**Example 2**

Figure 2.3 shows the compiled code for `partition`. To obtain this code, we translate the functions `partition` and `loop` and adaptive applications into CPS, CPS-convert **put/get** into **putk/getk** with an explicit continuation, and memoize `loop` as `loop_memo`. To do so, `loop_memo` memoizes its continuation and writes it into a modifiable. It then calls `loop` with a continuation that, when invoked, reads and invokes the original continuation. Since the application of `loop` is memoizing, 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 memoizing the continuation modifiable chosen for each argument modref-based list.

**Implementation**

We validate the feasibility of the compilation-based approach proposal with the ΔML language and its implementation. The ΔML extends Standard ML with the Src-level primitives for self-adjusting computation. The implementation is realized as a transformation pass in the whole-program optimizing compiler MLton [MLt] together with a run-time library that provides Tgt-level functionality for self-adjusting computation.

We perform an experimental analysis by compiling self-adjusting versions of a number of (annotated) benchmarks. Our experiments 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 versions). The experiments indicate that the compilation approach is asymptotically comparable to the previous evaluation of self-adjusting computation based on manual rewriting using a monadic library [Acar et al., 2006b,a].

### 2.1.3 Presentation

We present the Src language in Chapter 3, the Tgt language in Chapter 4, and the formal ACPS translation in Chapter 5. We present the ΔML language implementation in Chapter 6 and an experimental evaluation in Chapter 7.

## 2.2 Formal Reasoning: Cost Semantics and Trace Distance

### 2.2.1 Background

The applicability of self-adjusting computation has been demonstrated through experiments [Acar et al., 2006b] and semi-formal algorithmic [Acar et al., 2006c, 2003, Acar, 2005] analysis of the modal and monadic approaches. Previous applications of the approach often only give experimental results to illustrate performance gains [Acar et al., 2006b,d, 2008b]. Giving asymptotic bounds requires integrating change-propagation into the algorithm by considering a low-level machine model akin to the RAM model [Acar et al., 2004], thus the bounds derived only apply indirectly to the code as written. Source-level reasoning about the overhead of self-adjusting programs and the efficiency of change-

propagation is difficult due to the complex semantics of change-propagation and the indirect nature of previously proposed language techniques [Acar et al., 2006b].

To see the first difficulty, consider executing program with some input and later changing the input. As a self-adjusting program executes, information about the execution (such as data and control dependencies) is recorded. After the input is changed, change-propagation updates the output by using the recorded dependence information to find the parts of the computation affected by the change and updating stale computation by re-executing code. When re-executing code, change-propagation may reuse previous computations with a form of computation memoization. Since change-propagation selectively re-executes parts of the code snippets under a new program state but reuses other parts of the execution, it is hard to reason about its complexity. In particular, the user may need to reason about the contexts in which sub-expressions are evaluated to distinguish changed and unchanged data, which can be difficult even with limited forms of computation reuse techniques such as lazy evaluation [Wadler and Hughes, 1987, Sands, 1990a,b].

The second difficulty arises from the nature of the modal and monadic primitives. These approaches require the programmer to explicitly allocate locations for changeable data prior to initialization and identify their data dependencies, delimit the static scope of the operation that reads changeable data and identify their control dependencies, and apply memoization by carefully considering whether the data dependencies are local or non-local [Acar et al., 2006b]. Depending on the choice of the scope for the primitives and the use of memoization, the programmer may observe drastically different performance by even a slight change to the code.

### 2.2.2 Foreground

We propose formal reasoning for compilation-based self-adjusting programs by means of a *cost semantics* that captures an abstract cost of evaluation of from-scratch runs and change-propagation for updating runs, and a formal theory of *trace distance* that quantifies the dissimilarities between two runs under (possibly) different stores and coincides with the cost of change-propagation. By defining a cost semantics and trace distance for both the Src and Tgt languages and showing they are related by the ACPS translation, we provide realistic source-level reasoning techniques that guarantee performance. In particular, we show that the translation (1) preserves the extensional (result) semantics of the source programs and intensional (cost) semantics of from-scratch runs, and (2) ensures that change-propagation between two evaluations takes time bounded by their relative distance.

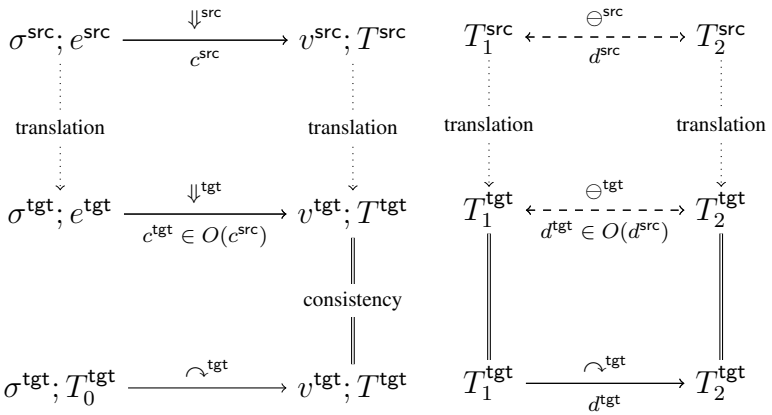



Figure 2.4: The left diagram illustrates the correspondence between the source and target from-scratch runs and the consistency of change-propagation in the target. The right diagram illustrates the correspondence between distance in the source and target, and the time for change-propagation in the target.

Unlike previous techniques, this semantics-based approach enables composing and generalizing the results from concrete evaluations to asymptotic bounds. We develop a notion of trace contexts, which are traces with holes that can be filled with other traces. We prove that, under certain conditions, distance is additive under substitution: the distance between traces obtained via substitution into two contexts is the same as the distance between the substituted traces plus the distance between the contexts. In particular, the approach enables programmer to quantify how effective change-propagation can be with a given input for a given change. Trace contexts allow bounds on concrete evaluations to be generalized to asymptotic bounds for any input. Furthermore, trace contexts enable compositional reasoning such that bounds for complex programs can be obtained by combining the bounds for its individual components.

Figure 2.4 illustrates our approach. The dynamic and cost semantics evaluates a Src expression ($e^{\mathsf{src}}$) in the context of a store ($\sigma^{\mathsf{src}}$) and yields a value ($v^{\mathsf{src}}$), a trace of the evaluation ($T^{\mathsf{src}}$), and a step count ($c^{\mathsf{src}}$). We quantify the dissimilarity between evaluations of Src programs with a trace distance ($T_1^{\mathsf{src}} \ominus^{\mathsf{src}} T_2^{\mathsf{src}} = d^{\mathsf{src}}$ states that the distance between the traces $T_1^{\mathsf{src}}$ and $T_2^{\mathsf{src}}$ is $d^{\mathsf{src}}$). Intuitively, the trace distance measures the *edit distance* between evaluations. To give an effective distance, we show that it suffices to record function calls and store operations in the trace. We don't record complete stores or evaluation contexts. We use evaluation contexts, which describe how results are used, to prove our meta-theoretic results, but they are *not* necessary for source-level reasoning.

Since our language is stateful, recording complete stores would lead to a distance measure that overestimates distance significantly; requiring evaluation contexts would make reasoning cumbersome.

Evaluation of a Tgt expression ($e^{\mathsf{tgt}}$) takes place in the context of a store ($\sigma^{\mathsf{tgt}}$) and yields a value ($v^{\mathsf{tgt}}$), a trace ($T^{\mathsf{tgt}}$), and a step count ($c^{\mathsf{src}}$). The semantics includes a change-propagation mechanism ($\curvearrowright^{\mathsf{tgt}}$) that can replay a trace from a previous run (e.g., $T_0^{\mathsf{tgt}}$) in a store ($\sigma^{\mathsf{tgt}}$) to produce a value and a trace that are consistent with a from-scratch execution, while reusing the work from the initial trace ($T_0^{\mathsf{tgt}}$). We give a cost semantics for the Tgt language that counts steps of evaluation, but not steps of change-propagation. As in Src, we define a Tgt distance for traces ($\ominus^{\mathsf{tgt}}$) and bound the time for change-propagation by the distance between the computation traces before and after propagation.

We prove the following properties of the translation (*cf.*, Figure 2.4).

- **Extensional semantics.** The translation preserves the evaluation of Src programs (top left square). If $e^{\mathsf{src}}$ evaluates to $v^{\mathsf{src}}$ in store $\sigma^{\mathsf{src}}$, then the translated expression $e^{\mathsf{tgt}}$ evaluates to the translated value $v^{\mathsf{tgt}}$ in the translated store $\sigma^{\mathsf{tgt}}$.

- **Intensional semantics.** The translation preserves the asymptotic cost of from-scratch runs (top left square). If evaluation takes $c$ steps in Src, then running the translated expressions takes $O(c)$ steps in Tgt.

- **Consistency of change-propagation.** Change-propagation in Tgt preserves the extensional semantics of from-scratch runs (bottom left square). If $T^{\mathsf{tgt}}$ is the trace from evaluating a Tgt expression in some store (i.e., $\sigma_0; e \Downarrow^{\mathsf{tgt}} v_0; T_0$), then change-propagating the trace $T_0$ under another store $\sigma$ yields the same value and trace as running $e$ from-scratch in $\sigma$.

- **Trace distance.** Translated programs have asymptotically the same trace distance as their source (top right square). If two Src derivations have distance $d$, then the distance between their Tgt translations is $O(d)$.

- **Change-propagation time.** Time for change-propagation in Tgt coincides asymptotically with Src trace distance (right diagram).

To prove the first two properties, we generalize a folklore theorem about CPS to show that an ACPS-compiled program preserves the evaluation and asymptotic complexity of a Src program. The ACPS translation is more complicated than the standard translation because it threads continuations through the store. We give a simple, structural proof of the consistency of change-propagation by casting it as a full replay mechanism. We prove

the fourth property by establishing a relation between the traces of Src and Tgt programs. This property also bounds the time for change-propagation (the last property) by showing that change-propagation in Tgt takes time proportional to the Tgt distance.

### Trace Distance

We develop techniques for reasoning about the effectiveness of change-propagation by means of *trace distance*. A trace represents the execution of a program and is an abstraction of the program's evaluation derivation restricted to memoizing function applications and store operations. Here, we discuss trace distance in terms of the dissimilarities between evaluation derivations to explain which parts of the derivation are essential. The idea is to compare the evaluation derivations of a program with two different, typically similar, inputs and compute the *edit distance* between these derivations. Our results guarantee that compiled programs can respond to a change in time proportional to the distance between the corresponding traces.

**Example 3**
Consider the following standard implementation of a `mapA` function that applies a function `i2c` pointwise to map integers to alphabet characters. As a compilation-based self-adjusting program, we define linked lists with modifiable reference pointers and memoize the `mapA` function by declaring it with **mfun**.

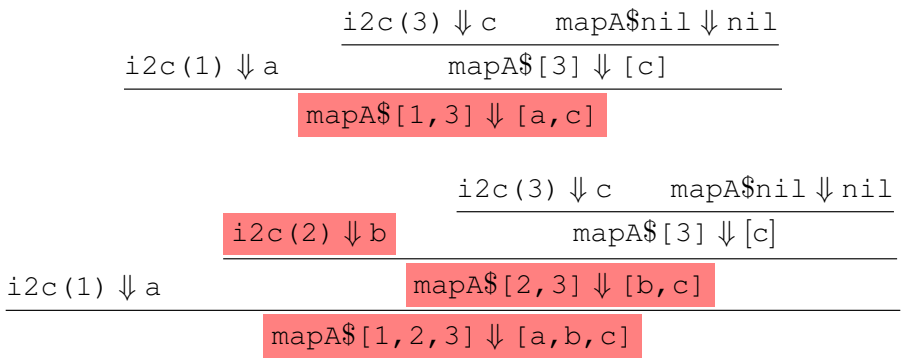
```
datatype 'a cell = nil | ::  of 'a * 'a list
withtype 'a list = 'a cell modref

mfun mapA l =
  case get $ l of
    nil => put $ nil
  | h::t =>
    let t' = mapA $ t
    in put $ ((i2c h)::t') end
```

Running `mapA` with initial input `[1,3]` produces the result `[a,c]` in linear time in the length of the list. The input can be changed to `[1,2,3]` by splicing a new cons cell into the first tail pointer. After this change, we can run change change-propagation to update the output to `[a,b,c]`. Self-adjusting computation tracks dependence information that enables change-propagation to update the initial execution and output of `mapA` efficiently. Intuitively, change-propagation could translate the new integer into a letter

and insert the new element in the right position in the output in constant time. Trace distance facilitates source-level formal reasoning about the efficiency of change-propagation in terms of the similarities between the two runs, instead of target-level reasoning about the compiled program and its operational behavior.

**Distance between pure runs.** A first cut at trace distance is to compare the two runs of a purely functional program, i.e., without the use of references. Since a purely functional program explicitly passes data through the program, any changes to the input as well as subsequent updates to the output become apparent in the evaluation derivation. This prevents the computation change from beihng localized at the point where the input change affects its dependencies. Therefore, the two runs would differ wherever the input changes and any subsequent changes are passed through the program, which is an overly pessimistic approximation and would be problematic for compositional reasoning to obtain asymptotic results.

**Example 4**
The evaluation derivations of a purely functional version of `mapA` with inputs `[1,3]` and `[1,2,3]` are shown below, using the notation $e \Downarrow v'$ for evaluating an expression $e$ to value $v'$. The similar computations are the conversion of the integers `1` and `3` to characters and the application of `mapA` to the suffix list `[3]`. The differences between the runs are highlighted. The obvious difference is the conversion of the new element `2`. However, due to the purely functional evaluation, the new element shows up in any list that contains the new element. In this example, the application `mapA $ [1,2,3]` also differs. If the list had a longer prefix, say $[n_1, \ldots, n_k, 1, 2, 3]$, then all prefix applications `mapA` $[n_1, \ldots, n_k, 1, 2, 3]$, `mapA` $[n_2, \ldots, n_k, 1, 2, 3]$, $\ldots$, `mapA` $[n_k, 1, 2, 3]$ would also differ. Therefore the difference between runs would be linear in the length of the list—far larger than the constant that we expect.



**Distance between impure runs with stores.** To make the data and computation differences between runs only appear locally, self-adjusting computation requires changeable data to be manipulated via modifiable references. However, directly comparing evaluation derivations with explicit stores yields a distance that is too coarse. Since inputs are represented in the store and the store is threaded through the derivation, stores won't match and all derivation steps differ.

**Example 5**
The evaluation derivations of `mapA` with explicit use of references are shown below, using the notation $\sigma \; ; \; e \; \Downarrow \; \sigma' \; ; \; v'$ for the evaluation of expression $e$ under store $\sigma$ to value $v'$ under updated store $\sigma'$. For brevity, we show the `mapA` function applications, but omit reference operations and `i2c` applications. All derivation steps differ between the runs for `[1,3]` and `[1,2,3]` because they all contain different highlighted stores. Therefore the difference between runs would again be linear in the size of the input.

$$\dfrac{\dfrac{\dfrac{}{\sigma_{13};\texttt{mapA\$}\ell \Downarrow \sigma_{13}[\ell' = \texttt{nil}];\ell'}}{\sigma_{13};\texttt{mapA\$}\ell_3 \Downarrow \sigma_{13}[\ell' = \texttt{nil}][\ell_c = \texttt{c::}\ell'];\ell_c}}{\sigma_{13};\texttt{mapA\$}\ell_1 \Downarrow \sigma_{13}[\ell' = \texttt{nil}][\ell_c = \texttt{c::}\ell'][\ell_a = \texttt{a::}\ell_c];\ell_a}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma_{123};\texttt{mapA\$}\ell \Downarrow \sigma_{123}[\ell' = \texttt{nil}];\ell'}}{\sigma_{123};\texttt{mapA\$}\ell_3 \Downarrow \sigma_{123}[\ell' = \texttt{nil}][\ell_c = \texttt{c::}\ell'];\ell_c}}{\sigma_{123};\texttt{mapA\$}\ell_2 \Downarrow \sigma_{123}[\ell' = \texttt{nil}][\ell_c = \texttt{c::}\ell'][\ell_b = \texttt{b::}\ell_c];\ell_b}}{\sigma_{123};\texttt{mapA\$}\ell_1 \Downarrow \sigma_{123}[\ell' = \texttt{nil}][\ell_c = \texttt{c::}\ell'][\ell_b = \texttt{b::}\ell_c][\ell_a = \texttt{a::}\ell_b];\ell_a}$$

**Distance between impure runs with store operations.** To recognize the similarity between the derivations and restrict the dissimilarities to computation changes, we exclude the store from the derivations and include the store operations instead. Of course, it is possible to make the "distance" between derivations arbitrarily small when we suppress arbitrary parts of the derivation. We prove that this distance is in fact realistic by showing that the translation preserves distance asymptotically between the Src and Tgt languages, and that the Tgt language has provably efficient change-propagation, i.e., that change-propagation takes time proportional to the distance.

**Example 6**
The evaluation derivations of `mapA` with store operations are shown below, omitting the store and representing store operations explicitly. The notation $\ell \xrightarrow{\texttt{get}} v$ represents fetching

value $v$ from location $\ell$, and $\ell \overset{\texttt{put}}{\leftarrow} v$ represents allocating and initializing $\ell$ with value $v$. The only differences between the derivations (highlighted) are the operations on the new element `2`. Note that the difference remains the same even if we add more elements to these lists (e.g., `[...,0,1,3,4,...]` and `[...,0,1,2,3,4,...]`). Later, we show that the difference between two evaluations of `mapA` that differ by one element remains the same as shown in this figure regardless of the input size.

$$\dfrac{\ell_1 \overset{\texttt{get}}{\rightarrow} \texttt{1}::\ell_3 \qquad \dfrac{\ell_3 \overset{\texttt{get}}{\rightarrow} \texttt{3}::\ell \qquad \dfrac{\ell \overset{\texttt{get}}{\rightarrow} \texttt{nil} \quad \ell' \overset{\texttt{put}}{\leftarrow} \texttt{nil}}{\texttt{mapA}\$\ell \Downarrow \ell'} \qquad \ell_3 \overset{\texttt{put}}{\leftarrow} \texttt{c}::\ell'}{\texttt{mapA}\$\ell_3 \Downarrow \ell_c} \qquad \ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}::\ell_c}{\texttt{mapA}\$\ell_1 \Downarrow \ell_a}$$

$$\dfrac{\ell_1 \overset{\texttt{get}}{\rightarrow} \texttt{1}::\ell_2 \qquad \dfrac{\ell_2 \overset{\texttt{get}}{\rightarrow} \texttt{2}::\ell_3 \qquad \dfrac{\ell_3 \overset{\texttt{get}}{\rightarrow} \texttt{3}::\ell \qquad \dfrac{\ell \overset{\texttt{get}}{\rightarrow} \texttt{nil} \quad \ell' \overset{\texttt{put}}{\leftarrow} \texttt{nil}}{\texttt{mapA}\$\ell \Downarrow \ell'} \qquad \ell_c \overset{\texttt{put}}{\leftarrow} \texttt{c}::\ell'}{\texttt{mapA}\$\ell_3 \Downarrow \ell_c} \qquad \ell_b \overset{\texttt{put}}{\leftarrow} \texttt{b}::\ell_c}{\texttt{mapA}\$\ell_2 \Downarrow \ell_b} \qquad \ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}::\ell_b}{\texttt{mapA}\$\ell_1 \Downarrow \ell_a}$$

There are several properties of trace distance that we would like to note. First, trace distance is defined relationally, which allows the approach to apply to any concrete implementation technique consistent with the semantics: our main theorems state that our translation can match any Src distance computed relationally. Second, trace distance is sensitive to the choice of locations because trace distance compares concrete evaluations. Previous implementations of self-adjusting computations can often choose locations to minimize the trace distance. Since our theorems can match any distance computed, they apply to existing implementations. The problem of whether an implementation can efficiently achieve the minimum possible distance is not well understood: this is undecidable in general but these impossibility results typically involve programs that don't arise in practice.

### Trace Contexts

To reason about the asymptotic complexity bounds for distance, and hence change-propagation, we need to compute distance for all input sizes under a particular class of changes. This is difficult with the distance described above because it requires two concrete executions. To facilitate asymptotic analysis, we use trace contexts (i.e., traces with

holes). As with trace distance, we use derivations and derivation contexts (i.e., contexts with holes) to explain the main ideas. We write $\nabla^{e \Downarrow v}$ for a context hole that expects an evaluation of $e \Downarrow v$. We can obtain a derivation from a derivation context by substituting a derivation for a hole.

Let $\mathscr{D}_1[\nabla]$ and $\mathscr{D}_2[\nabla]$ be derivation contexts and let $D_1$ and $D_2$ be derivations. We prove that the distance between $\mathscr{D}_1[D_1]$ and $\mathscr{D}_2[D_2]$ is the sum of the distances between $\mathscr{D}_1[\nabla]$ and $\mathscr{D}_2[\nabla]$ and between $D_1$ and $D_2$, for suitably-shaped contexts. Note that not all substitutions yield well-formed derivations; in particular, the choice of locations needs to be consistent. This result enables generalizing concrete distances to arbitrary inputs and deriving asymptotic complexity bounds, which is generally difficult with concrete cost semantics [Sands, 1990a,b, Sansom and Jones, 1995, Blelloch and Greiner, 1995, 1996, Spoonhower et al., 2008].

**Example 7**
Consider the evaluation derivation of `mapA`, shown below, applied to the integer list $[n_1, \ldots, n_k]@\square$ where $\square$ represents an unspecified list. In the derivation $\ell_i$ (resp. $\ell'_i$) denotes the reference to the cons cell containing input $n_i$ (resp. output for $c_i$), and each $n_i$ is mapped to $c_i$. Given this derivation context, we can substitute the list `[1,3]` for $\square$ and obtain the derivation for that input by substituting the derivation of `[1,3]` in place of the hole. Generalizing and combining the above two analyses we can show that the distance between derivations of `mapA` with any pair of inputs that differ by one element is constant.

$$\dfrac{\ell_1 \overset{\texttt{get}}{\hookrightarrow} n_1 :: \ell_2 \qquad \dfrac{\dfrac{\ell_k \overset{\texttt{get}}{\hookrightarrow} n_k :: \ell_\square \qquad \nabla^{\texttt{mapA}\$\ell_\square \Downarrow \ell'_\square} \qquad \ell'_k \overset{\texttt{put}}{\hookleftarrow} c_k :: \ell'_\square}{\vdots}}{\texttt{mapA}\$\ell_2 \Downarrow \ell'_2} \qquad \ell'_1 \overset{\texttt{put}}{\hookleftarrow} c_1 :: \ell'_2}{\texttt{mapA}\$\ell_1 \Downarrow \ell'_1}$$

### 2.2.3 Presentation

In Chapters 3 and 4, we present cost semantics and trace distance for the Src and Tgt languages, respectively, and show the consistency of change-propagation. In Chapter 5, we relate the Src and Tgt semantics and trace distance.

## 2.3 Extensible Adaptivity: Traceable Data Types

### 2.3.1 Background

Modifiable references (introduced in Section 2.1) are sufficient to capture data and control dependencies, and allow the mutator to modify the inputs and force change-propagation to update a computation. Single-write modrefs [Acar et al., 2006c] can be created by either the core- or meta-level program but can only be overwritten by the meta-level mutator; the single-write restriction suffices to make purely functional programs self-adjusting by storing program data in modrefs. Multi-write modrefs [Acar et al., 2008a] further allow the core-level program to overwrite the contents of a modref.

Intuitively, creating a modref with contents $v$ and then dereferencing the modref multiple times yields a trace of the form $\ell \overset{\texttt{put}}{\leftarrow} v \cdot \ell \overset{\texttt{get}}{\rightarrow} v \cdots \ell \overset{\texttt{get}}{\rightarrow} v$. If the mutator changes the modref with a different initial value $v'$, the creation action becomes $\ell \overset{\texttt{put}}{\leftarrow} v'$ and the subsequent dereferences correspond to the computation that must be re-executed, so change-propagation adjusts the trace to $\ell \overset{\texttt{put}}{\leftarrow} v' \cdot \ell \overset{\texttt{get}}{\rightarrow} v' \cdots \ell \overset{\texttt{get}}{\rightarrow} v'$. Similarly, if an update action $\ell \overset{\texttt{set}}{\leftarrow} v$ changes to $\ell \overset{\texttt{set}}{\leftarrow} v'$, the subsequent dereferences must be adjusted to reflect the different contents of the location.

Tracking the computation at the granularity of single- or multi-write reference operations is fine-grained enough to express any purely functional or imperative program as a self-adjusting version. Tracking individual reference operations, however, can be too fine-grained because it limits the time and space efficiency of change-propagation:

- there is a considerable time overhead for tracking every memory operation in a from-scratch run and maintaining the fine-grained dependence information during change-propagation,
- the size of the trace is proportional to the number of memory operations and their fine-grained dependence information, limiting the scalability to large inputs,
- implementing a data type with modifiable references can have asymptotically suboptimal change-propagation time because it cannot take advantage of problem-specific structure: updates can cause many changes to the internal data type implementation even when the changes that propagate beyond the interface are small—i.e., a computation can be stable with respect to operations of the data type, but not stable with respect to individual cell accesses.

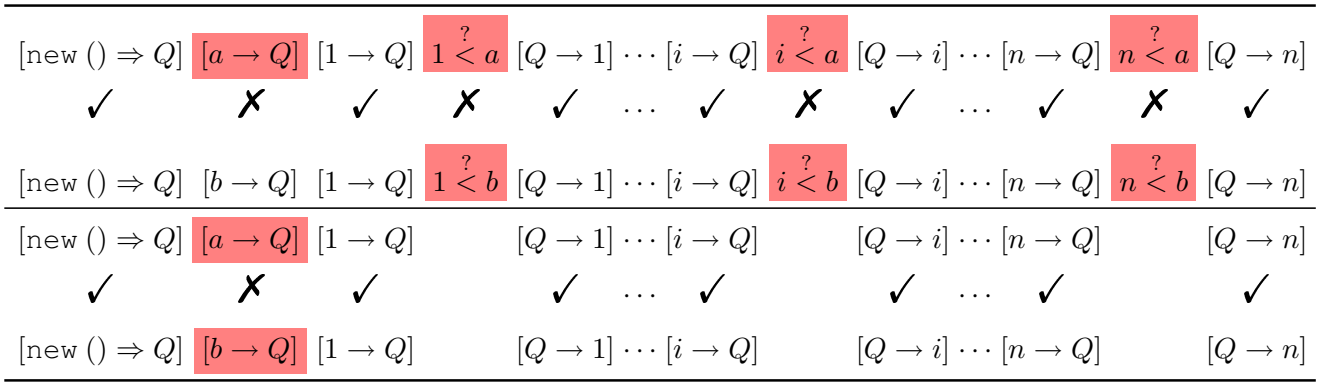



Figure 2.5: Two pairs of traces of a hypothetical program $P$ at the level of queue operations and comparisons (top) and at the level of abstract queue operations (bottom). Each pair corresponds to a run of $P$ with inputs $[a, 1, 2, \ldots, n]$ and $[b, 1, 2, \ldots, n]$.

**Example 8 (Priority Queue)**
Consider a self-adjusting priority queue with the following signature:

```
signature PRIORITY_QUEUE = sig
  type ('k,'v) t
  val new:  ('k * 'k -> order) -$> ('k,'v) t
  val insert:  ('k,'v) t * 'k * 'v -$> unit
  val delMin:  ('k,'v) t -$> ('k * 'v) option
end
```

The `new` operation takes a comparison function on keys and returns an empty priority queue, an operation to `insert` a key and a value into a priority queue, and a `delMin` operation to remove the element with the least priority. An ordinary heap or treap implementation can be instrumented with modifiable references to obtain a self-adjusting priority queue. In particular, a self-adjusting heap structure `PQ_Modifiable` of signature `PRIORITY_QUEUE` can be obtained by using modifiable references for the child pointers of each heap node. In such a modifiable-based implementation, the trace records every child pointer access and the associated comparison between priorities, which leads to a large trace overhead for from-scratch runs and suboptimal change-propagation performance due to fine-grained dependence-tracking.

We present a worst-case example where the modifiable-based implementation yields inefficient change-propagation. Consider a self-adjusting program $P$ that takes as input

an integer list, creates an empty queue and inserts the first element of the list into the priority queue. Starting with the second element, the program then inserts each element into the priority queue using the element both as a priority and as a value and removes the minimum element with `delMin`.

Consider two runs of $P$ with inputs $[a, 1, 2, \ldots, n]$ and $[b, 1, 2, \ldots, n]$, where $a, b > n$ and $a \neq b$. Figure 2.5 (top) shows traces, represented abstractly, for an execution of $P$ with a modifiable-based priority queue that must track every comparison. We write $[\texttt{new}\ () \Rightarrow Q]$ for creating an empty queue $Q$, $[i \rightarrow Q]$ for the operation $\texttt{insert}(Q, i, i)$, $[Q \rightarrow i]$ for the operation $\texttt{delMin}(Q)$ that returns $i$ as the minimum priority, and $i \stackrel{?}{<} j$ for a comparison of the keys $i$ and $j$ in the priority queue. In the figure, we use ✓ and ✗ to indicate the operations of the trace that match and that do not match, respectively.

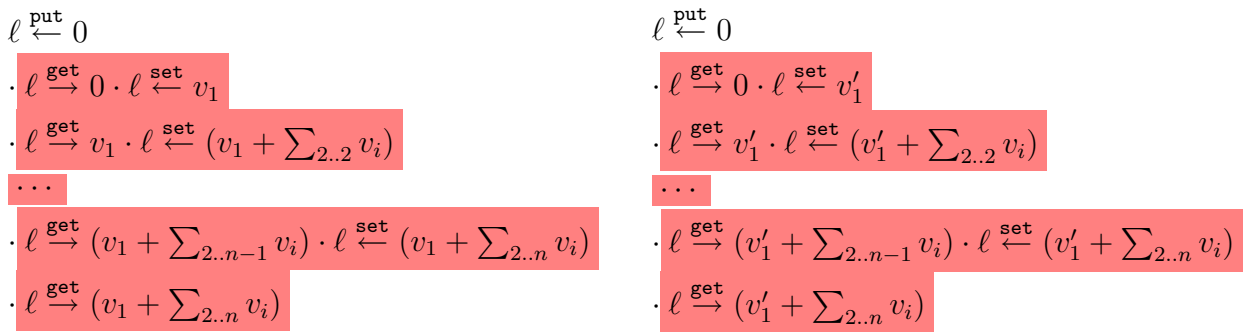
Every comparison in the first (resp. second) trace has the form $i \stackrel{?}{<} a$ (resp. $i \stackrel{?}{<} b$) for $1 \leq i \leq n$. Therefore no pair of comparisons match between the two runs and the difference between the two traces is $\Theta(n)$. Consequently, starting with the first input running the program $P$, changing the input by replacing $a$ by $b$, and performing change-propagation would require at least linear time to update the output.

This argument extends to any priority queue implementation based on modifiable references, because every time a new key $i$ is inserted, the priority queue contains only the element with the largest key (either $a$ or $b$) and thus a comparison with $i$ must be performed to determine the minimum priority required by the next operation. It is thus not possible to use change-propagation based on self-adjusting computation with modifiable references to update the output in less than linear time.

If the trace only recorded the priority queue operations but not the internal comparisons, the traces would be very similar as shown in Figure 2.5 (bottom) and only differ by the initial insertion. This example shows that recording dependencies at the level of priority queue operations, instead of the internal comparisons performed by the priority queue operations, yields smaller traces and results in fewer changes for change-propagation to handle.

**Example 9 (Accumulator)**
Suppose we want to add the elements of a list $[v_1, \ldots, v_n]$ using a multi-write modref to accumulate the prefix sums. We can initialize a modref with $0$ then traverse the list and add each member of the list to the accumulator. Adding a value $v$ to the accumulator is the composite operation of dereferencing the current accumulator value $v_{acc}$ followed by storing the updated sum $v'_{acc} = v_{acc} + v$. Extensionally, adding to the accumulator only requires the modref $\ell$ and value $v$ to be added, but the intensional implementation using modrefs makes the addition operation sensitive to the intermediate accumulator value $v_{acc}$. Finally, the total sum is obtained by dereferencing the modifiable. If we replace the first element of the list $v_1$ with a different value $v'_1$, then the inconsistent actions of trace must be updated.

Comparing the two traces restricted the trace to the accumulator actions:




Since $v_1$ and $v'_1$ are different, each intermediate value of the accumulator $v'_1 + \sum v_i$ also differs from $v_1 + \sum v_i$ in the previous run, so change-propagation takes linear time to update the trace.

In the case of integer addition—and more generally for any commutative group—we can achieve constant-time change-propagation by taking advantage of the inverse operation. We can replace the use of a multi-write modref with a specialized *accumulator modref* that has operations (an associated trace actions) to create an empty accumulator ($\ell \overset{\texttt{acc}}{\leftarrow} v$), add a value to the accumulator ($\ell \overset{\texttt{add}}{\leftarrow} v$) that is not sensitive to the intermediate

sums, and read the total ($\ell \overset{\texttt{total}}{\rightarrow} v$). The two runs yield the similar traces:

$$
\begin{array}{ll}
\ell \overset{\texttt{acc}}{\leftarrow} 0 & \ell \overset{\texttt{acc}}{\leftarrow} 0 \\
\cdot\, \ell \overset{\texttt{add}}{\leftarrow} v_1 & \cdot\, \ell \overset{\texttt{add}}{\leftarrow} v_1' \\
\cdot\, \ell \overset{\texttt{add}}{\leftarrow} v_2 & \cdot\, \ell \overset{\texttt{add}}{\leftarrow} v_2 \\
\cdots & \cdots \\
\cdot\, \ell \overset{\texttt{add}}{\leftarrow} v_n & \cdot\, \ell \overset{\texttt{add}}{\leftarrow} v_n \\
\cdot\, \ell \overset{\texttt{total}}{\rightarrow} (v_1 + \sum_{2..n} v_i) & \cdot\, \ell \overset{\texttt{total}}{\rightarrow} (v_1' + \sum_{2..n} v_i)
\end{array}
$$

Since the traces differ in the second action ($\ell \overset{\texttt{add}}{\leftarrow} v_1$ and $\ell \overset{\texttt{add}}{\leftarrow} v_1'$), change-propagation has to adjust the subsequent actions. Fortunately, the monolithic addition operation isn't sensitive to the current value, which eliminates the spurious dependence on the intermediate values of the accumulator. Only the last action is affected: the new total can be obtained by subtracting the old value $v_1$ and adding the new value $v_1'$, thus giving the correct new total $(v_1 + \sum_{2..n} v_i) - v_1 + v_1' = v_1' + \sum_{2..n} v_i$. Therefore change-propagation can update the total in constant time by replacing the last action $\ell \overset{\texttt{total}}{\rightarrow} v_1 + \sum_{2..n} v_i$ with $\ell \overset{\texttt{total}}{\rightarrow} v_1' + \sum_{2..n} v_i$ by using the group's inverse operation, without affecting the intervening additions $\ell \overset{\texttt{add}}{\leftarrow} v_i$ ($i \neq 1$) in the trace.

**Example 10 (Kinetic Motion Simulation)**
A kinetic motion simulator models a set of moving points and can compute some property (e.g., the convex hull) of the system. An existing implementation of a kinetic motion simulator in self-adjusting computation [Acar et al., 2008b] takes the points as inputs and dynamically compares points to compute the convex hull. The comparisons are intermediate data of the program, but achieving efficient change-propagation requires allowing the mutator to modify the results of comparisons, which violates the proper-usage guidelines for the mutator. Moreover, even though the moving points are functions of time, time itself isn't an adaptively-tracked input. The design choices of the existing motion simulator fail to make a clear distinction between the core- and meta-level computations, so the program is intrinsically tied to the implementation of the change-propagation mechanism.

Self-adjusting computation can be extended with a *modular* modifiable reference that discretizes a continuous value according to some partition, thus limiting recomputation to when the value crosses a partition boundary. By making time an explicit input and computing the comparisons with modular modrefs, we can recover the efficiency of the above motion simulator without sacrificing the safety features of the language. In particular, the mutator wouldn't have to explicitly manipulate the self-adjusting program's intermediate data, thus preserving the core-/meta-level distinction.

### 2.3.2 Foreground

We raise the level of abstraction of dependence-tracking by generalizing from individual memory operations to work at the granularity of the query and update operations of arbitrary abstract data types. Coarser-grained dependence-tracking reduces the number of operations and dependencies tracked by self-adjusting computation, which yields the following benefits:

- it can asymptotically reduce the time and space overhead for from-scratch runs,
- it can exploit the problem-specific structure of many problems and make them more stable, yielding asymptotically faster change-propagation relative to self-adjusting computation with modifiable references alone,
- it can greatly simplify the analysis of stability since the user only needs to consider the operations on the data type instead of all the memory accesses inside of it.

We make the Src and Tgt languages open-ended to extension by any data type that conforms to a *traceable data type* (TDT) interface. A traceable version of a data type [Demaine et al., 2004, Acar et al., 2007a] provides the same extensional functionality of an ordinary data type (i.e., update and query operations); additionally, it maintains an intensional trace history of operations for each instance and allows retroactively revising the operations of the trace. A TDT allows invoking or revoking an operation at any point in the trace history, which in turn reports the earliest subsequent query made inconsistent (i.e., that depends on and whose return value is affected) by the revision. By tracking operations at the desired level of granularity, change-propagation can update a computation more efficiently than an extensionally-equivalent modref-based implementation.

The uniform interface for TDTs enables the change-propagation mechanism to remain agnostic about the implementation details of each data type. Since each TDT provides a self-contained mechanism for trace revisions, any number of TDTs can be integrated orthogonally into the self-adjusting language and implementations of TDTs do not need to be aware of the change-propagation algorithm beyond the interface.

#### Abstract Data Types

We define an (conventional) abstract data type $D$ as a quadruple $(\tau\ \mathbf{tdt}, \mathcal{S}, \mathbf{mk}, \{\mathbf{op}\_i\})$ consisting of:

- a type constructor $\tau\ \mathbf{tdt}$,

- a state constructor $\mathcal{S}$,
- a creation operation $\mathbf{mk} : \tau_{mk} \rightarrow \tau\ \mathbf{tdt}$ to initialize an instance, and
- a set of query and update operations $\mathbf{op} : \tau_{arg} \rightarrow \tau_{res}$ to manipulate the state and compute some result.

The extensional behavior of a data type is specified by a state-transformation function for each operation. The state-transformation $v \overset{\mathbf{mk}}{\mapsto} \mathcal{S}_0$ for creating a TDT maps a value $v$ to an initial state $\mathcal{S}_0$. The state-transformation $\mathcal{S}; v_{arg} \overset{\mathbf{op}}{\mapsto} \mathcal{S}'; v_{res}$ for an operation $\mathbf{op}$ maps a state $\mathcal{S}$ and argument value $v_{arg}$ to another state $\mathcal{S}'$ and result value $v_{res}$. Any data type, however complex, can be specified in this way by coming up with an appropriate representation for the state and by specifying the state-transformation function.

### Traceable Data Types

A *traceable data type* (TDT) provides the extensional behavior of an abstract data type for from-scratch runs; additionally, it allows retroactively changing its history of operations, which enables change-propagation to update a run according to input changes. To talk about a sequence of operations, we let Time denote a totally-ordered set of time stamps. We define a *operation trace* $H$ for a data type as an initial state $\mathcal{S}_0$ and a sequence $\langle (t_i, o_i, v_i) \rangle_{i \in 1..n}$, where the $t_i \in \mathsf{Time}$, $t_i < t_{i+1}$, and each $o_i$ is an operation of the form $\mathbf{op}\_k\ v_i'$ that takes some $v_i'$ as an argument to return $v_i$.

Let $\mathsf{eval}(H, t)$ be the value returned by performing the sequence of operations $(o_1, o_2, \ldots)$ in $H$ up to time $t$, inclusive. We say that an element $(t_i, o_i, v_i) \in H$ of the operation trace is *inconsistent* if $\mathsf{eval}(H, t_i) \neq v_i$. We say that an operation trace is inconsistent if any element is inconsistent, and consistent otherwise.

For an abstract data type $D$, the *traceable version* $\mathsf{Tr}(D)$ abstractly maintains an operation trace $H$ for each instance and allows revisions to update the sequence of operations. The abstract data type $D$'s creation operation $\mathbf{mk}$ of type $\tau_{mk} \rightarrow \tau\ \mathbf{tdt}$ induces an analogous version `mk` for the traceable version that returns a new operation trace $H$ with initial state $\mathcal{S}_0$ (where $v \overset{\mathbf{mk}}{\mapsto} \mathcal{S}_0$) and empty operation sequence.

For each manipulation operation $\mathbf{op}$ of type $\tau_{arg} \rightarrow \tau_{res}$, the traceable version provides *invoke* and *revoke* analogues associated with each instance $H$. The invoke analogue `invoke_op : ts * `$\tau_{arg}$` -> ts option * `$\tau_{res}$ takes an argument value $v_{arg}$ and time stamp $t$, updates the operation trace $H$ by inserting $(t, \mathbf{op}\, v_{arg})$, and returns $v_{res} = \mathsf{eval}(H, t)$. The revoke analogue `revoke_op : ts -> ts option`

removes the operation with time $t$ from $H$ (if any). Both the invoke and revoke operations return an optional time stamp corresponding to the next operation, if any, that has been made inconsistent (e.g., its return value changes) by the revision. Note that type `ts` classifies time stamps Time.

We refer to `invoke` and `revoke` (meta-)operations as *revisions* and require them to be applied as part of a *revision sequence*—a sequence of revisions on an initially consistent operation trace such that (1) the times of the revisions are increasing, and (2) for each revision at time $t$, all operation at times before $t$ are consistent. Multiple revision sequences can be applied to an operation trace sequentially, each returning the operation trace to a consistent state before the next starts.

In previous semantics for self-adjusting computation with modrefs [Ley-Wild et al., 2008a, 2009], change-propagation would replay an action when possible or fall back to evaluation either nondeterministically or because the action couldn't be replayed. The novelty of TDTs lies in that the *consistency* of each TDT action in a trace is explicitly identified—i.e., whether the action can be replayed or not. Therefore change-propagation necessarily replays a TDT action iff it is consistent; the nondeterminism of memoization, however, is not eliminated. The replayability of a TDT action is determined by the state transformation semantics of the TDT operations. When a TDT operation is *invoked* by evaluation, in addition to performing the state transformation, the reuse trace is scanned to identify the next inconsistent action—i.e., the next action whose result differs because there is a new intervening TDT operation. Dually, when memoization matches the tail of a reuse trace, the prefix of the trace is discarded and those actions are explicitly *revoked* by scanning the rest of the trace to update the consistency of the remaining actions.

**Language Integration.** The proposed interface with invoke and revoke operations is significantly more cumbersome to use than the standard operations. Fortunately, the standard interface can be presented in the Src language, while the traceable version is integrated into the Tgt language. A TDT is integrated into each language by adding a type constructor for the data structure together with a primitive for each operation.

A self-adjusting Src program uses the same direct style update and query operations of the abstract data type. The only necessary change to a Src program is to replace the modref-based implementation by a traceable version with the same interface.

The continuation-based Tgt language generalizes traces to record each TDT operation (instead of only modref operations) and its associated continuation, which needs to be rerun if the operation becomes inconsistent. Creating a TDT instance extends the store with a location bound to a freshly initialized TDT state. The from-scratch semantics

of a TDT manipulation invokes the operation, which performs the corresponding state transformation and records a trace action indicating the operation with its argument and result values. The change-propagation semantics for a TDT action is to replay it when possible and falls back to evaluation if it cannot be replayed. The change-propagation implementation uses a time-ordered priority queue of inconsistent queries, but instead of tracking all inconsistencies for all TDT instances, it only keeps the earliest inconsistency for each instance, which is critical for efficiently handling certain usage patterns. When a segment of the trace is discarded, change-propagation uses the revoke operation of each action to notify its operation trace and update the priority queue of inconsistent queries.

**Remark.** It may seem odd that revisions only return the earliest inconsistent operation as opposed to all of them. In fact, this suffices because revision sequences require that the earliest inconsistency be fixed (revoked and possibly reinvoked) before proceeding to the next one. Fixing the first inconsistency will then return the next inconsistent operation, if any. This ability to return inconsistent operations lazily is critical for efficiency because otherwise we would have to maintain a potentially large sequence of inconsistent operations as some become consistent or others become inconsistent, and we would not be able to take advantage of subsequent revisions fixing inconsistencies.

For example imagine invoking an additional `insert` operation on a priority queue inserting an element with higher priority than all the others. This will cause all the rest of operations to become inconsistent. Invoking another `deleteMin` operation subsequently, however, would make all operations consistent by removing the newly inserted element.

### 2.3.3 Concrete TDTs

We present the formal abstract data type specifications of the following TDTs:

| TDT | Description |
|---|---|
| Pure | immutable (single-write) modifiable reference |
| Imp | mutable (multi-write) modifiable reference |
| Acc | accumulator modref for a commutative group |
| Mod | modular modref for discretizing a total order |
| Queue | first-in-first-out queue |
| PQueue | priority queue |
| Dict | dictionary |

For the presentation of these TDTs, we assume some standard types such as products, sums, and recursive types.

**Immutable and Mutable Modifiable References.** A modifiable reference (abbreviated *modref*) provides the functionality an ML-style reference with type constructor $\tau$ **modref** and state constructor `modref` $v$ where $v$ is a value of type $\tau$. The signature types and state-transformations are:

| operation : type | state-transformation |
|---|---|
| $\mathbf{put} : \tau \to \tau\ \mathbf{modref}$ | $v \overset{\mathbf{put}}{\mapsto} \mathsf{modref}\ v$ |
| $\mathbf{get} : \mathbf{unit} \to \tau$ | $\mathsf{modref}\ v; () \overset{\mathbf{get}}{\mapsto} \mathsf{modref}\ v; v$ |
| $\mathbf{set} : \tau \to \mathbf{unit}$ | $\mathsf{modref}\ v; v' \overset{\mathbf{set}}{\mapsto} \mathsf{modref}\ v'; ()$ |

The **put** operation creates a fresh modref and **get** dereferences its contents. Mutable modrefs additionally provide the **set** operation to update the contents of a modref. Intuitively, creating a modifiable with contents $v$ and then dereferencing the modifiable multiple times yields an operation trace with initial state `modref` $v$ and operation sequence $[(t_1, \mathbf{get}\ (), v), \ldots, (t_n, \mathbf{get}\ (), v)]$. If we change the initial value to $v'$ then the initial state becomes `modref` $v'$ and the time stamp $t_1$ identifies the earliest inconsistent operation. Change-propagation can successively reinvoke each revision to obtain the consistent sequence $[(t_1, \mathbf{get}\ (), v'), \ldots, (t_n, \mathbf{get}\ (), v')]$.

**Modular Modifiables.** In some applications, the domain of data may be continuous even when the computation produces a discrete result—e.g., a program computing the convex hull of a set of moving points represented combinatorially. In such a case, using modifiables makes change-propagation sensitive to any change, thus forcing recomputation even if the result is the same. In many of these cases, we partition the continuous domain into some discrete number of sets and consider values equal if they fall into the same set. For example, we may care only about the sign of a real number. A motion simulation would use modular modifiables to store the time variable.

A *modular modifiable* allows discretizing a totally-ordered (possibly continuous) set to avoid recomputation when modifications don't affect the discrete outcome. The type of modular modifiables is $\tau$ **mod** and the state constructor is `acc` $(v_c, v)$, where $v_c$ is a comparison function of type $\tau$ **cmp** $(= \tau \times \tau \to \mathbf{order})$ where **order** is the SML order datatype) and $v$ is the value of the modifiable. The signature types and state-transformations are:

| operation : type | state-transformation |
|---|---|
| $\mathbf{mod} : \tau\,\mathbf{cmp} \times \tau$ | $(v_\mathrm{c}, v) \overset{\mathbf{mod}}{\mapsto} \mathsf{mod}\,(v_\mathrm{c}, v)$ |
| $\mathbf{mget} : (\tau, \tau')\,\mathbf{dis} \to \tau'$ | $\mathsf{mod}\,(v_\mathrm{c}, v); v_\mathrm{b} \overset{\mathbf{mget}}{\mapsto} \mathsf{mod}\,(v_\mathrm{c}, v); v'$ |

Modular modifiables are created by the $\mathbf{mod}$ operation and manipulated by the modular dereferencing operation $\mathbf{mget}$. A modular dereference takes a *discretization* argument $v_d$ of type $(\tau, \tau')\,\mathbf{dis}$ which is a (finite) partition of the (continuous) type $\tau$ together with an assignment of values from the (discrete) type $\tau'$ to each equivalence class. Formally, the discretization is represented by a list $[c_1, \ldots, c_n]$ that partitions $\tau$ into intervals and the assignment is a list $[d_0, \ldots, d_n]$ of $\tau'$ elements. The result of such a dereference is $v' = d_i$ where the current value of the modular modifiable is $c_i \leq v < c_{i+1}$. Due to the structure of the partition, the outcome of a modular dereference only changes when the value of the modular modifiable changes equivalence classes; this enables operating efficiently on continuously-varying values such as time.

**Accumulator Modifiables.** An *accumulator modifiable* exploits the structure of a commutative group to provide the high-level operations of adding to and querying the total value of an accumulator. An accumulator modifiable has type constructor $\tau\ \mathbf{acc}$ and state constructor $\mathsf{acc}\,(v, v_{add}, v_{sub})$ where $v$ is a value of type $\tau$ and $v_{add}, v_{sub}$ are mutually-inverse binary operators of type $\tau\ \mathbf{bop}(= \tau \times \tau \to \tau)$. The signature types and state-transformation semantics are:

| operation : type | state-transformation |
|---|---|
| $\mathbf{acc} : \tau \times \tau\,\mathbf{bop} \times \tau\,\mathbf{bop} \to \tau\,\mathbf{acc}$ | $(v_\mathrm{u}, v_\mathrm{a}, v_\mathrm{s}) \overset{\mathbf{acc}}{\mapsto} \mathsf{acc}\,(v_\mathrm{u}, v_\mathrm{a}, v_\mathrm{s})$ |
| $\mathbf{add} : \tau \to \mathbf{unit}$ | $\mathsf{acc}\,(v, v_\mathrm{a}, v_\mathrm{s}); v_\mathrm{x} \overset{\mathbf{add}}{\mapsto} \mathsf{acc}\,(v', v_\mathrm{a}, v_\mathrm{s}); ()$ |
| $\mathbf{total} : \mathbf{unit} \to \tau$ | $\mathsf{acc}\,(v, v_\mathrm{a}, v_\mathrm{s}); () \overset{\mathbf{total}}{\mapsto} \mathsf{acc}\,(v, v_\mathrm{a}, v_\mathrm{s}); v$ |

where $v_u$ is the initial value of the accumulator (for the $\mathbf{acc}$ operation) and where $v' = v_a(v, v_x)$ (for $\mathbf{add}$). The creation operation $\mathbf{acc}$ initializes an accumulator with the group operations, the $\mathbf{add}$ operation updates the accumulator, $\mathbf{total}$ queries the current total.

**Queues.** A *queue* has type $\tau\ \mathbf{q}$ and state constructor $\mathsf{q}\,\mathcal{Q}$ where $\mathcal{Q}$ is a sequence of values $\langle v_i \rangle$ that represent the contents of the queue. Queue commands include the creation operation $\mathbf{q}$ and manipulation operations $\mathbf{push}$ for pushing onto the queue and $\mathbf{pop}$ for popping from the queue:

| operation : type | state-transformation |
|---|---|
| $\mathbf{q} : \mathbf{unit}$ | $() \overset{\mathbf{q}}{\mapsto} \mathsf{q}\,\langle\rangle$ |
| $\mathbf{push} : \tau \to \mathbf{unit}$ | $\mathsf{q}\,\mathcal{Q}; v \overset{\mathbf{push}}{\mapsto} \mathsf{q}\,(\mathcal{Q} + v); ()$ |
| $\mathbf{pop} : \mathbf{unit} \to \tau$ | $\mathsf{q}\,\mathcal{Q}; v \overset{\mathbf{pop}}{\mapsto} \mathsf{q}\,(\mathcal{Q} - v_1); v_1$ |

where $\mathcal{Q} + v$ adds $v$ to the end of the queue, and $\mathcal{Q} - v_1$ removes the first element $v_1$ from the queue.

**Priority Queues.** A *priority queue* with $\tau_k$ priorities and $\tau_v$ values has type $(\tau_k, \tau_v)$ $\mathbf{pq}$. The state constructor is $\mathsf{pq}\,\mathcal{PQ}$ where $\mathcal{PQ}$ is a sequence of pairs $\langle (v_{ki}, v_{vi}) \rangle$ where entry $v_{vi}$ has priority $v_{ki}$. Priority queue commands include the creation operation $\mathbf{pq}$ and manipulation operation $\mathbf{inspri}$ for inserting an element $v_v$ with priority $v_k$ and $\mathbf{delmin}$ for deleting the element with lowest priority:

| operation : type | state-transformation |
|---|---|
| $\mathbf{pq} : \mathbf{unit}$ | $() \overset{\mathbf{pq}}{\mapsto} \mathsf{pq}\,\langle\rangle$ |
| $\mathbf{inspri} : \tau_\mathrm{k} \times \tau_\mathrm{v} \to \mathbf{unit}$ | $\mathsf{pq}\,\mathcal{PQ}; (v_\mathrm{k}, v_\mathrm{v}) \overset{\mathbf{inspri}}{\mapsto} \mathsf{pq}\,(\mathcal{PQ} + (v_\mathrm{k}, v_\mathrm{v})); ()$ |
| $\mathbf{delmin} : \mathbf{unit} \to \tau_\mathrm{k} \times \tau_\mathrm{v}$ | $\mathsf{pq}\,\mathcal{PQ}; () \overset{\mathbf{delmin}}{\mapsto} \mathsf{pq}\,(\mathcal{PQ} - (v_\mathrm{k}, v_\mathrm{v})); (v_\mathrm{k}, v_\mathrm{v})$ |

where $\mathcal{PQ} + (v_k, v_v)$ adds the element $v_v$ with priority $v_k$, and $\mathcal{PQ} - (v_k, v_v)$ removes the element $v_v$ with highest priority $v_k$.

**Dictionaries.** A *dictionary* $(\tau_k, \tau_v)$ $\mathbf{dict}$ maps $\tau_k$ keys to $\tau_v$ values. The state constructor is $\mathtt{dict}\ \mathcal{M}$ where $\mathcal{M}$ is a mapping $\{v_{ki} \mapsto v_{vi}\}$ that binds each key $v_{ki}$ to value $v_{vi}$. Priority queue commands include the creation operation $\mathbf{dict}$ and manipulation operation $\mathbf{inspri}$ for inserting an element $v_v$ with priority $v_k$ and $\mathbf{delmin}$ for deleting the element with lowest priority:

| operation : type | state-transformation |
|---|---|
| $\mathbf{dict} : \mathbf{unit}$ | $() \overset{\mathbf{dict}}{\mapsto} \mathsf{dict}\,\{\}$ |
| $\mathbf{ins} : \tau_\mathrm{k} \times \tau_\mathrm{v} \to \mathbf{unit}$ | $\mathsf{dict}\,\mathcal{M}; (v_\mathrm{k}, v_\mathrm{v}) \overset{\mathbf{ins}}{\mapsto} \mathsf{dict}\,(\mathcal{M} + v_\mathrm{k} \mapsto v_\mathrm{v}); ()$ |
| $\mathbf{rem} : \tau_\mathrm{k} \to \tau_\mathrm{v}$ | $\mathsf{dict}\,\mathcal{M}; v_\mathrm{k} \overset{\mathbf{rem}}{\mapsto} \mathsf{dict}\,(\mathcal{M} - v_\mathrm{k} \mapsto v_\mathrm{v}); v_\mathrm{v}$ |
| $\mathbf{lookup} : \tau_\mathrm{k} \to \tau_\mathrm{v}$ | $\mathsf{dict}\,\mathcal{M}; v_\mathrm{ki} \overset{\mathbf{lookup}}{\mapsto} \mathsf{dict}\,\mathcal{M}; v_\mathrm{vi}$ |

where $\mathcal{M} + v_k \mapsto v_v$ represents the map $\mathcal{M}$ extended with key $v_k$ bound to value $v_v$, and $\mathcal{M} - v_k \mapsto v_v$ represents the map $\mathcal{M}$ without the binding for key $v_k$.

**Example 11**
The traceable version of the priority queue signature `PRIORITY_QUEUE` is:

```
signature PRIORITY_QUEUE_TRACEABLE = sig
  type ('k,'v) t
  val new:  ('k * 'k -> order) -> ('k,'v) t
  val invoke_insert:  ts * (('k,'v) t * 'k * 'v) -> ts option * unit
  val revoke_insert:  ts -> ts option
  val invoke_delMin:  ts * ('k,'v) t -> ts option * ('k * 'v) option
  val revoke_delMin:  ts -> ts option
end
```

A traceable priority queue implementation `PQTraceable` that matches this signature can be integrated into the ΔML compiler to produce a user-level structure `PQ_Traced` with the same signature `PRIORITY_QUEUE` as the modifiable-based `PQ_Modifiable`.

Suppose we insert the numbers $3, 2, 1$ and then remove three elements from a priority queue $Q$:

$$[3 \to Q]\ [2 \to Q]\ [2 \to Q]\ [Q \to 1]\ [Q \to 2]\ [Q \to 3]$$

The `delMin` operations return the values $1, 2, 3$ in sorted order. If we now `revoke_insert (pq,1)`, then all `delMin` operations are inconsistent and the revision will return then the first `delMin` operation because it is the earliest affected operation. Change-propagation will rerun the continuation of the first inconsistent operation until it encounters a memoization point, and thereafter rerun any subsequent priority queue operations that remain inconsistent.

```
structure PQ : PRIORITY_QUEUE = PQTraced
afun heapsort (compare, l) =
let
  val heap = PQ.new $ compare
  afun insert x = PQ.insert $ (heap, x, ())
  mfun loop m =
    case m of
      NONE => put $ nil
    | SOME (k, ()) =>
        let val t = loop $ (PQ.deleteMin $ heap)
        in put $ (k::t) end
in
  (List.app insert $ l;
   loop $ (PQ.deleteMin $ heap))
end
```

Figure 2.6: Code for heapsort in ΔML.

**Example 12 (Heapsort)**
Consider a ΔML implementation of heapsort that uses a traceable priority queue, as shown in Figure 2.6. The algorithm first allocates an empty priority queue and inserts all the keys in its input to the priority queue (with unit payload). It then constructs the sorted output list by repeatedly removing the minimum element until the queue is empty and accumulating it in a list. The self-adjusting version uses a traceable priority queue `PQTraced`, linked lists, and adaptive functions; `loop` is memoizing because it performs non-constant work.

This algorithm has has an optimal $O(n \log n)$ from-scratch running time. Moreover, it is highly stable under small modifications to its input when operations are tracked at the granularity of priority queue operations. Thus, with a traceable priority queue, we can obtain an efficient self-adjusting sorter.

```
structure Dict :  DICTIONARY = struct ...  end
structure PQ : PRIORITY_QUEUE = PQTraced
structure List :  LIST = struct ...  end
type node = ...
type dist = ...
type graph = (node, (node * dist) List.t) Dict.t
afun dijkstra (root:  node, graph:  graph) =
let
  val dict_sp:  (node, dist) Dict.t = Dict.new $ ()
  val pq_v:  (dist * node) PQ.t = PQ.new $ ()
  afun visit (u, d:dist) =
  let afun ins (v,w) = PQ.insert $ (pq_v, (d + w, v))
  in case Dict.lookup $ (graph, u) of
       NONE => ()
     | SOME ns => List.app ins $ ns
  end

  mfun loop (u:node, d:dist) =
  (if (Dict.lookup $ (dict_sp, u)) = NONE then
    (Dict.insert $ (dict_sp, u, d); visit (u,d))
   else ();
   case PQ.deleteMin $ pq_v of
     NONE => dict_sp
   | SOME (d, v) => loop $ (v,d))
in loop $ (root, 0) end
```

Figure 2.7: Code for Dijkstra's algorithm in ΔML.

**Example 13 (Dijkstra's algorithm)**
Consider Dijkstra's algorithm for computing single-source shortest-paths, whose ΔML code is shown in Figure 2.7. We omit some details to focus attention on the aspects relevant to our interest here. Dijkstra's algorithm takes a graph and a root node and finds the shortest-path distance from the root to every node in the graph. We represent the input graph as a dictionary of nodes (`t_graph`), mapping each node to the list of its neighbors along with the edge weights. Similarly, we represent the output as a dictionary of nodes (`dict_sp`), mapping each node to its distance to the root. The key idea in the algorithm is to maintain a set of explored vertices and their distances to the root and expand this set iteratively. For this purpose, we maintain a priority queue (`pq_v`) of visited vertices and their current distances.

The algorithm inserts the root into the priority queue with distance 0, and then `loop` iteratively visits the vertices in the order of their current distance. Given a vertex $u$ and its current distance `d`, the function `loop` checks if $u$ is already visited. If so, then it continues by removing the next vertex from the priority queue. Otherwise, the exact distance for $u$ is found and inserted in the output (`dict_sp`), and the node is visited. Visiting vertex $u$

traverses each outgoing edge $(u, v)$ and inserts the neighbor $v$ into the priority queue with an updated distance.

#### Implementation

To assess the effectiveness of the approach, we extend the ΔML implementation to support TDTs and implement traceable versions of several data types. We note that we do not expect that new data types would be implemented very often; various traceable data types were designed by Acar et al. [2007a]. The ΔML implementation provides a Src-level interface to TDTs query and update operations, which are compiled to use the corresponding invoke and revoke operations and integrated with change-propagation. The compiler generates the necessary code for tracking the invoke and revoke operations and for finding and re-executing them when necessary during change-propagation.

Furthermore, we perform an experimental evaluation between TDT and modifiable-based implementations with benchmarks drawn from many application areas. The benchmarks include heapsort, Dijkstra's shortest path algorithm, breadth-first search on graphs, Huffman coding, and interval stabbing (Section 7.3). The experiments show space and time improvements in the asymptotic performance and implementation overhead, sometimes by as much as two orders of magnitude. Even on moderate input sizes, the improvements range between a factor of $3$ and $20$ reduction in from-scratch running time, between a factor of $4$ and $50$ reduction in space, and between a factor of $4$ and $5,000$ reduction in update time compared to the version using only modifiable references.

### 2.3.4 Presentation

In Chapter 3, we present the Src-level TDT interface with update and query operations. In Chapter 4, we present the Tgt-level TDT usage in terms of invoke and revoke operations. In Subsections 6.5.3 and 6.5.4, we present how to implement efficient TDTs and integrate them into the ΔML library. We present an experimental evaluation in Section 7.3.

## 2.4 Extensible Computation Memoization: In-Order and Out-of-Order

### 2.4.1 Background

Given an input change, change-propagation traverses the trace *in execution order* with an edit cursor to update the computation. Adaptivity reruns the inconsistent subcomputations affected by the input change, insert the corresponding new subtraces, delete subtraces that are made obsolete by the change. Computation memoization reuses portions of the trace from the previous run if they are unaffected by the changes. In previous versions of self-adjusting computation [Acar et al., 2006b], computation memoization implicitly reuses work by moving the cursor forward past unaffected computation. Since computation is reused in execution order, it is referred to as *in-order* computation memoization.

Updating the trace with insertions and deletions means that, intuitively, change-propagation takes time proportional to the edit distance between the two traces. For the *monotone* class of computations, in which changes don't affect the order of computation, change-propagation takes time roughly proportional to the minimum edit distance between traces. Existing implementations employ a *greedy* reuse heuristic that eagerly reuses work from the previous run by deleting all intervening work between the cursor and the matching computation. Greedy in-order reuse is optimal for monotone computations Acar [2005] because it achieves minimum edit distance. Experience has shown that some programs can be made monotone by subtle code annotations that prevent certain subcomputations from matching and enabling efficient change-propagation with in-order reuse, albeit at the expense of a more complicated program.

In-order reuse, however, is grossly inefficient for changes that directly reorder the input or indirectly reorder the computation. For example, change-propagating from an initial run $f(x); g(y)$ to updated run $g(y); f(x)$ would greedily discard the original $f$ call so that it can reuse the $g$ call, and thus have to rerun the $f$ call afresh. The inefficiency is particularly problematic when $f$ and $g$ are both expensive because one of the two must be discarded; if chunks of computation could be reordered, change-propagation would only incur the cost of swapping them. More generally, certain classes of programs, such as lazy (call-by-need) computation with its unpredictable order of evaluation, are inherently incompatible with in-order reuse.

### 2.4.2 Foreground

We abstract computation memoization from change-propagation and consider an alternative form of *out-of-order* computation memoization, which allows for the efficient reordering of large chunks of the computation. Out-of-order reuse moves a matching computation backward in the trace past the edit cursor without deleting the intervening computation, effectively reordering the trace.

Out-of-order reuse broadens and simplifies the applicability of self-adjusting computation. Several classes of programs benefit asymptotically from out-of-order reuse because more computations are available for reuse during change-propagation: some pure programs no longer need to be restructured or annotated to be monotone, some imperative programs that operate on unstructured data (e.g., graphs) are no longer sensitive to certain input changes, lazy programs can reuse computation in spite of inherent reordering. Greedy out-of-order reuse is asymptotically optimal for computations with unique function calls, which is broader than the monotone class.

### 2.4.3 Presentation

In Chapter 3, we give an extension of the Src language with primitives for lazy computation which can be implemented with mutable modrefs. In Chapter 3 and Chapter 4, we give local and global notions of trace distance (see Section 2.2) to account for in-order and out-of-order reuse. Local trace distance quantifies the dissimilarity between two runs in execution order, while global trace distance quantifies the dissimilarity between two runs modulo reordering. In Chapter 6, we discuss the implementation overhead for supporting in-order and out-of-order memoization. In Chapter 7, we give examples where change-propagation requires out-of-order memoization to achieve an asymptotic speedup relative to from-scratch evaluation.

# Chapter 3

# The Src$*$ Languages

This chapter is based on work on a direct style language for self-adjusting computation with single-write modrefs [Ley-Wild et al., 2008b], a cost semantics and trace distance for a language with multi-write modrefs [Ley-Wild et al., 2009], and extensibility to traceable data types [Acar et al., 2010a].

## 3.1 Overview

The schematic Src($T$) language is a simply-typed, call-by-value $\lambda$-calculus that serves to write direct style programs. The language is parameterized by a traceable data type $T$ (possibly several) to represent changeable data. In addition to Src($-$) with TDTs, we present the SrcLazy language with suspensions for lazy (call-by-need) evaluation. We use Src$*$ to refer collectively to these languages, and Src to refer to a representative one.

Src has a standard static and dynamic semantics, as well as a cost semantics. The dynamic semantics determines the *extensional* meaning of a program as the result of evaluation while the cost semantics determines the *intensional* meaning as the number of evaluation steps. Although Src has no operational support for self-adjusting computation—i.e., a mechanism for updating a computation under input changes—, its dynamic and cost semantics produces an *execution trace*, which is an abstract representation of the evaluation derivation.

A theory of *trace distance* quantifies dissimilarities between runs: *global* trace distance permit computation reordering, which corresponds to out-of-order computation reuse, while *local* trace distance does not, which corresponds to in-order computation reuse.

Src programs can be compiled into equivalent self-adjusting Tgt programs (see Chapters 4 and 5), whose semantics include a *change-propagation* judgement that realizes updates and asymptotically matches Src distance.

## 3.2 Syntax

The syntax of Src($T$) for some is given by the following grammar, which defines types $\tau$, expressions $e$, and values $v$, using metavariables $f$ and $x$ for identifiers and $\ell$ for locations. We assume the schematic TDT $T$ has a type constructor $\mathbf{tdt}$, creation operation $\mathbf{mk}\ v_{mk}$, and manipulation operation $\mathbf{op}\ v_l\ v_{arg}$.

$$\begin{array}{rrcl} \text{types} & \tau & ::= & \mathbf{nat} \mid \tau_{\mathrm{x}} \to \tau \mid \tau_{\mathrm{x}} \xrightarrow{\$} \tau \mid \tau\ \mathbf{tdt} \\ \text{expressions} & e & ::= & v \mid \mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, x.e_{\mathrm{s}} \mid e_{\mathrm{f}}\, e_{\mathrm{x}} \mid e_{\mathrm{f}}\, \$\, e_{\mathrm{x}} \mid \mathbf{mk}\, v_{\mathrm{mk}} \mid \mathbf{op}\, v_{\mathrm{l}}\, v_{\mathrm{arg}} \\ \text{values} & v & ::= & x \mid \mathbf{zero} \mid \mathbf{succ}\, v \mid \mathbf{fun}\, f.x.e \mid \mathbf{afun}\, f.x.e \mid \mathbf{mfun}\, f.x.e \mid \ell \end{array}$$

In Section 3.6, we extend the syntax to support lazy computation.

Src($-$) (as well as the Tgt($-$) language of Chapter 4 and the translation from the former to the latter) includes natural numbers for didactic purposes and can easily be extended with products, sums, recursive types, etc.; we omit their formalization as they provide no additional insight, but assume them for the presentation of some TDTs. Natural numbers are introduced by the constructors $\mathbf{zero}$ and $\mathbf{succ}\, v$, while the $\mathbf{caseN}$ primitive case-analyzes a natural number $v_n$ and branches to $e_z$ or $e_s$ according to whether it is zero or a successor number. Note that the argument of $\mathbf{succ}$ and the scrutinee of $\mathbf{caseN}$ are restricted to value forms for technical simplicity.

The language includes normal (non-adaptive) functions for ordinary computation, and adaptive functions that demarcate self-adjusting computation subject to input changes. Normal (non-adaptive) functions are classified by the $\tau_x \to \tau$ type, introduced with the $\mathbf{fun}$ keyword, and eliminated by juxtaposition. Adaptive functions are classified by the $\tau_x \xrightarrow{\$} \tau$ type, introduced by the $\mathbf{afun}$ and $\mathbf{mfun}$ keywords, and eliminated by the infix $\$$ keyword. In addition to demarcating self-adjusting computation, the *memoizing* adaptive function $\mathbf{mfun}$ identifies opportunities for computation similarities across runs and indicates that the function should use memoization when compiled.

The Src$*$ language is open-ended to extension by any number of traceable data types. Here we describe TDTs schematically; concrete TDTs are presented in Section 2.3. A TDT *signature* $\Delta$ specifies a type constructor $\tau\ \mathbf{tdt}$ to classify instances of the data type, a state constructor $\mathcal{S}$ to represent the abstract state of the TDT, and (one or more) *creation* and *manipulation* operations.

A creation operation $\mathbf{mk}\ v_{mk}$ initializes a TDT instance with seed value $v_{mk}$. A manipulation (i.e., query or update) operation $\mathbf{op}\ v_l\ v_{arg}$ takes a reference $v_l$ to a TDT instance and an argument value $v_{arg}$. The static semantics of TDT operations is specified by type assignments $\mathbf{mk} : \tau_{mk} \to \tau\ \mathbf{tdt}$ and $\mathbf{op} : \tau_{arg} \to \tau_{res}$ in the TDT signature; we assume the types in the signature are also representable by types in $\mathsf{Src}*$ (and $\mathsf{Tgt}*$). The dynamic semantics of a TDT operations is specified by its *state-transformation* judgement: $v_{mk} \xrightarrow{\mathbf{mk}} \mathcal{S}'$ defines how a $\mathbf{mk}$ command initializes a state constructor $\mathcal{S}'$ with seed data $v_{mk}$, and $\mathcal{S}; v_{arg} \xrightarrow{\mathbf{op}} \mathcal{S}'; v_{res}$ defines how an $\mathbf{op}$ command with argument $v_{arg}$ transforms the state from $\mathcal{S}$ to $\mathcal{S}'$ and yields result $v_{res}$. As with naturals, the arguments to TDT commands are restricted to value forms for technical simplicity. This restriction can be avoided with syntactic sugar: for example, the unrestricted creation form $\mathbf{mk}\ e_{mk}$ can be defined as $(\mathbf{mfun}\ f.x.\mathbf{mk}\ x)\$e_{mk}$.

We take a $\mathsf{Src}(-)$ store $\sigma$ to be a finite map from locations $\ell$ to TDT state constructors $\mathcal{S}$; the notation $\sigma[\ell \mapsto \mathcal{S}]$ denotes the store $\sigma$ updated with $\ell$ mapped to $\mathcal{S}$. Contexts $\Gamma$ and $\Sigma$ are maps from variables and locations to types, respectively.

In the signature, the state-transformation semantics represents the *extensional* behavior of an operation on a data type. In $\mathsf{Src}(-)$, a TDT operation also provides a level of indirection through the store. Creating a TDT instance allocates a location in the store and any subsequent uses of the TDT must read from the store, making dependencies explicit for any changeable data. Therefore, 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.

## 3.3 Static Semantics

The typing judgement $\Sigma; \Gamma \vdash^{\delta} e : \tau$ ascribes the type $\tau$ to the expression $e$ at the mode $\delta$ (either normal $\flat$ or adaptive $\$$) in the store and variable typing contexts $\Sigma$ and $\Gamma$. The mode component of the typing judgement precludes adaptive applications and the TDT primitives from the body of normal functions, but allows normal applications, adaptive applications, and the TDT 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 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

$$\frac{x : \tau \in \Gamma}{\Sigma; \Gamma \vdash^{\delta} x : \tau}\ \textbf{var} \qquad \frac{}{\Sigma; \Gamma \vdash^{\delta} \mathbf{zero} : \mathbf{nat}}\ \textbf{zero} \qquad \frac{\Sigma; \Gamma \vdash^{\delta} v : \mathbf{nat}}{\Sigma; \Gamma \vdash^{\delta} \mathbf{succ}\, v : \mathbf{nat}}\ \textbf{succ}$$

$$\frac{\Sigma; \Gamma, f : \tau_{\mathrm{x}} \to \tau, x : \tau_{\mathrm{x}} \vdash^{\flat} e : \tau}{\Sigma; \Gamma \vdash^{\delta} \mathbf{fun}\, f.x.e : \tau_{\mathrm{x}} \to \tau}\ \textbf{fun} \qquad \frac{\Sigma; \Gamma, f : \tau_{\mathrm{x}} \xrightarrow{\$} \tau, x : \tau_{\mathrm{x}} \vdash^{\$} e : \tau}{\Sigma; \Gamma \vdash^{\delta} \mathbf{afun}\, f.x.e : \tau_{\mathrm{x}} \xrightarrow{\$} \tau}\ \textbf{afun}$$

$$\frac{\Sigma; \Gamma, f : \tau_{\mathrm{x}} \xrightarrow{\$} \tau, x : \tau_{\mathrm{x}} \vdash^{\$} e : \tau}{\Sigma; \Gamma \vdash^{\delta} \mathbf{mfun}\, f.x.e : \tau_{\mathrm{x}} \xrightarrow{\$} \tau}\ \textbf{mfun} \qquad \frac{\ell : \tau \in \Sigma}{\Sigma; \Gamma \vdash^{\delta} \ell : \tau}\ \textbf{loc}$$

$$\frac{\Sigma; \Gamma \vdash^{\delta} v_{\mathrm{n}} : \mathbf{nat} \qquad \Sigma; \Gamma \vdash^{\delta} e_{\mathrm{z}} : \tau \qquad \Sigma; \Gamma, x : \mathbf{nat} \vdash^{\delta} e_{\mathrm{s}} : \tau}{\Sigma; \Gamma \vdash^{\delta} \mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, x.e_{\mathrm{s}} : \tau}\ \textbf{caseN}$$

$$\frac{\Sigma; \Gamma \vdash^{\delta} e_{\mathrm{f}} : \tau_{\mathrm{x}} \to \tau \qquad \Sigma; \Gamma \vdash^{\delta} e_{\mathrm{x}} : \tau_{\mathrm{x}}}{\Sigma; \Gamma \vdash^{\delta} e_{\mathrm{f}}\, e_{\mathrm{x}} : \tau}\ \textbf{app} \qquad \frac{\Sigma; \Gamma \vdash^{\$} e_{\mathrm{f}} : \tau_{\mathrm{x}} \xrightarrow{\$} \tau \qquad \Sigma; \Gamma \vdash^{\$} e_{\mathrm{x}} : \tau_{\mathrm{x}}}{\Sigma; \Gamma \vdash^{\$} e_{\mathrm{f}}\, \$\, e_{\mathrm{x}} : \tau}\ \textbf{aapp}$$

$$\frac{\begin{array}{c}\mathbf{mk} : \tau_{\mathrm{mk}} \to \tau\, \mathbf{tdt} \in \Delta_{\mathrm{tdt}} \\ \Sigma; \Gamma \vdash^{\$} v_{\mathrm{mk}} : \tau_{\mathrm{mk}}\end{array}}{\Sigma; \Gamma \vdash^{\$} \mathbf{mk}\, v_{\mathrm{mk}} : \tau\ \mathbf{tdt}}\ \textbf{mk} \qquad \frac{\begin{array}{c}\mathbf{op} : \tau_{\mathrm{arg}} \to \tau_{\mathrm{res}} \in \Delta_{\mathrm{tdt}} \\ \Sigma; \Gamma \vdash^{\$} v_{\mathrm{l}} : \tau\ \mathbf{tdt} \\ \Sigma; \Gamma \vdash^{\$} v_{\mathrm{arg}} : \tau_{\mathrm{arg}}\end{array}}{\Sigma; \Gamma \vdash^{\$} \mathbf{op}\, v_{\mathrm{l}}\, v_{\mathrm{arg}} : \tau_{\mathrm{res}}}\ \textbf{op}$$

Figure 3.1: Src$*$ typing $\Sigma; \Gamma \vdash^{\delta} e : \tau$.

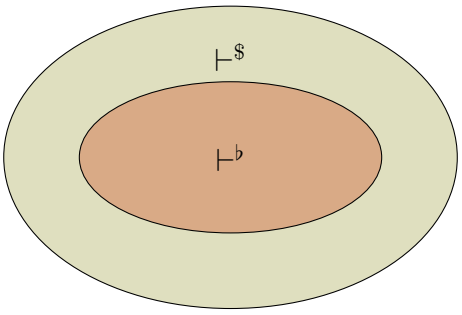



Figure 3.2: The relation between normal (pure) and self-adjusting programs.

distinguishes effectful and non-effectful functions and applications.

The type of a variable (resp. location) is determined by the variable (resp. store) typing context, independent of the mode.

The type of natural number values is determined structurally under either mode. A case-analysis must have a natural number scrutinee and both branches must agree on the type, independent of the mode.

A normal (resp. adaptive) function has a normal function type under any mode if the body of the function has the correct result type at the normal (resp. adaptive) mode by extending the context with the function and argument variables. A normal (resp. adaptive) application has the function expression's result type if the function expression has a normal (resp. adaptive) function type and the argument expression matches the argument type; the function and argument must be well-typed at the same mode as the application. A normal application may typecheck under either mode, but an adaptive application can only typecheck under the adaptive mode.

A TDT creation operation has the TDT type if the argument that matches the type prescribed by the TDT signature. A TDT manipulation operation has the result type prescribed by the TDT signature if the first argument is the location of a TDT and the second argument matches the TDT signature. Both TDT creation and manipulation operations, as well as their arguments, must typecheck under the adaptive mode.

## 3.4 Dynamic and Cost Semantics

The dynamic semantics of Src$*$ is defined by the large-step evaluation relation $\sigma; e \Downarrow \sigma'; v'$ to reduce expression $e$ in store $\sigma$ to value $v'$ in updated store $\sigma'$. The cost semantics is defined by the extended large-step evaluation relation $\mathcal{E};\ \sigma; e \Downarrow \sigma'; v'\ ; T'; c'$that additionally yields an execution *trace* $T'$ and a *cost* $c'$, and records the evaluation context $\mathcal{E}$. The dynamic semantics is defined for all Src$*$ programs, while the cost semantics is only defined for the sublanguage with memoizing recursive functions, but without normal or adaptive recursive functions. The dynamic semantics is given in Figure 3.3, the cost semantics is given simultaneously for the sublanguage on which it is defined.

The trace internalizes the *shape* of an evaluation derivation based on the use of memoizing functions and TDT operations, and serves to identify the dissimilarities between computations. The cost internalizes the *size* of a trace, and serves to relate the constant slowdown due to compiling Src programs to Tgt programs. The highlighted evaluation

$$\frac{}{\mathcal{E};\ \sigma; v\ \Downarrow\ \sigma; v\ ; \varepsilon; 0}\ \textbf{val}$$

$$\frac{\mathcal{E};\ \sigma; e_{\mathrm{z}}\ \Downarrow\ \sigma'; v'\ ; T'; c'}{\mathcal{E};\ \sigma; \mathbf{caseN}\ \mathbf{zero}\ e_{\mathrm{z}}\ x.e_{\mathrm{s}}\ \Downarrow\ \sigma'; v'\ ; T'; c'}\ \textbf{case/zero}$$

$$\frac{\mathcal{E};\ \sigma; [\, v_{\mathrm{n}} \,/\, x \,]\, e_{\mathrm{s}}\ \Downarrow\ \sigma'; v'\ ; T'; c'}{\mathcal{E};\ \sigma; \mathbf{caseN}\ (\mathbf{succ}\ v_{\mathrm{n}})\ e_{\mathrm{z}}\ x.e_{\mathrm{s}}\ \Downarrow\ \sigma'; v'\ ; T'; c'}\ \textbf{case/succ}$$

$$\frac{\begin{array}{c}\sigma; e_{\mathrm{f}}\ \Downarrow\ \sigma_{\mathrm{f}}; \mathbf{fun}\, f.x.e \\ \sigma_{\mathrm{f}}; e_{\mathrm{x}}\ \Downarrow\ \sigma_{\mathrm{x}}; v_{\mathrm{x}} \\ \sigma_{\mathrm{x}}; [\, \mathbf{fun}\, f.x.e \,/\, f \,]\, [\, v_{\mathrm{x}} \,/\, x \,]\, e\ \Downarrow\ \sigma'; v'\end{array}}{\sigma; e_{\mathrm{f}}\ e_{\mathrm{x}}\ \Downarrow\ \sigma'; v'}\ \textbf{fun} \qquad \frac{\begin{array}{c}\sigma; e_{\mathrm{f}}\ \Downarrow\ \sigma_{\mathrm{f}}; \mathbf{afun}\, f.x.e \\ \sigma_{\mathrm{f}}; e_{\mathrm{x}}\ \Downarrow\ \sigma_{\mathrm{x}}; v_{\mathrm{x}} \\ \sigma_{\mathrm{x}}; [\, \mathbf{afun}\, f.x.e \,/\, f \,]\, [\, v_{\mathrm{x}} \,/\, x \,]\, e\ \Downarrow\ \sigma'; v'\end{array}}{\sigma; e_{\mathrm{f}}\ \$\ e_{\mathrm{x}}\ \Downarrow\ \sigma'; v'}\ \textbf{afun}$$

$$\frac{\begin{array}{c}\mathcal{E}[\Box\ e_{\mathrm{x}}];\ \sigma; e_{\mathrm{f}}\ \Downarrow\ \sigma_{\mathrm{f}}; \mathbf{mfun}\, f.x.e\ ; T_{\mathrm{f}}; c_{\mathrm{f}} \\ \mathcal{E}[(\mathbf{mfun}\, f.x.e')\ \Box];\ \sigma_{\mathrm{f}}; e_{\mathrm{x}}\ \Downarrow\ \sigma_{\mathrm{x}}; v_{\mathrm{x}}\ ; T_{\mathrm{x}}; c_{\mathrm{x}} \\ \mathcal{E};\ \sigma_{\mathrm{x}}; [\, \mathbf{mfun}\, f.x.e \,/\, f \,]\, [\, v_{\mathrm{x}} \,/\, x \,]\, e\ \Downarrow\ \sigma'; v'\ ; T'; c'\end{array}}{\mathcal{E};\ \sigma; e_{\mathrm{f}}\ \$\ e_{\mathrm{x}}\ \Downarrow\ \sigma'; v'\ ; T_{\mathrm{f}} \cdot T_{\mathrm{x}} \cdot (\mathtt{app}_{\mathcal{E}}^{(\mathbf{mfun}\, f.x.e)\$v_{\mathrm{x}} \Downarrow v'}(T') \cdot \varepsilon); c_{\mathrm{f}} + c_{\mathrm{x}} + 1 + c'}\ \textbf{mfun}$$

$$\frac{\begin{array}{c}\ell \notin \mathrm{dom}\, \sigma \\ v_{\mathrm{arg}} \xrightarrow{\mathbf{mk}} \mathcal{S} \\ \sigma' = \sigma[\ell \mapsto \mathcal{S}]\end{array}}{\mathcal{E};\ \sigma; \mathbf{mk}\, v_{\mathrm{arg}}\ \Downarrow\ \sigma'; \ell\ ; \mathtt{mk}_{\mathcal{E}}^{v_{\mathrm{arg}} \uparrow \ell} \cdot \varepsilon; 1}\ \textbf{mk} \qquad \frac{\begin{array}{c}\ell \in \mathrm{dom}\, \sigma \\ \sigma(\ell) = \mathcal{S} \\ \mathcal{S}; v_{\mathrm{arg}} \xrightarrow{\mathbf{op}} \mathcal{S}'; v_{\mathrm{res}} \\ \sigma' = \sigma[\ell \mapsto \mathcal{S}']\end{array}}{\mathcal{E};\ \sigma; \mathbf{op}\, \ell\, v_{\mathrm{arg}}\ \Downarrow\ \sigma'; v_{\mathrm{res}}\ ; \mathtt{op}_{\mathcal{E}}^{\ell, v_{\mathrm{arg}} \downarrow v_{\mathrm{res}}} \cdot \varepsilon; 1}\ \textbf{op}$$

Figure 3.3: Src$*$ evaluation $\sigma; e\ \Downarrow\ \sigma'; v'$ (dynamic) and $\mathcal{E};\ \sigma; e\ \Downarrow\ \sigma'; v'\ ; T'; c'$(cost).

context $\mathcal{E}$ component is not necessary for reasoning about Src distance and can be ignored for the present time; it is revisited in Subsection 3.5.4 to give a more precise cost semantics and in Section 5.3 to relate Src and Tgt distances.

**Active and Passive Computation.** We distinguish *active* computation as work that may be used to identify similarities and differences in computation across runs. Evaluation of TDT operations and application of memoizing functions yield active computation. Case-analysis and (in the presence of products, sums, etc.) other forms of $\beta$-reduction are considered *passive* computation. Since normal and (non-memoizing) adaptive *recursive* functions can perform an arbitrary amount of computation between active computation, we omit them from the cost semantics; *non-recursive* normal and adaptive functions could be allowed because they perform a bounded amount of computation. This allows us to keep the cost semantics simple: we do not explicitly quantify passive work because it is always bounded by a constant multiple of active work. Intuitively, since a Src program can only perform a bounded amount of computation between function calls, memoizing functions suffice to account for all passive work. Including actions for passive work (e.g., case-analysis) would give a more accurate measure but isn't necessary for calculating asymptotic time complexity or distance. This property is formalized in Subsection 3.4.1.

**Traces.** A *trace* $T$ is an interleaving of *actions* that internalizes the *shape* of an evaluation derivation:

$$\begin{array}{rrcl} \text{traces} & T & ::= & \varepsilon \mid A{\cdot}T \\ \text{actions} & A & ::= & L \mid M(T) \\ \text{local actions} & L & ::= & \mathtt{mk}_{\mathcal{E}}^{v_{\mathrm{mk}}\uparrow\ell} \mid \mathtt{op}_{\mathcal{E}}^{\ell, v_{\mathrm{arg}}\downarrow v_{\mathrm{res}}} \\ \text{memoizing actions} & M & ::= & \mathtt{app}_{\mathcal{E}}^{v_{\mathrm{f}}\$v_{\mathrm{x}}\Downarrow v} \end{array}$$

*Actions* $A$ serve as markers for active work and consist of *local* and *memoizing* actions. *Local actions* $L$ include an action `mk` for each TDT creation command **mk** labeled with the seed data $v_{mk}$ and the location, and an action `op` for each TDT manipulation command **op** labeled with the location $\ell$, argument $v_{arg}$ and result $v_{res}$ values involved in each operation. *Memoization actions* $M$ include memoizing function application $\mathtt{app}^{v_f \$ v_x \Downarrow v}(T)$ labeled with a function $v_f$, argument $v_x$, and result $v$. A memo action identifies computation similarities by labeling the delimited trace $T$ with the computation it represents (e.g., the function application), and isolates the surrounding trace (e.g., the value returned to the caller) from any recomputation that may occur in the body. Actions are labeled with a highlighted evaluation context $\mathcal{E}$ that can be can be ignored for Src$*$-level reasoning; it is assigned by the cost semantics and is used by the translation to Tgt$*$ (Chapter 5). In Section 3.6, we extend local and memo actions to support lazy computation.

Traces facilitate identifying the similarities and differences between different runs of a program. More specifically, since store mutation is the only kind of observable side effect in Src, TDT operations uniquely determine the control flow of a closed program. Thus, by recording local actions in the trace, we can identify where program runs differ. Since memo actions identify explicitly similar computations (e.g., by matching arguments to and return values from function calls) and delimit the trace of the computation, they can be used to identify where program runs perform similar computations. Therefore traces are necessary and sufficient to isolate the similarities and differences between program runs, without having to capture pure computation (e.g., case-analysis) because it is determined by the rest of the trace.

**Dynamic and Cost Semantics.** Returning to the dynamic and cost semantics (Figure 3.3), evaluation extends the trace and increments the cost counter according to the kind of reduction. Cost grows in lock-step with the trace and could be defined as the *size* of the trace, but we keep it explicit to relate the intensional semantics of the Src and Tgt languages.

A value reduces to itself, produces an empty trace, and has no cost. A case-analysis reduces according to the branch prescribed by the scrutinee; the trace and cost are unchanged, because, as noted above, case-analysis only incurs passive work.

A function application—whether normal or adaptive—reduces the function $e_f$ and argument $e_x$ to values and then evaluates the redex. The cost semantics does not include normal or non-memoizing adaptive functions because they can generate an arbitrary amount of computation between active work. The cost semantics for a memoizing adaptive function application concatenates the function, argument, and redex traces to represent the sequencing of work; the redex trace is delimited by the memoizing function action to identify the scope of the function call; the cost of the traces are added and incremented by a unit of work for the $\beta$-reduction.

The Src semantics of TDT commands are induced by the TDT's standalone state-transformation semantics. The Src creation operation $\mathbf{mk}$ extends the store with a fresh location bound to the state specified by the operation's TDT initialization semantics, and returns the location. The Src manipulation operation $\mathbf{op}$ updates the location's state and returns the result value specified by the operation's TDT state-transformation semantics. In each case, the trace is the singleton action corresponding to the operation, and incurs a cost of $1$ for the current run.

Note that a creation command has the side-effect of extending the store, but a manipulation command only has a side-effect if the state-transformation actually mutates the state. Therefore, immutable (resp. mutable) modrefs contribute no computational power to

the language for from-scratch runs, but suffice to turn an ordinary pure (resp. imperative) program into a self-adjusting version.

### 3.4.1 Derivation Size and Cost

In this section, we show that the cost of an evaluation derivation, which quantifies active work, also bounds passive work. Formally, we show that cost bounds the size of a derivation, which includes both active and passive work, by a multiplicative factor that depends on the program and store.

We inductively define the *size* of a Src evaluation derivation $D$ with evaluation subderivations $D_1, \ldots, D_\mathrm{n}$ to be $|D| = 1 + \sum_{i \in 1..n} |D_\mathrm{i}|$. Furthermore, we define the *spread* of an expression to capture the amount of work done up to a function application. We inductively define the *local spread* $\langle e \rangle$ of a Src evaluation expression $e$ to be the longest path from the root of an expression to a leaf expression or function application.

$$\begin{aligned}
\langle v \rangle &= 1 \\
\langle \mathbf{caseN}\ v_\mathrm{n}\ e_\mathrm{z}\ x.e_\mathrm{s} \rangle &= 1 + \max\{\langle e_\mathrm{z} \rangle, \langle e_\mathrm{s} \rangle\} \\
\langle e_\mathrm{f}\ \$\ e_\mathrm{x} \rangle &= 1 \\
\langle \mathbf{mk}\ _ \rangle &= 1 \\
\langle \mathbf{op}\ _ \rangle &= 1
\end{aligned}$$

We define the *global spread* $\langle\langle e \rangle\rangle := \max_{e' \preceq e} \langle e \rangle$ of a Src evaluation expression $e$ to be the maximum local spread of the subexpressions $e'$ of $e$ ($e' \preceq e$). We extend the definition to a store and expression as $\langle\langle \sigma, e \rangle\rangle = \max_{e' \in \mathrm{rng}\,\sigma, e} \langle\langle e' \rangle\rangle$ and to an evaluation derivation as $\langle\langle \mathcal{E};\ \sigma; e\ \Downarrow\ \sigma'; v'\ ; T; c \rangle\rangle = \langle\langle \sigma, e \rangle\rangle$. Next, we establish several lemmas and show the size of a derivation is bounded by its cost times the global spread of a derivation.

**Lemma 14**
*For any* $e$, $\langle [\, v \,/\, x \,]\, e \rangle = \langle e \rangle$,

**Proof:** By straightforward induction on the expression $e$. ■

**Lemma 15**
*If* $D :: \mathcal{E};\ \sigma; e\ \Downarrow\ \sigma'; v'\ ; T; c$, *then* $\langle\langle \sigma', v' \rangle\rangle \leq \langle\langle \sigma, e \rangle\rangle$.

**Proof:** By straightforward induction on the derivation $D$. ■

**Lemma 16**
*If* $D :: \mathcal{E};\ \sigma; e \ \Downarrow\ \sigma'; v'\ ; T; c$ *with evaluation subderivations* $D_{\mathrm{i}}$ *(*$i \in 1..n$*), then* $\langle\!\langle D_{\mathrm{i}} \rangle\!\rangle \leq \langle\!\langle D \rangle\!\rangle$ *(*$i \in 1..n$*).*

**Proof:** By straightforward induction on the derivation $D$. ■

**Theorem 17**
*Fix* $D :: \mathcal{E};\ \sigma; e \ \Downarrow\ \sigma'; v'\ ; T; c$*, then* $|D| \leq \langle e \rangle + 3 \cdot c \langle\!\langle D \rangle\!\rangle$.

**Proof:** By induction on the derivation $D$.

Case **value**.

| | |
|---|---|
| $D :: ;\ \sigma; v \ \Downarrow\ \sigma; v\ ; \varepsilon; 0$ | derivation |
| $\lvert D \rvert = 1 \leq 1 = \langle v \rangle + 3 \cdot 0 \langle\!\langle D \rangle\!\rangle$ | arithmetic |

Case **caseZ** (**caseS** is analogous).

| | |
|---|---|
| $D :: \mathcal{E};\ \sigma; \mathbf{caseN}\ \mathbf{zero}\ e_{\mathrm{z}}\ x.e_{\mathrm{s}} \ \Downarrow\ \sigma'; v'\ ; T; c$ | derivation |
| $D' :: \mathcal{E};\ \sigma; e_{\mathrm{z}} \ \Downarrow\ \sigma'; v'\ ; T; c$ | subderivation |
| $\lvert D' \rvert \leq \langle e_{\mathrm{z}} \rangle + 3 \cdot c \langle\!\langle D' \rangle\!\rangle$ | i.h. |
| $\langle\!\langle D' \rangle\!\rangle \leq \langle\!\langle D \rangle\!\rangle$ | Lemma 16 |
| $\lvert D \rvert = 1 + \lvert D' \rvert \leq 1 + \langle e_{\mathrm{z}} \rangle + 3 \cdot c \langle\!\langle D' \rangle\!\rangle \leq \langle \mathbf{caseN}\ v_{\mathrm{n}}\ e_{\mathrm{z}}\ x.e_{\mathrm{s}} \rangle + 3 \cdot c \langle\!\langle D \rangle\!\rangle$ | arithmetic |

Case **app**.

| | |
|---|---|
| $D :: \mathcal{E};\ \sigma; e_{\mathrm{f}}\ \$\ e_{\mathrm{x}} \ \Downarrow\ \sigma'; v'\ ; T_{\mathrm{f}} \cdot T_{\mathrm{x}} \cdot (\mathsf{app}^{(\mathbf{mfun}\, f.x.e) \$ v_{\mathrm{x}} \Downarrow v'}_{\mathcal{E}(\ell)}(T) \cdot \varepsilon); c_{\mathrm{f}} + c_{\mathrm{x}} + 1 + c$ | derivation |
| $D_f :: \mathcal{E}[\Box\ e_{\mathrm{x}}];\ \sigma; e_{\mathrm{f}} \ \Downarrow\ \sigma_{\mathrm{f}}; \mathbf{mfun}\, f.x.e\ ; T_{\mathrm{f}}; c_{\mathrm{f}}$ | subderivation |
| $D_x :: \mathcal{E}[(\mathbf{mfun}\, f.x.e')\ \Box];\ \sigma_{\mathrm{f}}; e_{\mathrm{x}} \ \Downarrow\ \sigma_{\mathrm{x}}; v_{\mathrm{x}}\ ; T_{\mathrm{x}}; c_{\mathrm{x}}$ | subderivation |
| $D' :: \mathcal{E};\ \sigma_{\mathrm{x}}; [\, v_{\mathrm{x}} \,/\, x \,]\, [\, \mathbf{mfun}\, f.x.e \,/\, f \,]\, e \ \Downarrow\ \sigma'; v'\ ; T; c$ | subderivation |
| $\lvert D_{\mathrm{f}} \rvert \leq \langle e_{\mathrm{f}} \rangle + 3 \cdot c_f \langle\!\langle D_{\mathrm{f}} \rangle\!\rangle$ | i.h. |
| $\lvert D_{\mathrm{x}} \rvert \leq \langle e_{\mathrm{x}} \rangle + 3 \cdot c_x \langle\!\langle D_{\mathrm{x}} \rangle\!\rangle$ | i.h. |
| $\lvert D' \rvert \leq \langle [\, v_{\mathrm{x}} \,/\, x \,]\, [\, \mathbf{mfun}\, f.x.e \,/\, f \,]\, e \rangle + 3 \cdot c' \langle\!\langle D' \rangle\!\rangle$ | i.h. |
| $\langle\!\langle D_{\mathrm{f}} \rangle\!\rangle, \langle\!\langle D_{\mathrm{x}} \rangle\!\rangle, \langle\!\langle D' \rangle\!\rangle \leq \langle\!\langle D \rangle\!\rangle$ | Lemma 16 |
| $\langle e_{\mathrm{f}} \rangle, \langle e_{\mathrm{x}} \rangle, \langle [\, v_{\mathrm{x}} \,/\, x \,]\, [\, \mathbf{mfun}\, f.x.e \,/\, f \,]\, e \rangle \leq \langle\!\langle D \rangle\!\rangle$ | consequence |
| $\lvert D \rvert = 1 + \lvert D_f \rvert + \lvert D_x \rvert + \lvert D' \rvert$ $\leq 1 + (\langle e_{\mathrm{f}} \rangle + 3 \cdot c_f \langle\!\langle D_{\mathrm{f}} \rangle\!\rangle) + (\langle e_{\mathrm{x}} \rangle + 3 \cdot c_x \langle\!\langle D_{\mathrm{x}} \rangle\!\rangle)$ | |

$$
\begin{aligned}
&\quad +(\langle [\, v_{\mathrm{x}} \,/\, x \,]\, [\, \mathbf{mfun}\, f.x.e \,/\, f \,]\, e \rangle + 3 \cdot c' \langle\!\langle D' \rangle\!\rangle) \\
&\leq 1 + 3(c_f + c_x + 1 + c') \langle\!\langle D \rangle\!\rangle \\
&= \langle e_{\mathrm{f}} \,\$\, e_{\mathrm{x}} \rangle + 3(c_f + c_x + 1 + c') \langle\!\langle D \rangle\!\rangle && \text{arithmetic}
\end{aligned}
$$

Case **put** (**get** and **set** are analogous).

$$
\begin{aligned}
&D :: \mathcal{E};\ \sigma; \mathbf{put}\, v \ \Downarrow\ \sigma'; \ell\ ; \mathtt{put}_{\mathcal{E}}^{v \uparrow \ell} \cdot \varepsilon; 1 && \text{derivation} \\
&|D| = 1 \leq \langle \mathbf{put}\, v \rangle + 3(1) \langle\!\langle D \rangle\!\rangle && \text{arithmetic}
\end{aligned}
$$

■

**Corollary 18**
*Fix* $D :: \mathcal{E};\ \sigma; e \ \Downarrow\ \sigma'; v'\ ; T; c$, *then* $|D| \leq (1 + 3 \cdot c) \langle\!\langle D \rangle\!\rangle$.

**Proof:** Immediate from $\langle e \rangle \leq \langle\!\langle D \rangle\!\rangle$ and Theorem 17. ■

## 3.5 Trace Distance

Traces can be used for high-level reasoning about the effectiveness of self-adjusting computation with modrefs by means of a *trace distance* that quantifies the dissimilarities between two runs. Adaptivity, which identifies inconsistencies between runs, is reflected in the trace distance by accounting for differences between traces. Computation memoization, which identifies matching work between runs, is reflected in the trace distance by aligning similar traces.

In Tgt, computation memoization comes in two flavors: *in-order* for programs in which the order of subcomputations is preserved, and *out-of-order* in which the subcomputations may be reordered. Src trace distance comes in two corresponding flavors: *local* trace distance to compare two runs in execution order, which intuitively corresponds to an edit distance, and *global* trace distance which compares two runs modulo reordering, which intuitively corresponds to a hybrid of set difference and edit distance. Global trace distance decomposes runs into fragments and adds the local trace distance of the fragments. As a consequence, both local and global distance are defined for *trace slices*—intuitively, segments of an execution.

$$\frac{}{L \gg L, \bullet} \qquad \frac{S \gg S', \overline{S}'}{M(S) \gg M(S'), \overline{S}'} \qquad \frac{S \gg S', \overline{S}'}{M(S) \gg M(\circ), M(S')\cdot\varepsilon, \overline{S}'} \qquad \frac{}{M(\circ) \gg M(\circ), \bullet}$$

$$\frac{}{\varepsilon \gg \varepsilon, \bullet} \qquad \frac{B \gg B', \overline{S}'_1 \qquad S \gg S', \overline{S}'_2}{B\cdot S \gg B'\cdot S', \overline{S}'_1, \overline{S}'_2}$$

Figure 3.4: Src$*$ decomposition for traces $S \gg S', \overline{S}'$ and actions $B \gg B', \overline{S}'$.

**Trace Slices.** *Actions slices* $B$ and *trace slices* $S$ are the analogues of actions and traces, except that memo actions delimit an *optional trace slice* $\dot{S}$, which, if absent, corresponds to computation that has been reordered.

$$B \ ::= \ L \mid M(\dot{S}) \quad S \ ::= \ \varepsilon \mid B\cdot S \quad \dot{S} \ ::= \ \circ \mid S$$

Note that a trace is also a trace slice with no holes. The notation $\overline{S}$ denotes a list of slices and the metavariable $U$ ranges over non-empty lists of traces. A memo action $M(T)$ can be decomposed into a (skeleton) action slice with a hole $M(\circ)$ and an extracted trace $T$. Figure 3.4 extends this operation to the slicing judgement $S \gg S', \overline{S}'$ to structurally traverse the slice $S$ and decompose it into a (skeleton) slice $S'$ with (nondeterministically) extracted slices $\overline{S}'$. Note that all slices in $\overline{S}'$ are of the form $M(\dot{S})\cdot\varepsilon$, representing a subcomputation of $M$ extracted from $S$.

### 3.5.1 Local Trace Distance

Under in-order reuse, the traces produced by the dynamic and cost semantics are compared in execution order and thus trace distance intuitively captures their edit distance. Consider running a single program under two different stores: intuitively, the executions will be identical up to the first differing TDT operation, after which the traces may correspond to different subprograms (e.g., because a TDT creation allocated different locations or a TDT query produced a different result). In terms of traces, they will have a common prefix up to the first differing TDT action. Conservatively, the only similarity between two runs would be the common prefix. Memo actions, however, enable recognizing similar computations that occur *after* two runs have diverged because they identify the trace of the same function application or same first forcing of a suspension. Nevertheless, two runs of the same expression may have different traces and return different results due to interactions with the store.

$$\frac{}{\varepsilon \boxminus \varepsilon = \langle 0,0 \rangle}$$

$$\frac{S_1 \boxminus S_2 = d}{L{\cdot}S_1 \boxminus S_2 = \langle 1,0 \rangle + d}\ \textbf{search/l/L} \qquad \frac{S_1 \boxminus S_2 = d}{S_1 \boxminus L{\cdot}S_2 = \langle 0,1 \rangle + d}\ \textbf{search/l/R}$$

$$\frac{S_1{\cdot}S_1' \boxminus S_2 = d}{M(S_1){\cdot}S_1' \boxminus S_2 = \langle 1,0 \rangle + d}\ \textbf{search/m/L} \qquad \frac{S_1 \boxminus S_2{\cdot}S_2' = d}{S_1 \boxminus M(S_2){\cdot}S_2' = \langle 0,1 \rangle + d}\ \textbf{search/m/R}$$

$$\frac{S_1' \boxminus S_2 = d}{M(\circ){\cdot}S_1' \boxminus S_2 = \langle 1,0 \rangle + d}\ \textbf{search/none/L} \qquad \frac{S_1 \boxminus S_2' = d}{S_1 \boxminus M(\circ){\cdot}S_2' = \langle 0,1 \rangle + d}\ \textbf{search/none/R}$$

$$\frac{M_1 \approx M_2 \qquad S_1 \ominus S_2 = d \qquad S_1' \boxminus S_2' = d'}{M_1(S_1){\cdot}S_1' \boxminus M_2(S_2){\cdot}S_2' = \langle 1,1 \rangle + d + d'}\ \textbf{search/synch}$$

---

$$\frac{}{\varepsilon \ominus \varepsilon = \langle 0,0 \rangle} \qquad \frac{S_1 \ominus S_2 = d}{L{\cdot}S_1 \ominus L{\cdot}S_2 = d}\ \textbf{synch/l} \qquad \frac{S_1 \ominus S_2 = d \qquad S_1' \ominus S_2' = d'}{M(S_1){\cdot}S_1' \ominus M(S_2){\cdot}S_2' = d + d'}\ \textbf{synch/m}$$

$$\frac{S_1 \boxminus S_2 = d}{S_1 \ominus S_2 = d}\ \textbf{synch/search}$$

Figure 3.5: Src$*$ local search distance $S_1 \boxminus S_2 = d$ and synchronization distance $S_1 \ominus S_2 = d$.

**Example 19**
Two runs of a map function on `[...,0,1,3,4,...]` and `[...,0,1,2,3,4,...]` agree on the prefix `...,0,1` because they aren't sensitive to the new element `2`. Furthermore, even though new computation must be performed for the new element `2`, the two runs agree again on the suffix `3,4,...`.

More generally, comparing two traces alternates between *searching* for a point where traces align (i.e., memo action) and *synchronizing* the two similar traces until they again differ (i.e., local actions). These two complementary ways of scanning traces suggest two corresponding ways for quantifying the distance of two runs. The *synchronization distance* optimistically assumes the two runs are identical and have distance zero. The *search distance* pessimistically assumes the two runs are distinct and have distance proportional to the size of both runs. Since the work common to both runs may be interspersed throughout the two traces, intuitively, the distance between two runs alternates between the synchronization distance of the common work and the search distance of the leftover work.

*Local distance* is formally captured by the search distance $S_1 \boxminus S_2 = d$ and synchronization distance $S_1 \ominus S_2 = d$ judgements (given in Figure 3.5), defined by structural induction on the two trace slices. The search mode *can* switch to synchronization if it encounters similar program fragments (as identified by memo actions), and the synchronization mode *must* switch to search mode if the trace actions differ at some point. Intuitively, the trace distance measures the symmetric difference between two traces (i.e., the size of trace segments that don't occur in both traces). Concretely, we quantify distance $d = \langle c_1, c_2 \rangle$ between traces $S_1$ and $S_2$ as a pair of costs, where $c_1$ is the amount of work in $S_1$ that isn't shared with $S_2$ and $c_2$ is the amount of work in $S_2$ that isn't shared with $S_1$. We let $d + d'$ denote pointwise addition for distance.

Since traces approximate the shape of an evaluation derivation, trace distance approximates a higher-order distance judgement on evaluation derivations that quantifies the dis/similarities of active work between two runs, modulo the stores. The dynamic semantics of Tgt has nondeterministic allocation and memoization in order to avoid committing to an implementation. Consequently, the definition of Src trace distance is a *relation* because the allocation depends on the particular trace chosen and memoization depends on the synchronization chosen in the local distance. We can show that any distance derivable for Src programs is preserved in Tgt (Theorem 40).

**Search Distance.** The search distance $S_1 \boxminus S_2 = d$ accounts for traces that don't match, but switches to synchronization mode if it can align memoization actions. The search distance between empty traces is zero. Upon simultaneously encountering similar memo

actions $M_1(S_1)\cdot S_1'$ and $M_2(S_2)\cdot S_2'$ (**search/synch** rule), the search distance can switch to synchronizing the bodies $S_1$ and $S_2$, while separately searching for further synchronization of the tails $S_1'$ and $S_2'$. Two memo actions are *similar* $M_1 \approx M_2$ if they represent the same computation even if the return values need not coincide. In the case of a memoizing application, both actions must have the same function and argument: $\mathsf{app}^{v_f \$ v_x \Downarrow v_1} \approx \mathsf{app}^{v_f \$ v_x \Downarrow v_2}$. The cost of the synchronization and search are added to the cost of 1 for the memoization match in each trace.

Finally, skipping an action in search mode incurs a cost of 1 in addition to the distance between the tail of the trace (**search/memo** rules and **search/store** rules).

**Synchronization Distance.** Turning to the synchronization distance, the $S_1 \ominus S_2 = d$ judgement attempts to structurally match the two traces. Identical work in both traces incurs no cost, but synchronization returns to search mode when work cannot be reused because traces don't match or nondeterministically otherwise. Synchronization mode is only meant to be used on traces generated by the evaluation of the same expression under (possibly) different stores.

The synchronization distance between empty traces is zero. Encountering identical store actions allows distance to remain in synchronization mode without cost (**synch/l** rule). Synchronizing memo actions (**synch/m** rule) requires the actions to be identical; this allows the bodies as well as the tails to be synchronized separately and their distance compounded. Note that even if the bodies don't match completely and return to search mode, memoizing functions provide a degree of isolation because tails can be matched independently. Synchronization falls back to search mode (**synch/search** rule) necessarily when the actions are distinct (e.g., because local or memo actions don't match) or nondeterministically otherwise.

Observe that the definition of synchronization distance is quasi-symmetric: $S_1 \ominus S_2 = \langle c_1, c_2 \rangle$ iff $S_2 \ominus S_1 = \langle c_2, c_1 \rangle$, and similarly for search distance. Furthermore, note that local distance of Src programs is defined by induction on the two traces: both judgements traverse traces left-to-right either matching work or accounting for skipping it. This means that common work consists of a subsequence of actions that appears in both traces, which precludes reordering work under local distance. For example, comparing runs $\mathsf{app}^{f \$ x \Downarrow a}(\_)\cdot\mathsf{app}^{g \$ y \Downarrow b}(\_)\cdot\_$ and $\mathsf{app}^{g \$ y \Downarrow b}(\_)\cdot\mathsf{app}^{f \$ x \Downarrow a}(\_)\cdot\_$ can only synchronize one of the calls, the other call must be skipped. This restriction avoids having to search for permutations for matching computations and simplifies the implementation requirements of Tgt; however, this limitation obviously hinders the efficiency of self-adjusting computation for certain classes of computations.

### 3.5.2 Global Trace Distance

Under out-of-order reuse, trace distance accounts for reordering and thus trace distance is a hybrid of set difference and edit distance. Intuitively, the difference between two runs can be obtained by globally decomposing each run into a set of subcomputations and locally comparing subcomputations pairwise under some matching. More specifically, globally decomposing a computation slices a trace into a set of traces with holes, and pairwise compares locally traces

The *global distance* $S_1 \boxminus^{\gg} S_2 = d$ between two slices $S_1$ and $S_2$ is obtained by decomposing each slice into the same number of slices, matching slices from each set , and adding up the local distance between each pair of slices:

$$\frac{S_1 \gg \langle S'_{1\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \quad S_2 \gg \langle S'_{2\mathrm{j}} \rangle_{\mathrm{j} \in 1..n} \quad B \in \mathsf{Perm}(n) \quad (\forall (i,j) \in B . S'_{1\mathrm{i}} \boxminus S'_{2\mathrm{j}} = d_{\mathrm{ij}}) \quad d = \sum_{(i,j) \in B} d_{\mathrm{ij}}}{S_1 \boxminus^{\gg} S_2 = d}$$

Each $S_k$ ($k \in 1..2$) is sliced into $n$ slices $\langle S_{ki} \rangle$, which are chosen nondeterministically by the slicing judgement. Next, a bijection $B$ on $n$ elements (i.e., from the set $\mathsf{Perm}(n)$ of permutations on $n$ elements) is chosen nondeterministically. Then for each pair of indices related by the bijection ($(i,j) \in B$), the local distance ($d_{ij}$) is calculated for the corresponding trace slices ($S_{1i}$ and $S_{2j}$). Finally the global distance between the slices is the total of the local trace distances ($d = \sum_{(i,j) \in B} d_{ij}$). Observe that global distance is a relation because of the nondeterminism of slicing the trace slices, choosing a bijection, and due to the nondeterminism of local distance itself.

### 3.5.3 Trace Contexts

To reason compositionally about local distance and obtain asymptotic results, distance is generalized to *trace contexts* $\mathscr{T}$, which are traces with holes.

$$\mathscr{T} \quad ::= \quad \Box \mid A \cdot \mathscr{T} \mid M(\mathscr{T}) \cdot T \mid M(T) \cdot \mathscr{T}$$

Trace context distances $\mathscr{T}_1 \boxminus \mathscr{T}_2 = d$ and $\mathscr{T}_1 \ominus \mathscr{T}_2 = d$ are obtained by lifting distance on traces to trace contexts, extended with the following rules for holes (in the multi-hole analogue, holes are uniquely labeled from left-to-right and labels must also coincide):

$$\Box \boxminus \Box = \langle 0, 0 \rangle \qquad\qquad \Box \ominus \Box = \langle 0, 0 \rangle$$

By requiring holes to coincide when comparing trace contexts, we can reason separately about trace contexts and traces, and then combine the results. Intuitively, the iden-

tity theorem for traces means a common suffix subcomputation incurs no cost. The identity theorem for trace contexts and the substitution theorem show that a common prefix computation does not affect distance either: provided the hole in both trace contexts is immediately bounded by a memoization action of the same function and argument, context and trace distance can be combined additively. The identity theorems are proved by induction on the subject trace $T$ or trace context $\mathscr{T}$. The generalization to multi-holed contexts requires holes to be uniquely labeled in left-to-right order and to line up between two trace contexts.

**Theorem 20 (Identity for Traces)**
*For any trace* $T$, $T \ominus T = \langle 0, 0 \rangle$.

**Proof:** By straightforward induction on the trace $T$.

Case $\varepsilon$. Immediate by search distance rule for $\varepsilon$.

Case $L{\cdot}T$. By the i.h. and **synch/l** rule.

Case $M(T'){\cdot}T$. By the i.h.'s and the **synch/m**.

■

**Theorem 21 (Identity for Trace Contexts)**
*For any trace context* $\mathscr{T}$,
$\mathscr{T}[M(\Box){\cdot}T] \ominus \mathscr{T}[M(\Box){\cdot}T] = \langle 0, 0 \rangle$.

**Proof:** By induction on the trace context $\mathscr{T}$.

Case $\Box$. By the the identity theorem for $T$ and the **synch/m** rule.

The remaining cases appeal to the identity theorem for traces and the i.h..

■

**Theorem 22 (Substitution)**
*Assume* $M_1 \approx M_2$ *and* $T_1' \ominus T_2' = d'$.
*If* $\mathscr{T}_1[M_1(\Box){\cdot}T_1] \boxminus \mathscr{T}_2[M_2(\Box){\cdot}T_2] = d$
*then* $\mathscr{T}_1[M_1(T_1'){\cdot}T_1] \boxminus \mathscr{T}_2[M_2(T_2'){\cdot}T_2] = d + d'$.
*If* $\mathscr{T}_1[M_1(\Box){\cdot}T_1] \ominus \mathscr{T}_2[M_2(\Box){\cdot}T_2] = d$
*then* $\mathscr{T}_1[M_1(T_1'){\cdot}T_1] \ominus \mathscr{T}_2[M_2(T_2'){\cdot}T_2] = d + d'$.

**Proof:** By simultaneous induction on the first derivation of each statement.

Case $\Box \boxminus \Box = d$. By assumption $T_1' \ominus T_2' = d'$.

Case $L{\cdot}\mathscr{T}_1[\Box] \boxminus \mathscr{T}_2[\Box] = \langle 1, 0\rangle + d$.

| | |
|---|---|
| $\mathscr{T}_1[\Box] \boxminus \mathscr{T}_2[\Box] = d$ | subderivation |
| $\mathscr{T}_1[T_1] \boxminus \mathscr{T}_2[T_2] = d + d'$ | i.h. |
| $L{\cdot}\mathscr{T}_1[T_1] \boxminus \mathscr{T}_2[T_2] = d + d'$ | rule |

The remaining cases are analogous, they appeal to the i.h. and replay the distance rule used by the trace contexts.

■

### 3.5.4 Precise Local Trace Distance

The rules of Figure 3.5 are useful for high level reasoning, but aren't rich enough to demonstrate a correspondence with Tgt trace distance. We introduce *failure actions* to explicitly indicate where synchronization mode must switch back to search mode after memoization. Next, we give a alternate rule system for *precise* local Src trace distance that subsumes the above system and serves as an intermediary for proving the preservation of distance under compilation. Precise local distance is exactly matched by Tgt trace distance and asymptotically equivalent to the (simple) local Src trace distance presented above. Finally, we present *evaluation contexts* that label help the translation determine a trace's continuation.

**Failure Actions.** Recall that the **search/synch** rule (given in Figure 3.5) separately synchronizes the bodies and searches the tails when it encounters matching memoization actions. While this rule is useful, it precludes memoization between one body and another tail; for example, it doesn't allow splitting $T_1$ as $T_{11}{\cdot}T_{12}$ and synchronizing $T_{11}$ with a prefix of $T_2$ and searching $T_{12}$ against the rest of $T_2$. The naïve rule:

$$\frac{T_1{\cdot}T_1' \ominus T_2{\cdot}T_2' = d}{\mathtt{app}_{\mathcal{E}_1}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_1}(T_1){\cdot}T_1' \boxminus \mathtt{app}_{\mathcal{E}_2}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_2}(T_2){\cdot}T_2' = \langle 1, 1\rangle + d}\ \textbf{memo/naive}$$

would allow splitting both traces, but it is unsound because it may fully synchronize $T_1{\cdot}T_1'$ with $T_2{\cdot}T_2'$, even though the trace concatenation may *not* have been generated by

$$\frac{S_1{\cdot}\mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}{\cdot}S_1' \boxminus S_2 = d}{M_{\mathcal{E}(\ell)}(S_1){\cdot}S_1' \boxminus S_2 = \langle 1,0\rangle + d}\ \textbf{search/m/L'} \qquad \frac{S_1 \boxminus S_2{\cdot}\mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}{\cdot}S_2' = d}{S_1 \boxminus M_{\mathcal{E}(\ell)}(S_2){\cdot}S_2' = \langle 0,1\rangle + d}\ \textbf{search/m/R'}$$

$$\frac{S_1 \boxminus S_2 = d}{\mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}{\cdot}S_1 \boxminus S_2 = d}\ \textbf{search/fail/L} \qquad \frac{S_1 \boxminus S_2 = d}{S_1 \boxminus \mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}{\cdot}S_2 = d}\ \textbf{search/fail/R}$$

$$\frac{S_1{\cdot}\mathtt{fail}^{\Downarrow v_1}_{\mathcal{E}_1(\ell_1)}{\cdot}S_1' \ominus S_2{\cdot}\mathtt{fail}^{\Downarrow v_2}_{\mathcal{E}_2(\ell_2)}{\cdot}S_2' = d}{\mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_1}_{\mathcal{E}_1(\ell_1)}(S_1){\cdot}S_1' \boxminus \mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_2}_{\mathcal{E}_2(\ell_2)}(S_2){\cdot}S_2' = \langle 1,1\rangle + d}\ \textbf{search/synch'}$$

Figure 3.6: Additional rules for Src distance with explicit failure.

the same expression, violating the well-formedness condition of synchronization distance. We remedy this by introducing the local action for failure to explicitly force the synchronization mode to switch back to search mode.

$$L ::= \cdots \mid \mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}$$

The failure action is labeled by a result $v$, an evaluation context $\mathcal{E}$ expecting the result, and location $\ell$. We also label the memoization action by a location $\ell$. The evaluation context and location of memoization and failure actions can be ignored when reasoning about Src distance; the evaluation contexts is needed to reify the calling context of an action as a continuation and the location is needed to thread the continuation through the store. Since the compilation of Src memoizing functions inserts Tgt memo primitives before and after the function body, failure actions also serve a technical purpose for establishing the correspondence between Src and Tgt traces.

The revised system is obtained by removing the **search/synch** and **search/m/*** rules from Figure 3.5 and adding the rules in Figure 3.6.

The new **search/m/*'** rules insert an explicit failure action between the body and tail of a memoization action, and still incur a cost of 1 for failing to match. The **search/fail** rules allow search to skip a failure action without cost. Observe that, in Figure 3.5, a trace is subjected to synchronization if it is delimited by a memoization action and failure actions never occur in the scope of a memoization action, so failure actions never appear in synchronization mode. Therefore the **search/memo'** and **search/fail** rules subsume the (replaced) **search/memo** rules: any distance derivable from the failure-free deductive system is also derivable from the system with explicit failure.

The **search/synch'** rule identifies matching memoization actions and switches to syn-

chronizing the concatenation of the body, failure action, and tail. Since there are no new synchronization distance rules, leading failure actions force synchronization to switch to search (only the **synch/search** rule applies). Therefore the **search/synch'** rule enables synchronizing part of $T_1$ with $T_2$ and then searching the remainder of $T_1$ against $T_2'$ (after encountering the failure action between $T_2$ and $T_2'$). The **search/synch'** rule subsumes the (replaced) **search/synch** rule.

**Precise Distance.** Since Src actions are translated into multiple Tgt actions (Chapter 5), the simple Src distance presented above uses amortization to avoid exact accounting and to simplify reasoning. We define a variant of Src's distance relation with precise accounting for memoization at function call and return points.

The original Src distance and the new precise Src distance are presented simultaneously in Figure 3.7; the latter extends the former with additional data to exactly account for failure actions and the alternation between search and synchronization. The $T_1 \boxminus T_2 = d; d_f, b_o, d_o$ and $T_1 \ominus T_2 = d; d_f, b_o, d_o$ judgements include the *simple* distance $d$, and the *precise* distance $d_o$ with an auxiliary distance $d_f$ that counts the number of failure actions in each trace and a Boolean flag $b_o$ indicating if synchronization ran to completion. The traces $T_1, T_2$ and the auxiliary distance $d_f$ can be read as inputs to the distance judgements, while the simple distance $d$, flag $b_o$, and precise distance $d_o$ are outputs.

Note that Src traces initially do not contain failure actions, and the number of failure actions introduced by trace distance is bounded by the original distance (*cf.* rules **search/memo'** and **search/synch/flat**). Therefore the following theorem shows that the original Src distance bounds the precise Src distance by a constant factor. The precise Src distance will be related to Tgt distance, thus showing that the original Src distance is preserved in Tgt.

**Lemma 23**
*If* $T_1 \ominus T_2 = \langle 0,0\rangle; \langle 0,0\rangle, b_o, \_$,
*then* $T_1 \ominus T_2 = \langle 0,0\rangle; \langle 0,0\rangle, \mathsf{true}, \_$.

**Proof:** By induction on the distance derivation.

Case $\varepsilon \ominus \varepsilon = \langle 0,0\rangle; \langle 0,0\rangle, \mathsf{true}, \langle 0,0\rangle$. Immediate.

Cases **synch/l** and **synch/m**. By the i.h. on the respective subderivations.

Case **synch/search**. The only possible subderivation is $\varepsilon \boxminus \varepsilon = \langle 0,0\rangle; \langle 0,0\rangle, \mathsf{false}, \langle 0,0\rangle$ which can be turned into $\varepsilon \ominus \varepsilon = \langle 0,0\rangle; \langle 0,0\rangle, \mathsf{true}, \langle 0,0\rangle$.

$$\frac{}{\varepsilon \boxminus \varepsilon = \langle 0,0\rangle; \langle 0,0\rangle, \mathsf{false}, \langle 0,0\rangle}$$

$$\frac{S_1 \boxminus S_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}}{L{\cdot}S_1 \boxminus S_2 = \langle 1,0\rangle + d; d_\mathrm{f}, b_\mathrm{o}, \langle 1,0\rangle + d_\mathrm{o}}\ \textbf{search/l/L}$$

$$\frac{S_1{\cdot}\mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}{\cdot}S_1' \boxminus S_2 = d; d_\mathrm{f} + \langle 2,0\rangle, b_\mathrm{o}, d_\mathrm{o}}{\mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v}_{\mathcal{E}(\ell)}(S_1){\cdot}S_1' \boxminus S_2 = \langle 1,0\rangle + d; d_\mathrm{f}, b_\mathrm{o}, \langle 2,0\rangle + d_\mathrm{o}}\ \textbf{search/m/L}$$

$$\frac{S_1' \boxminus S_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}}{M(\circ){\cdot}S_1' \boxminus S_2 = \langle 1,0\rangle + d; d_\mathrm{f}, b_\mathrm{o}, \langle 4,0\rangle + d_\mathrm{o}}\ \textbf{search/none/L}$$

$$\frac{S_1 \boxminus S_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}}{\mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}{\cdot}S_1 \boxminus S_2 = d; d_\mathrm{f} + \langle 1,0\rangle, b_\mathrm{o}, \langle 2,0\rangle + d_\mathrm{o}}\ \textbf{search/fail/L}$$

$$\frac{S_1 \ominus S_2 = d; \langle 0,0\rangle, \_, d_\mathrm{o} \qquad S_1' \boxminus S_2' = d'; d_\mathrm{f}', b_\mathrm{o}', d_\mathrm{o}'}{\mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_1}_{\mathcal{E}_1(\ell)}(S_1){\cdot}S_1' \boxminus \mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_2}_{\mathcal{E}_2(\ell)}(S_2){\cdot}S_2' = \langle 1,1\rangle + d + d'; d_\mathrm{f}', b_\mathrm{o}', \langle 4,4\rangle + d_\mathrm{o} + d_\mathrm{o}'}\ \textbf{search/synch}$$

$$\frac{S_1{\cdot}\mathtt{fail}^{\Downarrow v_1}_{\mathcal{E}_1(\ell_1)}{\cdot}S_1' \ominus S_2{\cdot}\mathtt{fail}^{\Downarrow v_2}_{\mathcal{E}_2(\ell_2)}{\cdot}S_2' = d; d_\mathrm{f} + \langle 2,2\rangle, b_\mathrm{o}, d_\mathrm{o}}{\mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_1}_{\mathcal{E}_1(\ell_1)}(S_1){\cdot}S_1' \boxminus \mathtt{app}^{v_\mathrm{f}\$v_\mathrm{x}\Downarrow v_2}_{\mathcal{E}_2(\ell_2)}(S_2){\cdot}S_2' = \langle 1,1\rangle + d; d_\mathrm{f}, b_\mathrm{o}, \langle 2,2\rangle + d_\mathrm{o}}\ \textbf{search/synch'}$$

---

$$\frac{}{\varepsilon \ominus \varepsilon = \langle 0,0\rangle; \langle 0,0\rangle, \mathsf{true}, \langle 0,0\rangle} \qquad \frac{S_1 \ominus S_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}}{L{\cdot}S_1 \ominus L{\cdot}S_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}}\ \textbf{synch/l}$$

$$\frac{S_1 \ominus S_2 = d; \langle 0,0\rangle, b_\mathrm{o}, d_\mathrm{o} \qquad S_1' \ominus S_2' = d'; d_\mathrm{f}', b_\mathrm{o}', d_\mathrm{o}'}{M(S_1){\cdot}S_1' \ominus M(S_2){\cdot}S_2' = d + d'; d_\mathrm{f}', b_\mathrm{o}', d_\mathrm{o} + (\mathsf{if}\ b_\mathrm{o}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle) + d_\mathrm{o}'}\ \textbf{synch/m}$$

$$\frac{S_1 \boxminus S_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}}{S_1 \ominus S_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}}\ \textbf{synch/search}$$

Figure 3.7: Src$*$ (simple and precise) search distance $T_1 \boxminus T_2 = d; d_f, b_o, d_o$ (top) (fragment) and synchronization distance $T_1 \ominus T_2 = d; d_f, b_o, d_o$ (bottom).

■

**Theorem 24** (Src **simple/precise soundness)**

1. *Assume* $T_1 \boxminus T_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}$.
   *If* $d = \langle 0,0\rangle$,
   *then* $d_\mathrm{f} = d_\mathrm{o}$,
   *else* $(6 \cdot d + d_\mathrm{f}) \geq (d_\mathrm{o} + \mathsf{if}\ b_\mathrm{o}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle)$ *and* $d_\mathrm{o} \geq d$.

2. *Assume* $T_1 \ominus T_2 = d; d_\mathrm{f}, b_\mathrm{o}, d_\mathrm{o}$,
   *If* $d = \langle 0,0\rangle$,
   *then* $d_\mathrm{f} = d_\mathrm{o}$,
   *else* $(6 \cdot d + d_\mathrm{f}) \geq (d_\mathrm{o} + \mathsf{if}\ b_\mathrm{o}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle)$ *and* $d_\mathrm{o} \geq d$.

**Proof:** By simultaneous induction on the distance derivation of each statement.

We show the cases for **search/synch**, **search/synch'**, **search/m/L'**, and **synch/m**. The remaining cases follow by straightforward induction and arithmetic.

Case **search/synch**.

Subcase $d, d' = \langle 0,0\rangle$.

$d_\mathrm{o} = \langle 0,0\rangle$ i.h.(2) on $D_1$
$d'_\mathrm{o} = d'_\mathrm{f}$ i.h.(1) on $D_2$
$6 \cdot (\langle 1,1\rangle + d + d') + d'_\mathrm{f} \geq (\langle 4,4\rangle + d_\mathrm{o} + d'_\mathrm{o}) + \mathsf{if}\ b'_\mathrm{o}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

Subcase $d = \langle 0,0\rangle \neq d'$.

$d_\mathrm{o} = \langle 0,0\rangle$ i.h.(2) on $D_1$
$6 \cdot d' + d'_\mathrm{f} \geq d'_\mathrm{o} + \mathsf{if}\ b'_\mathrm{o}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ i.h.(1) on $D_2$
$6 \cdot (\langle 1,1\rangle + d + d') + d'_\mathrm{f} = \langle 6,6\rangle + (6 \cdot d' + d'_\mathrm{f})$
$\geq \langle 4,4\rangle + d'_\mathrm{o} + \mathsf{if}\ b'_\mathrm{o}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$
$= (\langle 4,4\rangle + d_\mathrm{o} + d'_\mathrm{o}) + \mathsf{if}\ b'_\mathrm{o}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

Subcase $d \neq \langle 0,0\rangle = d'$.

$6 \cdot d + \langle 0,0 \rangle \geq d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ i.h.(2) on $D_1$
$d'_{\mathrm{o}} = d'_{\mathrm{f}}$ i.h.(1) on $D_2$
$6 \cdot (\langle 1,1 \rangle + d + d') + d'_{\mathrm{f}} = \langle 6,6 \rangle + (6 \cdot d) + d'_{\mathrm{f}}$
$\geq \langle 4,4 \rangle + (d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle) + d'_{\mathrm{f}}$
$= (\langle 4,4 \rangle + d_{\mathrm{o}} + d'_{\mathrm{o}}) + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ arithmetic

Subcase $d, d' \neq \langle 0,0 \rangle$.

$6 \cdot d + \langle 0,0 \rangle \geq d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ i.h.(2) on $D_1$
$6 \cdot d' + d'_{\mathrm{f}} \geq d'_{\mathrm{o}} + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ i.h.(1) on $D_2$
$6 \cdot (\langle 1,1 \rangle + d + d') + d'_{\mathrm{f}} = \langle 6,6 \rangle + (6 \cdot d) + (6 \cdot d' + d'_{\mathrm{f}})$
$\geq \langle 4,4 \rangle + (d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle) + (d'_{\mathrm{o}} + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle)$
$\geq (\langle 4,4 \rangle + d_{\mathrm{o}} + d'_{\mathrm{o}}) + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ arithmetic

All subcases.

$d_{\mathrm{o}} \geq d$ i.h.(2) on $D_1$
$d'_{\mathrm{o}} \geq d'$ i.h.(1) on $D_2$
$\langle 4,4 \rangle + d_{\mathrm{o}} + d'_{\mathrm{o}} \geq \langle 1,1 \rangle + d + d'$ arithmetic

Case **search/synch'**.

Subcase $d = \langle 0,0 \rangle$.

$d_{\mathrm{f}} + \langle 2,2 \rangle = d_{\mathrm{o}}$ i.h.(2) on $D_1$
$6 \cdot \langle 1,1 \rangle + d + d_{\mathrm{f}} = \langle 4,4 \rangle + (d_{\mathrm{f}} + \langle 2,2 \rangle))$
$\geq (\langle 2,2 \rangle + d_{\mathrm{o}}) + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ arithmetic

Subcase $d \neq \langle 0,0 \rangle$.

$6 \cdot d + (d_{\mathrm{f}} + \langle 2,2 \rangle) \geq d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ i.h.(2) on $D_1$
$6 \cdot (\langle 1,1 \rangle + d) + d_{\mathrm{f}} = \langle 4,4 \rangle + (6 \cdot d + (d_{\mathrm{f}} + \langle 2,2 \rangle))$
$\geq (\langle 2,2 \rangle + d_{\mathrm{o}}) + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0 \rangle\ \mathsf{else}\ \langle 2,2 \rangle$ arithmetic

All subcases.

$d_{\mathrm{o}} \geq d$ i.h.
$\langle 2,2\rangle + d_{\mathrm{o}} \geq \langle 1,1\rangle + d$ arithmetic

Case **search/m/L'** (**search/m/R'** is symmetric).

Subcase $d = \langle 0,0\rangle$.

$d_{\mathrm{f}} + \langle 0,2\rangle = d_{\mathrm{o}}$ i.h.(2) on $D_1$
$6 \cdot \langle 1,0\rangle + d + d_{\mathrm{f}} = \langle 4,0\rangle + (d_{\mathrm{f}} + \langle 0,2\rangle)$
$\geq (\langle 2,0\rangle + d_{\mathrm{o}}) + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

Subcase $d \neq \langle 0,0\rangle$.

$6 \cdot d + (d_{\mathrm{f}} + \langle 2,0\rangle) \geq d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ i.h.
$6 \cdot (\langle 1,0\rangle + d) + d_{\mathrm{f}} = \langle 4,0\rangle + (6 \cdot d + d_{\mathrm{f}} + \langle 2,0\rangle)$
$\geq (\langle 2,0\rangle + d_{\mathrm{o}}) + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

All subcases.

$d_{\mathrm{o}} \geq d$ i.h.
$\langle 2,0\rangle + d_{\mathrm{o}} \geq \langle 1,0\rangle + d$ arithmetic

Case **synch/m**.

Subcase $d, d' = \langle 0,0\rangle$.

$\langle 0,0\rangle = d_{\mathrm{o}}$ i.h.(2) on $D_1$
$d'_{\mathrm{f}} = d'_{\mathrm{o}}$ i.h.(2) on $D_2$
$b_{\mathrm{o}} = \mathsf{true}$ wlog by Lemma 23 on $D_1$
$b'_{\mathrm{o}} = \mathsf{true}$ wlog by Lemma 23 on $D_2$
$d'_{\mathrm{f}} = (d_{\mathrm{o}} + (\mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle) + d'_{\mathrm{o}}) + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

Subcase $d = \langle 0,0\rangle \neq d'$.

$\langle 0,0\rangle = d_{\mathrm{o}}$ i.h.(2) on $D_1$
$6 \cdot d' + d'_{\mathrm{f}} \geq d'_{\mathrm{o}} + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ i.h.(2) on $D_2$
$b_{\mathrm{o}} = \mathsf{true}$ wlog by Lemma 23 on $D_1$
$6 \cdot (d + d') + d'_{\mathrm{f}} = 6 \cdot d' + d'_{\mathrm{f}}$
$\geq d'_{\mathrm{o}} + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$
$\geq (d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle + d'_{\mathrm{o}}) + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

Subcase $d \neq \langle 0,0\rangle = d'$.

$6 \cdot d + \langle 0,0\rangle \geq d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ i.h.(2) on $D_1$
$d'_{\mathrm{f}} = d'_{\mathrm{o}}$ i.h.(2) on $D_2$
$b'_{\mathrm{o}} = \mathsf{true}$ wlog by Lemma 23 on $D_2$
$6 \cdot (d + d') + d'_{\mathrm{f}} \geq (d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle) + d'_{\mathrm{f}}$
$\geq (d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle + d'_{\mathrm{o}}) + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

Subcase $d, d' \neq \langle 0,0\rangle$.

$6 \cdot d + \langle 0,0\rangle \geq d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ i.h.(2) on $D_1$
$6 \cdot d' + d'_{\mathrm{f}} \geq d'_{\mathrm{o}} + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ i.h.(2) on $D_2$
$6 \cdot (d + d') + d'_{\mathrm{f}} \geq (d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle) + (d'_{\mathrm{o}} + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle)$
$\geq (d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle + d'_{\mathrm{o}}) + \mathsf{if}\ b'_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle$ arithmetic

All subcases.

$d_{\mathrm{o}} \geq d$ i.h.
$d'_{\mathrm{o}} \geq d'$ i.h.
$d_{\mathrm{o}} + \mathsf{if}\ b_{\mathrm{o}}\ \mathsf{then}\ \langle 0,0\rangle\ \mathsf{else}\ \langle 2,2\rangle + d'_{\mathrm{o}} \geq d + d'$ arithmetic

■

**Evaluation Contexts.** The *evaluation contexts* $\mathcal{E}$ in Src evaluation and traces are necessary for relating Src and Tgt traces in Chapter 5, but can be ignored when reasoning about Src evaluation and distance (in the deductive systems with and without failure). An evaluation context is built up throughout evaluation (Figure 3.3) to capture the shape of the surrounding evaluation derivation, up to the nearest memoizing function application:

$$\mathcal{E} ::= \Box \mid \mathcal{E}\, e_{\mathrm{x}} \mid v_{\mathrm{f}}\, \mathcal{E}$$

The language restriction on the occurrence of expressions avoids explicit forms for case-analysis or TDT manipulation. The evaluation of a memoizing function application extends the context for evaluating the function and argument expressions, but *resets* the context for evaluating the redex; passive $\beta$-reduction (e.g., case-analysis) passes the context unchanged. The accumulated context is used to label the actions with the current context and is used by the ACPS trace translation to reify the continuation.

Intuitively, contexts help identify whether computation *after* a memoizing function application can be reused. The **search/synch** rule ignores the contexts of each trace, the **search/m/*** rules pass the context and result to the failure action. The **synch/l** and **synch/m** rules formally require the contexts to be identical. Since synchronization begins at memo actions $\mathtt{app}^{v_f \$ v_x \Downarrow v_1}_{\mathcal{E}_1}(T_1)$ and $\mathtt{app}^{v_f \$ v_x \Downarrow v_2}_{\mathcal{E}_2}(T_2)$ (*cf.*, **search/synch**), the bodies $T_1$ and $T_2$ result from the evaluation of the *same* expression in the *same* reset context (*cf.*, application evaluation) but under (possibly) different stores. Synchronization distance inductively preserves the property that the two traces being compared result from the *same* expression in the *same* context. In particular, the evaluation contexts and results match in the **synch/memo** rule, so the property holds for the tails justifying why they can be synchronized independently of the bodies. Therefore, contexts in synchronization mode are necessarily equal, and can be ignored when reasoning about Src distance.

## 3.6 SrcLazy

The SrcLazy language extends Src$(-)$ with *suspensions* for lazy (call-by-need) evaluation, which combines delayed evaluation and result memoization. A suspended expression delays its evaluation until it is forced; the first time a suspension is forced, the expression is evaluated and the result is cached; on subsequent forces, the cached result is retrieved. Since the evaluation of suspensions relies on both adaptivity and computation memoization, they constitute active computation (*cf.* Section 3.4) and cannot be formulated as a TDT, which only provides adaptivity. We show how to implement SrcLazy suspensions in terms of SrcImp modrefs and memoizing functions.

$$\frac{\Sigma;\Gamma \vdash^{\$} e : \tau}{\Sigma;\Gamma \vdash^{\$} \mathbf{delay}\, e : \tau\ \mathbf{susp}} \qquad \frac{\Sigma;\Gamma, x : \tau\ \mathbf{susp} \vdash^{\$} e : \tau}{\Sigma;\Gamma \vdash^{\$} \mathbf{recdelay}\, x.e : \tau\ \mathbf{susp}} \qquad \frac{\Sigma;\Gamma \vdash^{\$} v_{\mathrm{l}} : \tau\ \mathbf{susp}}{\Sigma;\Gamma \vdash^{\$} \mathbf{force}\, v_{\mathrm{l}} : \tau}$$

Figure 3.8: SrcLazy typing $\Sigma;\Gamma \vdash^{\delta} e : \tau$ for suspensions.

Lazy programming assumes that the result of a suspension is independent of order of evaluation, which therefore requires the absence of effects for referential transparency (e.g., the result of forcing a suspension shouldn't be sensitive to when it is forced). Therefore, while the suspensions of SrcLazy are orthogonal to TDTs, the language should not be instantiated with any effectful TDTs.

**Syntax.** SrcLazy extends Src$(-)$ syntax with a type for suspensions, constructors to create normal (non-recursive) and recursively-defined suspensions, and a primitive to force a suspension.

$$\begin{array}{rrcl} \text{types} & \tau & ::= & \cdots \mid \tau\ \mathbf{susp} \\ \text{expressions} & e & ::= & \cdots \mid \mathbf{delay}\, e \mid \mathbf{recdelay}\, x.e \mid \mathbf{force}\, v_{\mathrm{l}} \end{array}$$

State constructors are extended with suspended expressions, cached result values, and a black hole to indicate a suspension is in the middle of being forced.

$$\text{state constructors} \quad \mathcal{S} \quad ::= \quad \cdots \mid \mathsf{susp}\, e \mid \mathsf{cache}\, v \mid \mathsf{blackhole}$$

**Static Semantics.** Figure 3.8 gives the static semantics for suspensions. A suspension of type $\tau\ \mathbf{susp}$ is introduced by $\mathbf{delay}\, xe$ and $\mathbf{delay}\, x.e$ if $e$ is an expression of type $\tau$, in the latter case $x$ has the type of the suspension, and eliminated by $\mathbf{force}\, v_l$.

**Traces.** SrcLazy traces extend Src$(-)$ traces with local and memo actions for suspensions. Suspension actions include creation (`delay` and `recdelay`) labeled by the suspended expression and location of the suspension, forcing (`force`) labeled by the location, expression, and result, and cached-result lookup (`cache`) labeled by the location and result.

$$\begin{array}{rrcl} \text{local actions} & L & ::= & \cdots \mid \mathtt{delay}_{\mathcal{E}}^{e\uparrow\ell} \mid \mathtt{recdelay}_{\mathcal{E}}^{x.e\uparrow\ell} \mid \mathtt{cache}_{\mathcal{E}}^{\ell\downarrow v} \\ \text{memoizing actions} & M & ::= & \cdots \mid \mathtt{force}_{\mathcal{E}}^{\ell,e\Downarrow v} \end{array}$$

For the purposes of trace distance, two first forces are similar if they operate on the same location and expression, even if they don't return the same result: $\mathtt{force}^{\ell,e\Downarrow v_1} \approx \mathtt{force}^{\ell,e\Downarrow v_2}$.

$$\frac{\ell \notin \operatorname{dom}\sigma \qquad \sigma' = \sigma[\ell \mapsto \mathsf{susp}\, e]}{\mathcal{E};\ \sigma; \mathbf{delay}\, e \ \Downarrow\ \sigma'; \ell\ ; \mathtt{delay}_{\mathcal{E}}^{e\uparrow\ell} \cdot \varepsilon; 1}\ \textbf{delay}$$

$$\frac{\ell \notin \operatorname{dom}\sigma \qquad \sigma' = \sigma[\ell \mapsto \mathsf{susp}\,([\,\ell\,/\,x\,]\,e)]}{\mathcal{E};\ \sigma; \mathbf{recdelay}\, x.e \ \Downarrow\ \sigma'; \ell\ ; \mathtt{recdelay}_{\mathcal{E}}^{x.e\uparrow\ell} \cdot \varepsilon; 1}\ \textbf{recdelay}$$

$$\frac{\begin{array}{c}\ell \in \operatorname{dom}\sigma \qquad \sigma(\ell) = \mathsf{susp}\, e \\ \sigma_{\mathrm{h}} = \sigma[\ell \mapsto \mathsf{blackhole}] \qquad \mathcal{E};\ \sigma_{\mathrm{h}}; e \ \Downarrow\ \sigma'_{\mathrm{h}}; v\ ; T'; c' \qquad \sigma' = \sigma'_{\mathrm{h}}[\ell \mapsto \mathsf{cache}\, v]\end{array}}{\mathcal{E};\ \sigma; \mathbf{force}\, \ell \ \Downarrow\ \sigma'; v\ ; \mathtt{force}_{\mathcal{E}}^{\ell, e \Downarrow v}(T') \cdot \varepsilon; 1 + c'}\ \textbf{force/miss}$$

$$\frac{\begin{array}{c}\ell \in \operatorname{dom}\sigma \\ \sigma(\ell) = \mathsf{cache}\, v\end{array}}{\mathcal{E};\ \sigma; \mathbf{force}\, \ell \ \Downarrow\ \sigma; v\ ; \mathtt{cache}_{\mathcal{E}}^{\ell \downarrow v} \cdot \varepsilon; 1}\ \textbf{force/hit}$$

Figure 3.9: SrcLazy evaluation $\mathcal{E};\ \sigma; e \ \Downarrow\ \sigma'; v'\ ; T; c$ (cost) for suspensions.

**Dynamic Semantics.** Figure 3.9 gives the dynamic and cost semantics for suspensions. A suspension can be created by $\mathbf{delay}\, e$ and $\mathbf{delay}\, x.e$ to extend the store with a fresh location initialized with the delayed expression $e$, in the latter case the suspension is recursively bound to variable $x$ for cyclical data structures.

Laziness is extensionally pure because forcing a suspension always returns the same result, but it is intensionally effectful—as captured in the trace—because forcing the first time performs the computation and stores the cached result, while subsequent forces only fetch the cached result. Forcing a suspension for the first time finds an unevaluated expression $\mathsf{susp}\, e$ in the store, which is reset to contain a blackhole state, the expression is evaluated, and the location is updated with the cached result $\mathsf{cache}\, v$ state. Forcing an already-evaluated suspension returns the cached result $v$ from the state $\mathsf{cache}\, v$. Forcing an in-evaluation suspension, however, finds a blackhole state and has undefined behavior; a precise specification could define it to diverge or yield an error. In each case, the trace is the action corresponding to the primitive and the work is 1, except for forcing the first time which delimits the trace of evaluating the suspended expression and includes its cost.

**Translation.** The suspension primitives of SrcLazy are useful for lazy programming and high-level reasoning about trace distance, but for technical convenience we give a semantics- and trace distance-preserving translation from SrcLazy to SrcImp. To translate

$$[\![\tau\ \mathbf{susp}]\!]^{\mathrm{s}} = [\![\tau]\!]^{\mathrm{s}}\ \mathbf{dsusp}\ \mathbf{modref}$$

$$\begin{aligned}
[\![\mathbf{delay}\ e]\!]^{\mathrm{s}} &= \mathbf{put}\ (\mathbf{susp}\ (\mathbf{mfun}\ y_{\mathrm{f}}.y_{\mathrm{u}}.\ [\![e]\!]^{\mathrm{s}})) \\
[\![\mathbf{recdelay}\ x.e]\!]^{\mathrm{s}} &= \\
&\quad \mathbf{let}\ x = \mathbf{put}\ \mathbf{blackhole} \\
&\qquad _ = \mathbf{set}\ x\ (\mathbf{susp}\ (\mathbf{mfun}\ y_{\mathrm{f}}.y_{\mathrm{u}}.\ [\![e]\!]^{\mathrm{s}})) \\
&\quad \mathbf{in}\ x \\
[\![\mathbf{force}\ v_{\mathrm{s}}]\!]^{\mathrm{s}} &= \\
&\quad \mathbf{let}\ x_{\mathrm{s}} = \mathbf{get}\ [\![v_{\mathrm{s}}]\!]^{\mathrm{s}} \\
&\quad \mathbf{in\ caseS}\ x_{\mathrm{s}} \\
&\qquad |\ \mathbf{blackhole} \Rightarrow \mathbf{diverge} \\
&\qquad |\ \mathbf{susp}\ x_{\mathrm{f}} \Rightarrow \\
&\qquad\quad \mathbf{let}\ _ = \mathbf{set}\ [\![v_{\mathrm{s}}]\!]^{\mathrm{s}}\ \mathbf{blackhole} \\
&\qquad\qquad x = x_{\mathrm{f}}\ () \\
&\qquad\qquad _ = \mathbf{set}\ [\![v_{\mathrm{s}}]\!]^{\mathrm{s}}\ (\mathbf{cache}\ x) \\
&\qquad\quad \mathbf{in}\ x \\
&\qquad |\ \mathbf{cache}\ x_{\mathrm{res}} \Rightarrow x_{\mathrm{res}}
\end{aligned}$$

$$\begin{aligned}
\left[\!\left[\mathtt{delay}_{\mathcal{E}}^{e\uparrow\ell}\cdot S\right]\!\right]^{\mathrm{s}} &= \mathtt{put}_{\mathcal{E}}^{\mathbf{susp}\ (\mathbf{mfun}\ y_{\mathrm{f}}.y_{\mathrm{u}}.[\![e]\!]^{\mathrm{s}})\uparrow\ell}\cdot [\![S]\!]^{\mathrm{s}} \\
\left[\!\left[\mathtt{recdelay}_{\mathcal{E}}^{x.e\uparrow\ell}\cdot S\right]\!\right]^{\mathrm{s}} &= \mathtt{put}_{\mathcal{E}}^{\mathbf{blackhole}\uparrow\ell}\cdot\mathtt{set}_{\mathcal{E}}^{\ell\leftarrow v_{\mathrm{f}}}\cdot [\![S]\!]^{\mathrm{s}} \\
\left[\!\left[\mathtt{force}_{\mathcal{E}}^{\ell,e\Downarrow v}(S_{\mathrm{e}})\cdot S\right]\!\right]^{\mathrm{s}} &= \mathtt{get}_{\mathcal{E}}^{\ell\downarrow\mathbf{susp}\ v_{\mathrm{f}}}\cdot\mathtt{set}_{\mathcal{E}}^{\ell\leftarrow\mathbf{blackhole}} \\
&\qquad \cdot\mathtt{app}_{\mathcal{E}}^{v_{\mathrm{f}}\$()\Downarrow[\![v]\!]^{\mathrm{s}}}([\![S_{\mathrm{e}}]\!]^{\mathrm{s}})\cdot\mathtt{set}_{\mathcal{E}}^{\ell\leftarrow\mathbf{cache}\ [\![v]\!]^{\mathrm{s}}}\cdot [\![S]\!]^{\mathrm{s}} \\
&\quad \text{where}\ v_{\mathrm{f}} = \mathbf{mfun}\ y_{\mathrm{f}}.y_{\mathrm{u}}.\ [\![e]\!]^{\mathrm{s}} \\
\left[\!\left[\mathtt{cache}_{\mathcal{E}}^{\ell\downarrow v}\cdot S\right]\!\right]^{\mathrm{s}} &= \mathtt{get}_{\mathcal{E}}^{\ell\downarrow\mathbf{cache}\ [\![v]\!]^{\mathrm{s}}}\cdot [\![S]\!]^{\mathrm{s}}
\end{aligned}$$

Figure 3.10: Translation from SrcLazy to SrcImp (fragment).

from SrcLazy to SrcImp, the latter must be extended with a unit type and a datatype for suspensions, which yield passive computation.

$$\begin{array}{rrcl} \text{types} & \tau & ::= & \cdots \mid \mathbf{unit} \mid \tau\ \mathbf{dsusp} \\ \text{expressions} & e & ::= & \cdots \mid \mathbf{caseS}\, v\, e_{\mathrm{bh}}\, (x_{\mathrm{d}}.e_{\mathrm{d}})\, (x_{\mathrm{c}}.e_{\mathrm{c}}) \\ \text{values} & v & ::= & \cdots \mid () \mid \mathbf{susp}\, v \mid \mathbf{cache}\, v \mid \mathbf{blackhole} \end{array}$$

Intuitively, the datatype constructors simulate states for suspensions. The $\tau\ \mathbf{dsusp}$ datatype has constructors $\mathbf{susp}\, v$ for a thunk $v$ of type $\mathbf{unit} \xrightarrow{\$} \tau$ of an unevaluated suspension, and $\mathbf{cache}\, v$ for the cached value $v$ of type $\tau$ of an evaluated suspension, and $\mathbf{blackhole}$ to indicate a suspension is being evaluated and thus identify a circular dependency if it is evaluated a second time. The $\mathbf{caseS}$ elimination form analyzes a value $v$ of type $\tau\ \mathbf{dsusp}$ and dispatches to the branch prescribed by the scrutinee. The static, dynamic and cost semantics are straightforward.

Figure 3.10 shows the translation $[\![-]\!]^{\mathrm{s}}$ from SrcLazy to SrcImp for types $[\![\tau^{\mathsf{lazy}}]\!]^{\mathrm{s}} = \tau^{\mathsf{imp}}$, expressions $[\![e^{\mathsf{lazy}}]\!]^{\mathrm{s}} = e^{\mathsf{imp}}$, and trace slices $[\![S^{\mathsf{lazy}}]\!] = S^{\mathsf{imp}}$, using ML-style let-binding and pattern-matching syntactic sugar, and metavariable $y$ for identifiers introduced by the translation. The omitted cases of the translation are structural.

A suspension type is translated to reference containing a datatype. A (non-recursive) suspension creates a reference with a thunk for the expression. A recursive suspension creates a reference with a black hole, then updates it with a thunk for the expression with the variable substituted with the reference, and returns the reference. Forcing a suspension case-analyzes the contents of the reference: a black hole leads to divergence, consistent with the undefined behavior in the semantics; finding a thunk replaces the reference with a black hole, runs the thunk, updates the reference with the cached result, and returns the result; finding a cached result returns the result. The type translation is extended pointwise to SrcLazy store and variable typing contexts $\Sigma$ and $\Gamma$; the value translation is extended pointwise to SrcLazy stores $\sigma$. The trace translation $[\![S^{\mathsf{lazy}}]\!] = S^{\mathsf{imp}}$ is uniquely determined by the expression translation.

**Meta-Theory.** The translation preserves the static semantics, asymptotically the dynamic and cost semantics, and asymptotically the trace distance. The correctness of the translation crucially depends on the use of a thunk in the translation of delaying an expression: forcing the thunk produces a memoizing action for function application that corresponds to the one for forcing.

**Theorem 25 (Suspension Typing Soundness)**
*If* $\Sigma; \Gamma \vdash^{\delta} e : \tau$, *then* $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}} \vdash^{\delta} [\![e]\!]^{\mathrm{s}} : [\![\tau]\!]^{\mathrm{s}}$.

**Proof:** By induction on the typing derivation.

Case $\Sigma; \Gamma \vdash^{\$} \mathbf{force}\, v_{\mathrm{s}} : \tau$, subcase $\mathsf{susp}\, e$.

| | |
|---|---|
| $\Sigma; \Gamma \vdash^{\$} v_{\mathrm{l}} : \tau\ \mathbf{susp}$ | subderivation |
| $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}} \vdash^{\$} [\![v_{\mathrm{s}}]\!]^{\mathrm{s}} : [\![\tau]\!]^{\mathrm{s}}\ \mathbf{dsusp}\ \mathbf{modref}$ | i.h. |
| $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}} \vdash^{\$} \mathbf{get}\ [\![v_{\mathrm{s}}]\!]^{\mathrm{s}} : [\![\tau]\!]^{\mathrm{s}}\ \mathbf{dsusp}$ | typing |
| $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}} \vdash^{\$} \mathbf{set}\ [\![v_{\mathrm{s}}]\!]^{\mathrm{s}}\ \mathbf{blackhole} : \mathbf{unit}$ | typing |
| $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}}, x_{\mathrm{f}} : \mathbf{unit} \xrightarrow{\$} [\![\tau]\!]^{\mathrm{s}} \vdash^{\$} x_{\mathrm{f}}\, \$\, () : [\![\tau]\!]^{\mathrm{s}}$ | typing |
| $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}}, x : [\![\tau]\!]^{\mathrm{s}} \vdash^{\$} \mathbf{set}\ [\![v_{\mathrm{s}}]\!]^{\mathrm{s}}\ (\mathbf{cache}\, x) : \mathbf{unit}$ | typing |
| $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}}, x : [\![\tau]\!]^{\mathrm{s}} \vdash^{\$} x : [\![\tau]\!]^{\mathrm{s}}$ | typing |
| $[\![\Sigma]\!]^{\mathrm{s}}; [\![\Gamma]\!]^{\mathrm{s}} \vdash^{\$} [\![\mathbf{force}\, v_{\mathrm{s}}]\!]^{\mathrm{s}} : [\![\tau]\!]^{\mathrm{s}}$ | typing |

The $\mathbf{force}\, \ell$ subcases $\mathsf{cache}\, v$ and $\mathsf{blackhole}$, and the $\mathbf{delay}\, e$ and $\mathbf{recdelay}\, x.e$ cases are analogous.

The remaining cases follow immediately by the induction hypothesis.

■

**Theorem 26 (Suspension Evaluation Soundness)**
*If* $\mathcal{E};\ \sigma; e \Downarrow \sigma'; v'; T; c$,
*then* $\mathcal{E};\ [\![\sigma]\!]^{\mathrm{s}}; [\![e]\!]^{\mathrm{s}} \Downarrow [\![\sigma']\!]^{\mathrm{s}}; [\![v']\!]^{\mathrm{s}}; [\![T]\!]^{\mathrm{s}}; c'$
*and* $c \leq c' \leq 4 \cdot c$, *whence* $c' \in \Theta(c)$.

**Proof:** By induction on the evaluation derivation.

Case $\mathcal{E};\ \sigma; \mathbf{force}\, \ell \Downarrow \sigma'; v; \mathtt{force}^{\ell, e \Downarrow v}_{\mathcal{E}}(T') \cdot \varepsilon; 1 + c_{\mathrm{h}}$, subcase $\mathsf{susp}\, e$.

| | |
|---|---|
| $\ell \in \mathrm{dom}\, \sigma$ | subderivation |
| $\sigma(\ell) = \mathsf{susp}\, e$ | subderivation |
| $[\![\sigma]\!]^{\mathrm{s}}(\ell) = \mathbf{susp}\, (\mathbf{mfun}\, y_{\mathrm{f}}.y_{\mathrm{u}}.\, [\![e]\!]^{\mathrm{s}})$ | translation |
| $\mathcal{E};\ [\![\sigma]\!]^{\mathrm{s}}; \mathbf{get}\, \ell \Downarrow \sigma; \mathbf{susp}\, (\mathbf{mfun}\, y_{\mathrm{f}}.y_{\mathrm{u}}.\, [\![e]\!]^{\mathrm{s}}); \mathtt{get}^{\ell \downarrow \mathbf{susp}\, (\mathbf{mfun}\, y_{\mathrm{f}}.y_{\mathrm{u}}.[\![e]\!]^{\mathrm{s}})}_{\mathcal{E}} \cdot \varepsilon; 1$ | evaluation |
| $\sigma_{\mathrm{h}} = \sigma[\ell \mapsto \mathsf{blackhole}]$ | subderivation |
| $\mathcal{E};\ [\![\sigma]\!]^{\mathrm{s}}; \mathbf{set}\, \ell\, \mathbf{blackhole} \Downarrow [\![\sigma_{\mathrm{h}}]\!]^{\mathrm{s}}; (); \mathtt{set}^{\ell, \mathbf{blackhole} \downarrow ()}_{\mathcal{E}} \cdot \varepsilon; 1$ | evaluation |

$\mathcal{E};\ \sigma_{\mathrm{h}}; e \ \Downarrow\ \sigma'_{\mathrm{h}}; v\ ; T'; c'$ — subderivation

$\mathcal{E};\ [\![\sigma_{\mathrm{h}}]\!]^{\mathrm{s}}\ ; [\![e]\!]^{\mathrm{s}}\ \Downarrow\ [\![\sigma'_{\mathrm{h}}]\!]^{\mathrm{s}}\ ; [\![v]\!]^{\mathrm{s}}\ ; [\![T']\!]^{\mathrm{s}}\ ; c'_{\mathrm{h}}$ — i.h.

$c_{\mathrm{h}} \leq c'_{\mathrm{h}} \leq 4 \cdot c_{\mathrm{h}}$ — i.h.

$\mathcal{E};\ [\![\sigma_{\mathrm{h}}]\!]^{\mathrm{s}}\ ; (\mathbf{mfun}\ y_{\mathrm{f}}.\, y_{\mathrm{u}}.\, [\![e]\!]^{\mathrm{s}})\ \$\ ()\ \Downarrow\ [\![\sigma'_{\mathrm{h}}]\!]^{\mathrm{s}}\ ; [\![v]\!]^{\mathrm{s}}\ ; \mathtt{app}_{\mathcal{E}}^{v_{\mathrm{f}}\$()\Downarrow[\![v]\!]^{\mathrm{s}}}([\![T']\!]^{\mathrm{s}})\cdot\varepsilon; 1 + c'_{\mathrm{h}}$ — evaluation

$\sigma' = \sigma'_{\mathrm{h}}[\ell \mapsto \mathsf{cache}\, v]$ — subderivation

$\mathcal{E};\ [\![\sigma'_{\mathrm{h}}]\!]^{\mathrm{s}}\ ; \mathbf{set}\ \ell\ [\![v]\!]^{\mathrm{s}}\ \Downarrow\ [\![\sigma']\!]^{\mathrm{s}}\ ; ()\ ; \mathtt{set}_{\mathcal{E}}^{\ell, [\![v]\!]^{\mathrm{s}}\downarrow()}\cdot\varepsilon; 1$ — evaluation

$\mathcal{E};\ [\![\sigma]\!]^{\mathrm{s}}\ ; [\![\mathbf{force}\,\ell]\!]^{\mathrm{s}}\ \Downarrow\ [\![\sigma']\!]^{\mathrm{s}}\ ; [\![v]\!]^{\mathrm{s}}\ ; \left[\!\!\left[\mathtt{force}_{\mathcal{E}}^{\ell, e\Downarrow v}(T')\cdot\varepsilon\right]\!\!\right]^{\mathrm{s}}\ ; 4 + c_{\mathrm{h}}$ — evaluation

$1 + c_{\mathrm{h}} \leq 4 + c'_{\mathrm{h}} \leq 4 \cdot (1 + c_{\mathrm{h}})$ — arithmetic

The $\mathbf{force}\,\ell$ subcases $\mathsf{cache}\, v$ and $\mathsf{blackhole}$, and the $\mathbf{delay}\, e$ and $\mathbf{recdelay}\, x.e$ cases are analogous, each incurs an additional cost of 1 or 2 depending on the number of modref operations in their translation.

The remaining cases follow immediately by the induction hypothesis.

■

**Theorem 27 (Suspension Local Distance Soundness)**
*If* $S_1 \boxminus S_2 = d$, *then* $[\![S_1]\!]^{\mathrm{s}} \boxminus [\![S_2]\!]^{\mathrm{s}} = d'$ *and* $d \leq d' \leq 4 \cdot d$, *whence* $d' \in \Theta(d)$.
*If* $S_1 \ominus S_2 = d$, *then* $[\![S_1]\!]^{\mathrm{s}} \ominus [\![S_2]\!]^{\mathrm{s}} = d'$ *and* $d \leq d' \leq 4 \cdot d$, *whence* $d' \in \Theta(d)$.

**Proof:** By simultaneous induction on the local distance derivation of each statement.

Cases **search/l/***, **search/m/***, and **search/none/*** involving suspension actions. The translation of a suspension action increments the cost by 1, 2, or 4 depending on the number of actions in its translation.

Case **search/synch** with similar force actions. The translation incurs a cost of 3 for the search discarding the `get` and two `set` actions of each trace slice, plus a cost of 1 for the synchronization of the memoized application; the distance for the translated body and tail slices are asymptotically the same by the i.h..

Cases **synch/l** and **synch/m**. The translation uses the same rules multiple times for the actions of the translation; the distance for the translated body and tail slices are asymptotically the same by the i.h..

The remaining cases follow immediately by the induction hypothesis.

■

**Theorem 28**
*If* $S_1 \boxminus^{\gg} S_2 = d$, *then* $[\![S_1]\!]^{\mathrm{s}} \boxminus^{\gg} [\![S_2]\!]^{\mathrm{s}} = d'$ *and* $d \leq d' \leq 4 \cdot d$, *whence* $d' \in \Theta(d)$.

**Proof:** By induction on the global distance derivation. We choose the analogous slicing for the translated trace slices and appeal to the translation result for local distance. ■

# Chapter 4

# The Tgt$*$ Languages

This chapter is based on work on a CPS language for self-adjusting computation with single-write modrefs [Ley-Wild et al., 2008b], a cost semantics and trace distance for a language with multi-write modrefs [Ley-Wild et al., 2009], and extensibility to traceable data types [Acar et al., 2010a].

## 4.1 Overview

The schematic Tgt($T$) language is a simply-typed, call-by-value $\lambda$-calculus that enforces a continuation-passing style (CPS) Appel [1991] discipline to help identify opportunities for reuse and computations for re-execution. The language is parameterized by a traceable data type $T$ (possibly several) specified by a signature. A TDT induces continuation-passing TDT operations in the Tgt language that use an indirection through the store to make dynamic data dependencies explicit and help identify which parts of an execution may be reused or must be recomputed. The language also includes a computation memoization primitive to identify opportunities for computation reuse across runs. We use Tgt$*$ to refer collectively to Tgt($-$) with TDTs, and Tgt to refer to a representative one.

Tgt$*$ is self-adjusting: its semantics includes evaluation and change-propagation judgements that can be used to reduce expressions to values and adapt computations to input changes. To update a program's output in response to changes in its input, the change-propagation mechanism re-executes the portions of the computation affected by the changes and reuses the unaffected portions. The dynamic and cost semantics produces an *execution trace* for change-propagating one evaluation into another, and also serves to quantify the abstract cost of change-propagation as a *trace distance* between runs.

The language supports both *in-order* computation memoization with TDTs and *out-of-order* computation memoization with mutable modrefs. In-order reuse requires segments of a previous computation to be used in the same execution order, which is captured by *local* trace distance. Out-of-order computation reuse allows the trace of a previous run can be sliced into subcomputations and reused with reordering, which is captured by *global* trace distance.

Chapter 5 shows how Src programs are compiled into self-adjusting Tgt programs by a CPS transformation that uses Src annotations to insert primitives for self-adjusting computation. The CPS transform sequentializes computations and names intermediate values, which isolate where TDTs should be created and used. The transform also reifies control flow as a continuation that represents the rest of the computation, this gives a conservative approximation to the scope of a modifiable dereference and allows memoizing the tail of a computation.

## 4.2 Syntax

The syntax of Tgt($T$) is given by the following grammar, which defines types $\tau$, expressions $e$, values $v$, and commands $\kappa$, using metavariables $f$ and $x$ for identifiers and $\ell$ for locations.

$$
\begin{array}{rrcl}
\text{types} & \tau & ::= & \mathbf{res} \mid \mathbf{nat} \mid \tau_{\mathrm{x}} \to \tau \mid \tau\,\mathbf{tdt} \\
\text{expressions} & e & ::= & v \mid \mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, (x.e_{\mathrm{s}}) \mid e_{\mathrm{f}}\, e_{\mathrm{x}} \\
\text{values} & v & ::= & x \mid \mathbf{zero} \mid \mathbf{succ}\, v \mid \mathbf{fun}\, f.x.e \mid \ell \mid \kappa \\
\text{commands} & \kappa & ::= & \mathbf{halt}\, v \mid \mathbf{memo}\, e \mid \mathbf{mk\_k}\, v_{\mathrm{mk}}\, v_{\mathrm{k}} \mid \mathbf{op\_k}\, v_{\mathrm{l}}\, v_{\mathrm{arg}}\, v_{\mathrm{k}} \\
& \lambda x.e & \stackrel{\text{def}}{=} & \mathbf{fun}\, f.x.e \qquad \text{with } f \notin \mathrm{FV}(e)
\end{array}
$$

Tgt enforces a continuation-passing style (CPS) discipline to help identify opportunities for reuse and computations for re-execution.

The CPS discipline allows pure computations (e.g., natural numbers and recursive functions) to be introduced by values and eliminated by expressions, with the $\mathbf{caseN}$ scrutinee. The $\mathbf{caseN}$ primitive case-analyzes a natural number $v_n$ and branches to $e_z$ or $e_s$ according to whether it is zero or a successor number. Functions are classified by the $\tau_x \to \tau$ type, introduced with the **fun** keyword, and eliminated by juxtaposition. In contrast to Src with normal and adaptive functions, Tgt only has one form of functions because adaptivity is directly handled by commands. Function application allows both the function and argument to be expressions. Self-adjusting programs, however, restrict the argument to be a value (*cf.* Chapter 5) to respect the CPS discipline.

Commands $\kappa$ follow a continuation-passing discipline that linearizes the computation and explicitly identifies adaptivity and computation memoization. The type $\mathbf{res}$ is an opaque answer type, while $\mathbf{halt}$ is a continuation that injects a final value into the $\mathbf{res}$ type. The dynamic semantics identifies opportunities for computation reuse at $\mathbf{memo}$ commands, which enable replaying the trace of a previous run.

Since adaptivity identifies the need for recomputation, Tgt programs use an indirection through the store to manipulate TDTs and isolate the differences between computations. TDT commands in Tgt are formulated in CPS with an explicit continuation $v_k$ identifying the computation that follows the command and use an indirection through the store to isolate data dependencies. A creation command $\mathbf{mk\_k}\ v_{mk}\ v_k$ initializes a TDT instance with seed value $v_{mk}$, and a manipulation (i.e., query or update) command $\mathbf{op\_k}\ v_l\ v_{arg}\ v_k$ takes a reference $v_l$ to a TDT instance in the store and an argument value $v_{arg}$. The invoke and revoke operations of TDTs are used by the semantics of the Tgt language to identify which parts of a computation—i.e., which actions of the trace—are affected by input changes.

As in Src, we take a store $\sigma$ to be a finite map of locations to TDT state constructors $\mathcal{S}$; the notation $\sigma[\ell \mapsto \mathcal{S}]$ denotes the store $\sigma$ updated with $\ell$ mapped to $\mathcal{S}$. Contexts $\Gamma$ and $\Sigma$ are maps from variables and locations to types, respectively.

## 4.3 Static Semantics

Figure 4.1 gives the static semantics of Tgt. The typing judgement $\Sigma; \Gamma \vdash e : \tau$ ascribes the type $\tau$ to the expression $e$ in the store and variable typing contexts $\Sigma$ and $\Gamma$.

The type of a variable (resp. location) is determined by the variable (resp. store) typing context, independent of the mode.

The type of natural number values is determined structurally. A case-analysis must have a natural number scrutinee and both branches must agree on the type.

A function is well-typed if the body of the function has the correct result type by extending the context with the function and argument variables. An application has the function expression's result type if the function expression has a function type and the argument expression matches the argument type.

Commands have answer type $\mathbf{res}$ if their arguments have the correct type. A $\mathbf{halt}$ command injects any well-typed value into the $\mathbf{res}$ type. A $\mathbf{memo}$ command preserves the $\mathbf{res}$ type of any expression. An adaptive TDT command requires the arguments to match the types prescribed by the TDT signature. A creation command $\mathbf{mk}$ must have

$$\frac{x : \tau \in \Gamma}{\Sigma; \Gamma \vdash x : \tau} \qquad \frac{}{\Sigma; \Gamma \vdash \mathbf{zero} : \mathbf{nat}} \qquad \frac{\Sigma; \Gamma \vdash v : \mathbf{nat}}{\Sigma; \Gamma \vdash \mathbf{succ}\, v : \mathbf{nat}}$$

$$\frac{\Sigma; \Gamma, f : \tau_{\mathrm{x}} \to \tau, x : \tau_{\mathrm{x}} \vdash e : \tau}{\Sigma; \Gamma \vdash \mathbf{fun}\, f.x.e : \tau} \qquad \frac{\ell : \tau \in \Sigma}{\Sigma; \Gamma \vdash \ell : \tau}$$

$$\frac{\Sigma; \Gamma \vdash v_{\mathrm{n}} : \mathbf{nat} \qquad \Sigma; \Gamma \vdash e_{\mathrm{z}} : \tau \qquad \Sigma; \Gamma, x : \mathbf{nat} \vdash e_{\mathrm{s}} : \tau}{\Sigma; \Gamma \vdash \mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, (x.e_{\mathrm{s}}) : \tau}$$

$$\frac{\Sigma; \Gamma \vdash e_{\mathrm{f}} : \tau_{\mathrm{x}} \to \tau \qquad \Sigma; \Gamma \vdash e_{\mathrm{x}} : \tau_{\mathrm{x}}}{\Sigma; \Gamma \vdash e_{\mathrm{f}}\, e_{\mathrm{x}} : \tau}$$

$$\frac{\Sigma; \Gamma \vdash v : \tau}{\Sigma; \Gamma \vdash \mathbf{halt}\, v : \mathbf{res}} \qquad \frac{\Sigma; \Gamma \vdash e : \mathbf{res}}{\Sigma; \Gamma \vdash \mathbf{memo}\, e : \mathbf{res}}$$

$$\frac{\begin{array}{c}\mathbf{mk} : \tau_{\mathrm{mk}} \to \tau\, \mathbf{tdt} \in \Delta_{\mathrm{tdt}} \\ \Sigma; \Gamma \vdash v_{\mathrm{mk}} : \tau_{\mathrm{mk}} \\ \Sigma; \Gamma \vdash v_{\mathrm{k}} : \tau\, \mathbf{tdt} \to \mathbf{res}\end{array}}{\Sigma; \Gamma \vdash \mathbf{mk\_k}\, v_{\mathrm{mk}}\, v_{\mathrm{k}} : \mathbf{res}} \qquad \frac{\begin{array}{c}\mathbf{op} : \tau_{\mathrm{arg}} \to \tau_{\mathrm{res}} \in \Delta_{\mathrm{tdt}} \\ \Sigma; \Gamma \vdash v_{\mathrm{l}} : \tau\, \mathbf{tdt} \\ \Sigma; \Gamma \vdash v_{\mathrm{arg}} : \tau_{\mathrm{arg}} \\ \Sigma; \Gamma \vdash v_{\mathrm{k}} : \tau_{\mathrm{res}} \to \mathbf{res}\end{array}}{\Sigma; \Gamma \vdash \mathbf{op\_k}\, v_{\mathrm{l}}\, v_{\mathrm{arg}}\, v_{\mathrm{k}} : \mathbf{res}}$$

Figure 4.1: Tgt typing $\Sigma; \Gamma \vdash e : \tau$.

an argument that matches the seed type and a continuation that expects a TDT. A manipulation command $\mathbf{op}$ must have a location argument of the TDT type, an argument and continuation matching the types specified by the signature.

## 4.4 Dynamic and Cost Semantics

The dynamic semantics includes both an *evaluation* judgement for executing a program from scratch (Subsection 4.4.1) and a *change-propagation* judgement for replaying a computation under a (possibly) different store (Subsection 4.4.3); each produces a value in an updated store as well a *trace* and *cost*. The semantics of the Tgt language uses traces to capture the structure of the computation, and are sufficient to update a run under different inputs through a combination of evaluation and change-propagation. Adaptivity identifies when change-propagation must revert to evaluation to re-execute the portions of the computation that differ, while computation memoization (Subsection 4.4.2) switches from evaluation to change-propagation to reuse the unaffected portions of the computation.

**Traces.** A Tgt *trace* $T$ is a sequence of memo and TDT actions $A$, ending in a halt action. A *trace slice* $S$ is a trace segment; it may end in a halt action or a hole marker $\texttt{hole}^e$ that indicates the rest of the trace (corresponding to the run of $e$) was stolen for out-of-order reuse. Note that trace actions correspond to Tgt commands, and a trace is also a sliced trace with no holes.

$$
\begin{array}{rrcl}
\text{traces} & T & ::= & \texttt{halt}^v \mid A{\cdot}T \\
\text{trace slices} & S & ::= & H \mid A{\cdot}S \\
\text{slice final actions} & H & ::= & \texttt{halt}^v \mid \texttt{hole}^e \\
\text{actions} & A & ::= & \texttt{memo}^e \mid \texttt{mk}_{v_\text{k}}^{v_\text{mk}\uparrow\ell} \mid \texttt{op}_{v_\text{k}\bigcirc}^{\ell,v_\text{arg}\downarrow v_\text{res}} \\
\text{checkmarks} & \bigcirc & ::= & \checkmark \mid \text{✗}
\end{array}
$$

Since continuations capture the rest of the computation, a memo action $\texttt{memo}^e$ has no explicit return point, so it is an atomic action instead of delimiting the trace of its body—as done by Src memo actions.

A creation action $\texttt{mk}_{v_k}^{v_{mk}\uparrow\ell}$ records the argument, result, and continuation of a TDT creation command. A manipulation action $\texttt{op}_{v_k\bigcirc}^{\ell,v_{arg}\downarrow v_{res}}$ records the location accessed, the argument, result, and continuation of a TDT manipulation command; the action is additionally labeled by a *checkmark* $\bigcirc$ to indicate whether the action can be replayed during change-propagation (i.e., the result of the operation is consistent with the current store). The dynamic semantics maintains *consistency* of the reuse trace, i.e., the prefix trace of

$$\frac{}{v \Downarrow v} \qquad \frac{e_\mathrm{z} \Downarrow v}{\mathbf{caseN}\,\mathbf{zero}\,e_\mathrm{z}\,(x.e_\mathrm{s}) \Downarrow v} \qquad \frac{[v_\mathrm{n}/x]e_\mathrm{s} \Downarrow v}{\mathbf{caseN}\,(\mathbf{succ}\,v_\mathrm{n})\,e_\mathrm{z}\,(x.e_\mathrm{s}) \Downarrow v}$$

$$\frac{e_\mathrm{f} \Downarrow \mathbf{fun}\,f.x.e \qquad e_\mathrm{v} \Downarrow v_\mathrm{x} \qquad [v_\mathrm{x}/x][\mathbf{fun}\,f.x.e/f]e \Downarrow v}{e_\mathrm{f}\,e_\mathrm{x} \Downarrow v}$$

---

$$\frac{e \Downarrow \kappa \qquad \overline{S}; \sigma; \kappa \Downarrow_\mathrm{K} T'; \sigma'; v'; d'}{\overline{S}; \sigma; e \Downarrow_\mathrm{E} T'; \sigma'; v'; d'} \qquad \frac{|\overline{S}| = c}{\overline{S}; \sigma; \mathbf{halt}\,v \Downarrow_\mathrm{K} \mathtt{halt}^v; \sigma; v; \langle c, 1\rangle}$$

$$\frac{\overline{S}; \sigma; e \Downarrow_\mathrm{E} T'; \sigma'; v'; d'}{\overline{S}; \sigma; \mathbf{memo}\,e \Downarrow_\mathrm{K} \mathtt{memo}^e{\cdot}T'; \sigma'; v'; \langle 0, 1\rangle + d'}\ \textbf{memo/miss}$$

$$\frac{\sigma; \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S_\mathrm{e}; c \qquad \overline{S}'; S_\mathrm{e}; \sigma \curvearrowright T'; \sigma'; v'; d'}{\overline{S}; \sigma; \mathbf{memo}\,e \Downarrow_\mathrm{K} \mathtt{memo}^e{\cdot}T'; \sigma'; v'; \langle 1, 1\rangle + d'}\ \textbf{memo/hit}$$

$$\frac{\begin{array}{c}\ell \notin \mathrm{dom}\,\sigma \qquad v_\mathrm{mk}; \overline{S} \overset{\mathbf{mk}\ \ell}{\Mapsto} \mathcal{S}'; \overline{S}_\mathrm{mk} \\ \sigma_\mathrm{l} = \sigma[\ell \mapsto \mathcal{S}'] \qquad \overline{S}_\mathrm{mk}; \sigma_\mathrm{l}; v_\mathrm{k}\,\ell \Downarrow_\mathrm{E} T'; \sigma'; v'; d'\end{array}}{\overline{S}; \sigma; \mathbf{mk\_k}\,v_\mathrm{mk}\,v_\mathrm{k} \Downarrow_\mathrm{K} \mathtt{mk}_{v_\mathrm{k}}^{v_\mathrm{mk}\uparrow\ell}{\cdot}T'; \sigma'; v'; \langle 0, 1\rangle + d'}\ \textbf{mk}$$

$$\frac{\begin{array}{c}\ell \in \mathrm{dom}\,\sigma \qquad \sigma(\ell) = \mathcal{S} \qquad \mathcal{S}; v_\mathrm{arg}; \overline{S} \overset{\mathbf{op}\ \ell}{\Mapsto} \mathcal{S}'; v_\mathrm{res}; \overline{S}_\mathrm{op} \\ \sigma_\mathrm{l} = \sigma[\ell \mapsto \mathcal{S}'] \qquad \overline{S}_\mathrm{op}; \sigma_\mathrm{l}; v_\mathrm{k}\,v_\mathrm{res} \Downarrow_\mathrm{E} T'; \sigma'; v'; d'\end{array}}{\overline{S}; \sigma; \mathbf{op\_k}\,\ell\,v_\mathrm{arg}\,v_\mathrm{k} \Downarrow_\mathrm{K} \mathtt{op}_{v_\mathrm{k}\checkmark}^{\ell, v_\mathrm{arg}\downarrow v_\mathrm{res}}{\cdot}T'; \sigma'; v'; \langle 0, 1\rangle + d'}\ \textbf{op}$$

Figure 4.2: Reduction $e \Downarrow v$ (top) and evaluation $\overline{S}; \sigma; e \Downarrow_\mathrm{E} T'; \sigma'; v'; d'$ and $\overline{S}; \sigma; \kappa \Downarrow_\mathrm{K} T'; \sigma'; v'; d'$ (bottom).

actions with a valid checkmark ✓ are consistent with the store and thus replayable by change-propagation, and the earliest (if any) inconsistent manipulation action has an invalid checkmark ✗ and must be re-executed by change-propagation.

The metavariables $\dot{T}$ and $\dot{S}$ range over optional traces and trace slices. The metavariable $\overline{S}$ ranges over $\bullet$-terminated, comma-separated lists of trace slices. The metavariable $U$ ranges over non-empty lists of slices; concatenation extends to the first slice: $A{\cdot}(S, \overline{S}) = (A{\cdot}S, \overline{S})$, we omit the parentheses when they are clear from context.

$$\begin{array}{rrcl} \text{optional trace} & \dot{T} & ::= & \circ \mid T \\ \text{optional slice} & \dot{S} & ::= & \circ \mid S \\ \text{slice list} & \overline{S} & ::= & \bullet \mid S, \overline{S} \\ \text{non-empty slice list} & U & ::= & S, \overline{S} \end{array}$$

### 4.4.1 Evaluation

**Overview.** Figure 4.2 gives the large-step evaluation relation $\overline{S}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d'$ (resp. $\overline{S}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'$) reduces the expression $e$ (resp. the command $\kappa$) under the store $\sigma$, yielding the value $v'$ and the updated store $\sigma'$. Evaluation also takes a list of trace slices $\overline{S}$ for reuse, and produces an execution trace $T'$ for the current run and a pair of costs $d' = \langle c, c' \rangle$ for work $c$ discarded from the reuse trace slices and new work $c'$ performed for the current run. The auxiliary evaluation relation $e \Downarrow v'$ reduces an expression $e$ to a value $v'$; such evaluation is pure and independent of the store. Note that unlike the dynamic semantics of Src, where the computation trace and cost proceeded in lock-step, Tgt traces do not subsume the cost of the current run because the latter only accounts for *new* work that wasn't taken from the reuse trace.

The three evaluation relations model the execution of a self-adjusting program as the interleaving of pure computations and commands. As in Src, 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 and represent the program's changeable input.

The trace slices $\overline{S}$ represent segments of computation from a previous evaluation. A memo-match during evaluation can pick a trace slice and supply it to change-propagation to reuse the computation. Evaluation with an empty list corresponds to a from-scratch run without the possibility of reuse from a previous run. In the presence of a non-empty list of reuse slices, we can combine in-order reuse with any TDT and out-of-order reuse with single- and multi-write modrefs.

**In-Order Reuse with TDTs.** In-order computation memoization reuses work from the previous run in execution order; thus $\overline{S}$ is either empty because it is a from-scratch run or a singleton trace $T$ that corresponds to a tail of the previous run. A TDT is compatible with in-order memoization because it is straightforward to maintain the consistency of the tail trace $T$.

**Out-of-Order Reuse with Modrefs.** Out-of-order computation memoization can reuse work from the previous in any order; thus $\overline{S}$ can be several trace slices from the previous run. As presented here, TDTs are not compatible with out-of-order memoization because there isn't an immediate way to maintain the consistency of multiple trace slices. Out-of-order reuse can be combined with single- and multi-write modrefs, however, by explicitly checking which actions can be replayed instead of relying on a consistency invariant. In Chapter 8, we discuss how the semantics may be generalized to combine out-of-order reuse with arbitrary TDTs.

**Evaluation Rules.** The $\mathbf{halt}\ v$ command yields a computation's final value, with a cost of 1 for the current run and a cost of $|\overline{S}| = c$ for discarding the reuse trace slices $\overline{S}$. The cost of a trace slice $|S| = c$ is the number of actions (except holes, which don't represent previous work) in the trace:

$$\begin{array}{rcl} |\mathtt{hole}^e| & = & 0 \\ |\mathtt{halt}^v| & = & 1 \\ |A{\cdot}T| & = & |A| + |T| \end{array}$$

The cost of a list of trace slices $|\overline{S}| = c$ sums the cost of each trace slice:

$$\begin{array}{rcl} |\bullet| & = & 0 \\ |S, \overline{S}| & = & |S| + |\overline{S}| \end{array}$$

A memoized expression $\mathbf{memo}\ e$ in Tgt has no special behavior when evaluated from scratch (**memo/miss** rule): in the absence of a memo match, it evaluates the body $e$ and extends the trace with a memo action $\mathtt{memo}^e$ to identify the rest of the trace $T'$ as the evaluation of $e$, incurring a cost of 1 for the current run. Memoization enables the reuse of computations *across runs* during change-propagation. The **memo/hit** rule exploits the reuse trace from the previous evaluation and switches to change-propagation (Subsection 4.4.3) if the same expression was memoized and evaluated in the previous run. In self-adjusting computation, memoization enables a single reuse of a computation between runs, whereas classical memoization Michie [1968] permits sharing the result of a computation multiple times within a single run of a program.

The Tgt semantics of TDT commands are determined by the TDT's standalone state-transformation semantics. A $\mathbf{mk\_k}\ v_{mk}\ v_k$ creation command (**mk**) generates a TDT state $\mathcal{S}'$ of type $\tau\ \mathbf{tdt}$ with seed value $v_{mk}$ according to the state-transformation semantics, extends the store $\sigma$ with a fresh location $\ell$ bound to $\mathcal{S}'$, and delivers $\ell$ to the continuation $v_k$. Note that the choice of location $\ell$ is independent of the reuse trace slices $\overline{S}$. It is acceptable—and, indeed, often desirable—for the location $\ell$ to appear in a creation action $\mathtt{mk}_{v_k}^{v_{mk}\uparrow\ell}$ in the reuse trace; we say that such a location is (implicitly) *stolen* from the reuse trace. An $\mathbf{op\_k}\ \ell\ v_{arg}\ v_k$ manipulation command (**op**) fetches the TDT state $\mathcal{S}$ from the store $\sigma$ at $\ell$, performs the corresponding state-transformation, updates the store with $\ell$ bound to the new state $\mathcal{S}'$, and delivers the result $v_{res}$ to the continuation $v_k$. TDT manipulation (specifically, query) actions in a computation trace identify computations that must be re-executed by change-propagation if they become inconsistent with the store due to input changes. In each case, the trace is the singleton action corresponding to the primitive labeled by the relevant arguments and results, and the command incurs a cost of 1 for the current run. The $v_{mk}; \overline{S} \overset{\mathbf{mk}\,\ell}{\Mapsto} \mathcal{S}'; \overline{S}'$ and $\mathcal{S}; v_{arg}; \overline{S} \overset{\mathbf{op}\,\ell}{\Mapsto} \mathcal{S}'; v_{res}; \overline{S}'$ judgements may intuitively be read as the state-transformation judgements $v_{mk} \overset{\mathbf{mk}}{\mapsto} \mathcal{S}$ and $\mathcal{S}; v_{arg} \overset{\mathbf{op}}{\mapsto} \mathcal{S}'; v_{res}$ from the TDT signature. For in-order memoization with TDTs, the list trace slices is either a single trace or empty—e.g., $\overline{S} = \dot{T}$ and $\overline{S}' = \dot{T}'$—and the $\overset{mk}{\Mapsto}$ and $\overset{op}{\Mapsto}$ judgements additionally maintain the consistency of the trace (see below). For out-of-order memoization with modrefs, the $\overset{mk}{\Mapsto}$ and $\overset{op}{\Mapsto}$ judgements should only be read as the state-transformation judgements for modrefs leaving the trace slices unchanged—e.g. $\overline{S}' = \overline{S}$.

**Trace Reparation and Operation Invocation.** The *trace reparation* and *operation invocation* judgements (Figure 4.3) use the state-transformation rules to maintain trace consistency.

The trace reparation judgement $\mathcal{S}; \dot{T} \overset{\text{rep}\,\ell}{\Mapsto} \dot{T}'$ takes a TDT state $\mathcal{S}$ at location $\ell$ and an optional reuse trace $\dot{T}$ (with possible inconsistencies in actions that manipulate $\ell$) to produce the consistent optional trace $\dot{T}'$. Intuitively, trace reparation identifies the earliest inconsistent action that manipulates $\ell$ and marks it with an invalid checkmark.

A halt action isn't subject to any repair. Any action that does not manipulate location $\ell$ is preserved and the tail of the trace is recursively repaired (**rep/indep**). For any action that manipulates $\ell$, the state-transformation is simulated on the TDT state $\mathcal{S}$. If the state-transformation produces the same answer, the action receives a valid checkmark ✓ and the tail of the trace is recursively repaired with the simulated new TDT state $\mathcal{S}'$ (**rep/✓**). Otherwise the action receives an invalid checkmark ✗ and the resulting trace is consistent (**rep/✗**).

$$\frac{}{\mathcal{S};\circ \overset{\text{rep}\,\ell}{\Subset} \circ}\ \textbf{rep/none} \qquad \frac{}{\mathcal{S};\mathtt{halt}^v \overset{\text{rep}\,\ell}{\Subset} \mathtt{halt}^v}\ \textbf{rep/halt} \qquad \frac{A \neq \mathtt{op}^{\ell,\_\downarrow\_}_{\_\_} \qquad \mathcal{S};T \overset{\text{rep}\,\ell}{\Subset} T'}{\mathcal{S};A{\cdot}T \overset{\text{rep}\,\ell}{\Subset} A{\cdot}T'}\ \textbf{rep/indep}$$

$$\frac{\mathcal{S};v_{\text{arg}} \overset{\mathbf{op}}{\mapsto} \mathcal{S}';v_{\text{res}} \qquad \mathcal{S}';T \overset{\text{rep}\,\ell}{\Subset} T'}{\mathcal{S};\mathtt{op}^{\ell,v_{\text{arg}}\downarrow v_{\text{res}}}_{v_{\text{k}}\bigcirc}{\cdot}T \overset{\text{rep}\,\ell}{\Subset} \mathtt{op}^{\ell,v_{\text{arg}}\downarrow v_{\text{res}}}_{v_{\text{k}}\checkmark}{\cdot}T'}\ \textbf{rep/}\checkmark \qquad \frac{\mathcal{S};v_{\text{arg}} \overset{\mathbf{op}}{\mapsto} \mathcal{S}';v'_{\text{res}}}{\mathcal{S};\mathtt{op}^{\ell,v_{\text{arg}}\downarrow v_{\text{res}}}_{v_{\text{k}}\bigcirc}{\cdot}T \overset{\text{rep}\,\ell}{\Subset} \mathtt{op}^{\ell,v_{\text{arg}}\downarrow v_{\text{res}}}_{v_{\text{k}}\text{✗}}{\cdot}T}\ \textbf{rep/}\text{✗}$$

---

$$\frac{v_{\text{mk}} \overset{\mathbf{mk}}{\mapsto} \mathcal{S}' \qquad \mathcal{S}';\dot{T} \overset{\text{rep}\,\ell}{\Subset} \dot{T}'}{v_{\text{mk}};\dot{T} \overset{\mathbf{mk}\,\ell}{\Subset} \mathcal{S}';\dot{T}'}\ \textbf{mk/invoke} \qquad \frac{\mathcal{S};v_{\text{arg}} \overset{\mathbf{op}}{\mapsto} \mathcal{S}';v_{\text{res}} \qquad \mathcal{S}';\dot{T} \overset{\text{rep}\,\ell}{\Subset} \dot{T}'}{\mathcal{S};v_{\text{arg}};\dot{T} \overset{\mathbf{op}\,\ell}{\Subset} \mathcal{S}';v_{\text{res}};\dot{T}'}\ \textbf{op/invoke}$$

Figure 4.3: Trace reparation $\mathcal{S};\dot{T} \overset{\text{rep}\,\ell}{\Subset} \dot{T}'$ (top) and invocation $v_{\text{mk}};\dot{T} \overset{\mathbf{mk}\,\ell}{\Subset} \mathcal{S}';\dot{T}'$ and $\mathcal{S};v_{\text{arg}};\dot{T} \overset{\mathbf{op}\,\ell}{\Subset} \mathcal{S}';v_{\text{res}};\dot{T}'$ (bottom).

The invocation judgements $v_{\text{mk}};\dot{T} \overset{\mathbf{mk}\,\ell}{\Subset} \mathcal{S}';\dot{T}'$ and $\mathcal{S};v_{\text{arg}};\dot{T} \overset{\mathbf{op}\,\ell}{\Subset} \mathcal{S}';v_{\text{res}};\dot{T}'$ use the corresponding state-transformation judgements for creating and manipulating a TDT state. Furthermore, since invoking the operation may affect the consistency of actions in the reuse trace $\dot{T}$ (if any) that manipulate location $\ell$, the trace reparation judgement is used to maintain the consistency of the reuse trace (**mk/invoke** and **op/invoke**).

Hence, the **mk** and **op** evaluation rules use the invocation judgements to perform the state-transformation and preserve trace consistency; moreover the manipulation action is labeled by a valid checkmark because it is consistent with the rest of the execution trace.

### 4.4.2 Computation Memoization

**Overview.** Computation memoization acts as the interface between evaluation and change-propagation. When evaluation encounters a $\mathbf{memo}\ e$ expression, it can search the reuse traces for a previous run of $e$ and adapt it to the current store with change-propagation. More precisely, if memoization finds a trace $S_e$ that is preceded by the memoization action $\mathtt{memo}^e$, then it corresponds to a previous run of $e$ under a (possibly) different store. Hence, the **memo/hit** evaluation rule switches to change-propagating $T_e$ under the current store in order to correct any inconsistent TDT operations. This can be more efficient than fully evaluating $e$ because any work that isn't affected by the input changes can be reused.

$$\frac{}{\sigma; \mathtt{memo}^{e} \cdot T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T; 1}\ \textbf{hit} \qquad \frac{\sigma; T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; c}{\sigma; \mathtt{memo}^{e'} \cdot T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; 1 + c} \qquad \frac{\sigma; T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; c}{\sigma; \mathtt{mk}^{v_{\mathrm{mk}} \uparrow \ell}_{v_{\mathrm{k}}} \cdot T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; 1 + c}$$

$$\frac{\ell \notin \mathrm{dom}\, \sigma \qquad \sigma; T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; c}{\sigma; \mathtt{op}^{\ell, v_{\mathrm{arg}} \downarrow v_{\mathrm{res}}}_{v_{\mathrm{k}} \bigcirc} \cdot T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; 1 + c} \qquad \frac{\sigma(\ell) = \mathcal{S} \qquad \mathcal{S}; T \overset{\mathrm{rep}\, \ell}{\Rightarrow} T' \qquad \sigma; T'; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; c}{\sigma; \mathtt{op}^{\ell, v_{\mathrm{arg}} \downarrow v_{\mathrm{res}}}_{v_{\mathrm{k}} \bigcirc} \cdot T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; 1 + c}\ \textbf{op/rev}$$

Figure 4.4: In-order memoization with TDTs $\sigma; T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; c'$ .

In-order memoization reuses work from the previous run in execution order, whereas out-of-order memoization allows segments of computation to be reused out of order. We use the judgement $\sigma; \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S'_{\mathrm{e}}; c'$ for (in-order and out-of-order) memoization to search among reuse traces $\overline{S}$ for a trace $S'_e$ that corresponds to an evaluation of $e$, and leftover reuse traces $\overline{S}'$; the cost of the search is $c'$. For in-order reuse with TDTs, there is a single reuse trace $\overline{S} = T$ and the store $\sigma$ is used to produce a consistent trace $S'_e = T'_e$— i.e., a tail execution—, but there are no leftover reuse traces $\overline{S}' = \bullet$ because the prefix of $T$ preceding $S'_e$ is discarded and accounted for by the cost. For out-of-order reuse with modrefs, $\overline{S}'$ are any slices that remain in $\overline{S}$ besides $S'_e$, but ignores the store $\sigma$ and consistency of $S'_e$.

**In-Order Computation Memoization with TDTs.** The *in-order* memoization with TDTs judgement $\sigma; T; e \overset{\mathrm{m}}{\rightsquigarrow} \bullet; T_{\mathrm{e}}; c'$ (given in Figure 4.4) is a refinement of the memoization judgement that requires the list of reuse slices $\overline{S}$ to be a single reuse trace $T$, returns an empty list of leftover reuse slices and a trace $T_e$ for expression $e$. The store $\sigma$ is used to preserve the consistency of the trace by updating the checkmarks of trace actions according to which prefix actions are discarded.

This form of memoization searches the reuse trace $T$ for a suffix trace $T_e$ that follows a memoization action $\mathtt{memo}^e$. A matching memo action (**hit**) returns the tail of the trace for change-propagation.

Note that a memoization hit (evaluation rule **memo/hit** and memoization rule **hit**) requires the expression being evaluated to be identical ($\alpha$-equivalent) to the expression in the memo action. 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 **mk** evaluation rule: (re-)executing a TDT creation, using a location that appears in the

$$\frac{\_; S; e \overset{\mathrm{m}}{\rightsquigarrow} S'; S'_{\mathrm{e}}; c}{\_; A{\cdot}S; e \overset{\mathrm{m}}{\rightsquigarrow} A{\cdot}S'; S'_{\mathrm{e}}; c} \qquad \frac{}{\_; \mathtt{memo}^{e}{\cdot}S_{\mathrm{e}}; e \overset{\mathrm{m}}{\rightsquigarrow} \mathtt{hole}^{e}; S_{\mathrm{e}}; 1}\ \textbf{hit}$$

$$\frac{\_; \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S'_{\mathrm{e}}; c}{\_; S, \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} S, \overline{S}'; S'_{\mathrm{e}}; c} \qquad \frac{\_; S; e \overset{\mathrm{m}}{\rightsquigarrow} S'; S'_{\mathrm{e}}; c}{\_; S, \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} S', \overline{S}; S'_{\mathrm{e}}; c}$$

Figure 4.5: Out-of-order memoization with modrefs $\_; S; e \overset{\mathrm{m}}{\rightsquigarrow} S'; S'_{\mathrm{e}}; c'$ (top) and $\_; \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S'_{\mathrm{e}}; c'$ (bottom).

reuse trace, may allow a subsequent memoization action to match in the reuse trace.

Memo and TDT actions can be discarded by proceeding to match the tail of the trace. Discarding a memo does not affect the consistency of the trace because it does not touch the store. Discarding a creation action of location $\ell$ or a manipulation action on a location $\ell$ that is not in the store does not affect the consistency of the trace because the location ceases to be in the store; if the location is later re-allocated during evaluation (**mk**), then the reuse trace will be made consistent by the invocation judgement. A manipulation action $\mathtt{op}_{v_k \bigcirc}^{\ell, v_{arg} \downarrow v_{res}}$ on a location $\ell$ that is in the store (**op/rev**) must be explicitly *revoked* because it will no longer be performed, thus the tail of the trace must be repaired relative to the current state $\mathcal{S} = \sigma(\ell)$.

**Out-of-Order Computation Memoization with Modrefs.** Out-of-order memoization $\_; S; e \overset{\mathrm{m}}{\rightsquigarrow} S'; S'_{\mathrm{e}}; c'$ (given in Figure 4.5) splits the reuse trace $S$ into a suffix trace slice $S'_e$ that corresponds to a (partial) previous run of $e$ under a (possibly) different store, and a prefix trace $S'$ of the work preceding $S'_e$ with an explicit $\mathtt{hole}^e$ end marker to indicate the stolen tail. The memoization judgment extends to slice lists $\_; \overline{S}; e \overset{\mathrm{m}}{\rightsquigarrow} \overline{S}'; S'_{\mathrm{e}}; c'$ by memo-matching one trace from the list. Note that while the expression $e$ may have free locations, out-of-order memoization is independent of the store.

### 4.4.3 Change-Propagation

**Overview.** The change-propagation relation $\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'$ (given in Figure 4.6) replays the execution trace $S$ under the store $\sigma$, yielding the value $v'$ and the updated store $\sigma'$, with an updated execution trace $T'$ and a pair of costs $d' = \langle c, c' \rangle$ for work $c$ discarded

$$\frac{|\overline{S}| = c}{\overline{S}; \mathtt{halt}^v; \sigma \curvearrowright \mathtt{halt}^v; \sigma; v; \langle c, 0\rangle}\ \textbf{halt/reuse}$$

$$\frac{\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'}{\overline{S}; \mathtt{memo}^e \cdot S; \sigma \curvearrowright \mathtt{memo}^e \cdot T'; \sigma'; v'; d'}\ \textbf{memo/reuse}$$

$$\frac{\begin{array}{c}\ell \notin \operatorname{dom} \sigma \qquad v_{\mathrm{mk}} \overset{\mathbf{mk}}{\mapsto} \mathcal{S}' \\ \sigma_1 = \sigma[\ell \mapsto \mathcal{S}'] \qquad \overline{S}; S; \sigma_1 \curvearrowright T'; \sigma'; v'; d'\end{array}}{\overline{S}; \mathtt{mk}_{v_{\mathrm{k}}}^{v_{\mathrm{mk}} \uparrow \ell} \cdot S; \sigma \curvearrowright \mathtt{mk}_{v_{\mathrm{k}}}^{v_{\mathrm{mk}} \uparrow \ell} \cdot T'; \sigma'; v'; d'}\ \textbf{mk/reuse}$$

$$\frac{\begin{array}{c}\sigma(\ell) = \mathcal{S} \qquad \mathcal{S}; v_{\mathrm{arg}} \overset{\mathbf{op}}{\mapsto} \mathcal{S}'; v_{\mathrm{res}} \\ \sigma_1 = \sigma[\ell \mapsto \mathcal{S}'] \qquad \overline{S}; S; \sigma_1 \curvearrowright T'; \sigma'; v'; d'\end{array}}{\overline{S}; \mathtt{op}_{v_{\mathrm{k}}}^{\ell, v_{\mathrm{arg}} \downarrow v_{\mathrm{res}}} \cdot S; \sigma \curvearrowright \mathtt{op}_{v_{\mathrm{k}}}^{\ell, v_{\mathrm{arg}} \downarrow v_{\mathrm{res}}} \cdot T'; \sigma'; v'; d'}\ \textbf{op/reuse}$$

$$\frac{\lceil S \rceil = \kappa \qquad S, \overline{S}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d'}{\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'}\ \textbf{change}$$

Figure 4.6: Change-propagation $\overline{S}; S; \sigma \curvearrowright T'; \sigma'; v'; d'$.

from $S$ and $\overline{S}$, and new work $c'$ performed for $T'$. The additional reuse traces $\overline{S}$ are other computation from the previous run; it may be reused if change-propagation returns to evaluation and memo-matches on it.

**Change-Propagation Rules.** A halt action can be replayed to obtain the (unchanged) computation result, incurring the cost of discarding the leftover reuse traces.

An action can be replayed without cost if the action is consistent with the current store (**reuse** rules), the tail of the trace can be recursively change-propagated and then extended with the same action. Thus, replaying a memoization action or a consistent TDT action recursively change-propagates the tail of the trace, and the updated computation trace is extended with the appropriate action.

A creation operation $\mathtt{mk}_{v_k}^{v_{mk}\uparrow\ell}$ is consistent with the current store if $\ell \notin \operatorname{dom}\sigma$ and can thus be replayed (**mk/reuse**) by regenerating the TDT state $\mathcal{S}'$ with seed $v_{mk}$, extending the store with $\ell$ bound to $\mathcal{S}'$, and recursively change-propagating the tail of the trace. A manipulation operation $\mathtt{op}_{v_k\bigcirc}^{\ell,v_{arg}\downarrow v_{res}}$ is replayed (**op/reuse**) by re-executing the state-transformation to yield the same result $v_{res}$, updating the store with $\ell$ bound to the new state $\mathcal{S}'$, and recursively change-propagating the tail of the trace. For in-order memoization with TDTs, the consistency invariant of the trace guarantees that if the action has a valid checkmark ✓ then replaying the action will produce the same result $v_{res}$. For out-of-order memoization with modrefs, the consistency of the trace slices aren't preserved and the checkmark should be ignored.

Change-propagation falls back to evaluation either nondeterministically or because the head action is inconsistent with the current store and thus not replayable. A TDT creation is inconsistent with the current store if the location is already in the store, which thus forces the allocation of a fresh location to be initialized by a new TDT instance. A TDT manipulation is inconsistent with the current store if the state-transformation produces a different result as labeled by an invalid checkmark ✗, which thus forces the the operation to be re-executed.

Since actions capture their continuation, a sliced trace $S$ can be *reified* back into a command $\lceil S \rceil = \kappa$ that represents the rest of the computation:

$$\begin{aligned}
\lceil \mathtt{halt}^{v} \rceil &= \mathbf{halt}\, v \\
\lceil \mathtt{hole}^{e} \rceil &= \mathbf{memo}\, e \\
\lceil \mathtt{memo}^{e} \cdot S \rceil &= \mathbf{memo}\, e \\
\lceil \mathtt{mk}_{v_\mathrm{k}}^{v_\mathrm{mk}\uparrow\ell} \cdot S \rceil &= \mathbf{mk_k}\, v_\mathrm{mk}\, v_\mathrm{k} \\
\lceil \mathtt{op}_{v_\mathrm{k}\bigcirc}^{\ell,v_\mathrm{arg}\downarrow v_\mathrm{res}} \cdot S \rceil &= \mathbf{op_k}\, \ell\, v_\mathrm{arg}\, v_\mathrm{k}
\end{aligned}$$

Thus, change-propagation can reify and re-evaluate an inconsistent trace $S$ (**change** rule), while keeping the trace $S$ for possible reuse later. Note that the reified $\mathbf{mk}$ (resp. $\mathbf{op}$) command forgets the (stale) location (resp. result value).

The **change** rule does *not*, however, require the head action to be inconsistent. The rules are intentionally nondeterministic to avoid committing to particular allocation and memoization policies. Thus the **mk** evaluation rule may allocate locations in the reuse trace and the memoization judgement may match computations in the reuse trace 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 6.5 provides mechanisms that restrict when locations may be stolen and when memoization may match.

Note that change-propagation copies the prefix of the computation trace up to the first action (i.e., TDT creation or manipulation) 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.

### 4.4.4 Meta-Theory

We can now sketch the use of change-propagation by a host program that (re-)evaluates a self-adjusting computation. The following applies to Tgt with either in-order memoization with TDTs or out-of-order memoization with modrefs. Suppose we have a Tgt program $e$ such that $\Sigma; \cdot \vdash e : \mathbf{res}$ and an initial store $\sigma_1$ such that $\vdash \sigma_1 : \Sigma \uplus \Sigma_1$. Thus, we can initially evaluate $e$ under the store $\sigma_1$ without any reuse traces, yielding the (initial) result $v_1'$ and a computation trace $T_1'$: $\bullet; \sigma_1; e \Downarrow_{\mathrm{E}} T_1'; \sigma_1'; v_1'; \langle 0, c_1 \rangle$. Now, suppose we have a modified store $\sigma_2$ such that $\vdash \sigma_2 : \Sigma \uplus \Sigma_2$. We are interested in the result $v_2'$ yielded by (re-)evaluating $e$ under $\sigma_2$. To obtain $v_2'$, we may change-propagate the trace $T_1'$ under the store $\sigma_2$: $\bullet; T_1'; \sigma_2 \curvearrowright T_2'; \sigma_2'; v_2'; \langle c_1', c_2' \rangle$. Change-propagation may reuse some work from the previous run, therefore $c_1'$ only accounts for work discarded from $T_1'$ ($c_1' \leq c_1$) and $c_2'$ only accounts for new work in $T_2'$ ($c_2' \leq c_2$).

The consistency of change-propagation asserts that the result $v_2'$, store $\sigma_2'$, and trace $T_2'$ obtained via the change-propagation relation are identical to those obtained from a from-scratch evaluation (i.e., without any reuse traces): $\bullet; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \langle 0, c_2 \rangle$. This correspondence is intuitively captured in Figure 4.7. Hence, change-propagation suffices to determine the output of a program on changed inputs. We prove this consistency prop-

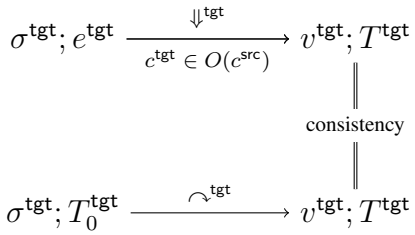



Figure 4.7: The consistency of change-propagation in the target.

erty for Tgt by giving a simple structural proof.

According to the form of computation memoization, change-propagation will reuse the previous run $T_1'$ in different ways. In-order reuse will only keep a suffix of $T_1'$ throughout change-propagation and evaluation. Out-of-order reuse may split $T_1'$ into slices corresponding to segments of computation that are reused and other fragments that are leftover.

The following theorem formalizes the consistency of change-propagation for Tgt with either in-order memoization with TDTs or out-of-order memoization with modrefs. We introduce the auxiliary judgements $S$ wfwrt $e$ to mean $S$ results from slicing a from-scratch execution of $e$ ($\bullet; \_; e \Downarrow_{\mathrm{E}} T'; \_; \_; \_$ and $\sigma; T'; e \overset{\mathrm{m}}{\rightsquigarrow} S; S_{\mathrm{e}}'; \_$), and $S$ wf to mean $S$ wfwrt $e$ for some $e$. Since the reuse trace may be sliced by out-of-order memoization, the statement requires the well-formedness of multiple trace slices $\overline{S}$. Consistency follows as a corollary by instantiating $\overline{S}$ as the empty list and $S_1'$ as $T_1'$. The cost of change-propagation is related to trace distance in Section 4.5.

**Lemma 29 (Trace Splitting)**
*If* $S_1$ wfwrt $e_1$ *and* $\_; S_1; e_2 \overset{\mathrm{m}}{\rightsquigarrow} S_1'; S_2'; \_$
*then* $S_1'$ wfwrt $e_1$ *and* $S_2'$ wfwrt $e_2$.

**Proof:** By induction on the second derivation (memo-matching).

Case **hit**. By **memo/miss**.

The remaining cases follow by the i.h..

■

**Theorem 30 (Consistency of change-propagation)**
*If* $\overline{S}$ wf, $S_1'$ wfwrt $e$, *and* $\overline{S}; S_1'; \sigma_2 \curvearrowright T_2'; \sigma_2'; v_2'; \_$,
*then* $\bullet; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$.

*If* $\overline{S\ \mathsf{wf}}$ *and* $\overline{S}; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$,
*then* $\bullet; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$.
*If* $\overline{S\ \mathsf{wf}}$, $S_1'\ \mathsf{wfwrt}\ \kappa$, *and* $\overline{S}; S_1'; \sigma_2 \curvearrowright T_2'; \sigma_2'; v_2'; \_$,
*then* $\bullet; \sigma_2; \kappa \Downarrow_{\mathrm{K}} T_2'; \sigma_2'; v_2'; \_$.
*If* $\overline{S\ \mathsf{wf}}$ *and* $\overline{S}; \sigma_2; \kappa \Downarrow_{\mathrm{K}} T_2'; \sigma_2'; v_2'; \_$,
*then* $\bullet; \sigma_2; \kappa \Downarrow_{\mathrm{K}} T_2'; \sigma_2'; v_2'; \_$.

**Proof:** By simultaneous induction on the last derivation (evaluation or change-propagation) of each statement.

Case $\Downarrow_{\mathrm{E}}\curvearrowright$.

Subcase $\curvearrowright$ **halt** ($\curvearrowright$ **memo**, $\curvearrowright$ **put**, $\curvearrowright$ **get**, $\curvearrowright$ **set** are analogous). By i.h.(3), and rule $\Downarrow_{\mathrm{K}}\Downarrow_{\mathrm{E}}$.

Subcase $\curvearrowright$ **change**. By i.h.(4).

Case $\Downarrow_{\mathrm{E}}$. By i.h.(4).

Case $\Downarrow_{\mathrm{K}}\curvearrowright$.

Subcase $\curvearrowright$ **halt**. By rule $\Downarrow_{\mathrm{K}}$ **halt**.

Subcase $\curvearrowright$ **memo** ($\curvearrowright$ **put**, $\curvearrowright$ **get**, $\curvearrowright$ **set** are analogous). By inversion, i.h.(1), and rule $\Downarrow_{\mathrm{K}}$ **memo/miss**.

Subcase $\curvearrowright$ **change**. By i.h.(4).

Case $\Downarrow_{\mathrm{K}}$.

Subcase $\Downarrow_{\mathrm{K}}$ **halt**. By rule $\Downarrow_{\mathrm{K}}$ **halt**.

Subcases $\Downarrow_{\mathrm{K}}$ **memo/miss** ($\Downarrow_{\mathrm{K}}$ **put**, $\Downarrow_{\mathrm{K}}$ **get**, $\Downarrow_{\mathrm{K}}$ **set** are analogous). By i.h.(2), and rule $\Downarrow_{\mathrm{K}}$ **memo/miss**.

Subcase $\Downarrow_{\mathrm{K}}$ **memo/hit**. By Lemma 29, i.h.(1), and rule $\Downarrow_{\mathrm{K}}$ **memo/miss**.

■

$$\frac{|H_1| = c_1 \qquad |H_2| = c_2}{H_1, \bullet \boxminus H_2, \bullet = \langle c_1, c_2 \rangle}$$

$$\frac{|H_1| = c_1 \qquad S_1, \overline{S}_1 \boxminus U_2 = d}{H_1, S_1, \overline{S}_1 \boxminus U_2 = \langle c_1, 0 \rangle + d} \textbf{ search/h/L} \qquad \frac{|H_2| = c_2 \qquad U_1 \boxminus S_2, \overline{S}_2 = d}{U_1 \boxminus H_2, S_2, \overline{S}_2 = \langle 0, c_2 \rangle + d} \textbf{ search/h/R}$$

$$\frac{S_1, \overline{S}_1 \boxminus U_2 = d}{A{\cdot}S_1, \overline{S}_1 \boxminus U_2 = \langle 1, 0 \rangle + d} \textbf{ search/a/L} \qquad \frac{U_1 \boxminus S_2, \overline{S}_2 = d}{U_1 \boxminus A{\cdot}S_2, \overline{S}_2 = \langle 0, 1 \rangle + d} \textbf{ search/a/R}$$

$$\frac{S_1, \overline{S}_1 \ominus S_2, \overline{S}_2 = d}{\texttt{memo}^e{\cdot}S_1, \overline{S}_1 \boxminus \texttt{memo}^e{\cdot}S_2, \overline{S}_2 = \langle 1, 1 \rangle + d} \textbf{ search/synch}$$

---

$$\frac{}{\texttt{halt}^v, \bullet \ominus \texttt{halt}^v, \bullet = \langle 0, 0 \rangle} \textbf{ synch/h} \qquad \frac{S_1, \overline{S}_1 \ominus S_2, \overline{S}_2 = d}{A{\cdot}S_1, \overline{S}_1 \ominus A{\cdot}S_2, \overline{S}_2 = d} \textbf{ synch/a}$$

$$\frac{U_1 \boxminus U_2 = d}{U_1 \ominus U_2 = d} \textbf{ synch/search}$$

Figure 4.8: Tgt local search distance $U_1 \boxminus U_2 = d$ and synchronization distance $U_1 \ominus U_2 = d$.

## 4.5 Trace Distance

Reasoning about the efficiency of by change-propagation is difficult because the dynamic and cost semantics include too many details. In particular, stores permeate the evaluation and change-propagation rules, so input changes appear throughout derivations. In this section, we introduce a theory of trace distance and show that the cost of change-propagation in the presence of modrefs is bounded by the distance between the input trace and updated trace. In Chapter 5, we show that Tgt distance is they are asymptotically the same as high-level Src distance. We present a theory of *local* trace distance (Subsection 4.5.1) that captures the cost of change-propagation with in-order memoization as an edit distance. We also present a theory of *global* trace distance (Subsection 4.5.2) that captures the cost of change-propagation with out-of-order memoization by decomposing traces into slices and comparing segments of computation pairwise.

### 4.5.1 Local Trace Distance

Since global distance accounts for computation reordering, the *local search distance* $U_1 \boxminus U_2 = d$ accounts for differences between traces in order until it finds matching memoization actions. At that point, it can use the *local synchronization distance* $U_1 \ominus U_2 = d$ to account for reuse between traces until they differ, at which point it must return to search distance. The distance $d = \langle c_1, c_2 \rangle$ quantifies the cost $c_1$ of work in $U_1$ that isn't shared with $U_2$ and the cost $c_2$ of work in $U_2$ that isn't shared with $U_1$.

**Search Distance.** The search distance (given in Figure 4.8) between halt or hole actions is the length of each action irrespective of the payload value or expression. Two identical memo actions incur a cost of $1$ each, but afford the possibility of switching from search to synchronization mode. Skipping an action incurs a cost of $1$ for the corresponding trace and requires distance to remain in search mode (**h/*** and **a/*** rules).

Two *identical* memo actions incur a cost of $1$ each and enable switching from search to synchronization mode because the tail traces belong to the evaluation of the same expression—under possibly different stores. Note that the **a/*** rules allow memo actions to remain in search mode; identical memo actions are never forced to synchronize.

**Synchronization Distance.** Although a synchronization distance is defined between any two traces, it is only meant to be used on traces generated by the evaluation of the same expression under (possibly) different stores. The synchronization distance between halt

actions is $\langle 0, 0\rangle$ (**synch/h**) and requires that both actions return the same value. Identical actions match without cost and allow distance to continue synchronizing the tail (**synch/a**). Synchronization may return to search mode (**synch/search**), either nondeterministically or because actions don't match. Just as Src distance, Tgt distance judgements are quasi-symmetric and induce a ternary relation due to the nondeterminism of memo matching.

### 4.5.2 Global Trace Distance

The *global (search) distance* between two traces decomposes each trace into a list of slices, pairs slices between the two runs, and sums the pairwise local search distance of slices.

**Trace Decomposition.** A Tgt trace slice $S$ can be decomposed into a non-empty list of slices $U'$ with the $S \gg U'$ judgement by (non-deterministically) replacing memo actions with holes.

$$\frac{}{H \gg H, \bullet} \qquad \frac{S \gg S', \overline{S}'}{A{\cdot}S \gg A{\cdot}S', \overline{S}'} \qquad \frac{S \gg S', \overline{S}'}{\mathtt{memo}^e{\cdot}S \gg \mathtt{hole}^e, \mathtt{memo}^e{\cdot}S', \overline{S}'}$$

The judgement extends to decomposing lists of slices $U \gg U'$ by appending the decomposition of each slice in the list. The judgement $U \overset{\mathsf{perm}}{\rightsquigarrow} U'$ means $U'$ is a permutation of $U$.

The global distance $U_1 \boxminus^{\gg} U_2 = d$ between two lists of slices $U_1$ and $U_2$ is formally obtained by slicing each $U_i$ into $U_i'$, permuting it into $U_i''$, and finally taking the local search distance of each (non-empty) list.

$$\frac{U_1 \gg U_1' \qquad U_1' \overset{\mathsf{perm}}{\rightsquigarrow} U_1'' \qquad U_2 \gg U_2' \qquad U_2' \overset{\mathsf{perm}}{\rightsquigarrow} U_2'' \qquad U_1'' \boxminus U_2'' = d}{U_1 \boxminus^{\gg} U_2 = d}$$

The global distance between two traces $T_1$ and $T_2$ is thus $T_1, \bullet \boxminus^{\gg} T_2, \bullet = d$.

### 4.5.3 Meta-Theory

**Trace Distance and Dynamic Semantics.** In light of the dynamic semantics, trace distance can be given an asymmetrical operational interpretation: distance is the amount of work that must be discarded from one run and executed to produce the other run. More precisely, a distance $\langle c_1, c_2\rangle$ between traces $T_1$ and $T_2$ intuitively means there is cost $c_1$ for discarding unusable work from the reuse trace $T_1$ and cost $c_2$ for performing new work

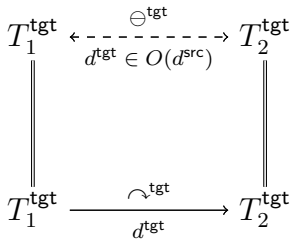



Figure 4.9: The correspondence between distance and the time for change-propagation in the target.

for $T_2$, but any common work that can be reused is free. Intuitively, the asymmetry arises from the fact that discarding work, while not free, is cheaper than performing work.

Search distance has an operational analogue realized by evaluation, while synchronization distance is realized by change-propagation. The following theorems relate distance to the dynamic semantics: the distance between a program's trace $T$ and some traces $\overline{S}$ coincides with the cost of evaluating the program with reuse traces $\overline{S}$. This correspondence is intuitively captured in Figure 4.9.

First, we introduce an auxiliary *asymmetric* global search $\overline{S} \boxminus^{\mathsf{perm}} T = d$ and synchronization $\overline{S} \ominus^{\mathsf{perm}} T = d$ distance (in the latter case $\overline{S}$ must be non-empty) that slices both sides but only permutes the left-hand side before comparing the two with local distance:

$$\frac{\langle S_{\mathrm{i}} \gg U'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \qquad T \gg U' \qquad \langle U'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \overset{\mathsf{perm}}{\rightsquigarrow} U'' \qquad U'' \boxminus U' = d}{\langle S_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \boxminus^{\mathsf{perm}} T = d}$$

$$\frac{\begin{array}{c}\langle S_{\mathrm{i}} \gg U'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \qquad U'_1 = S'_{11}, \langle S'_{1\mathrm{j}} \rangle_{\mathrm{j} \in 1..n_1} \qquad \langle S'_{1\mathrm{j}} \rangle_{\mathrm{j} \in 1..n_1}, \langle U'_{\mathrm{i}} \rangle_{\mathrm{i} \in 2..n} \overset{\mathsf{perm}}{\rightsquigarrow} U'' \\ T \gg U' \qquad S'_{11}, U'' \ominus U' = d\end{array}}{\langle S_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \ominus^{\mathsf{perm}} T = d}$$

**Lemma 31 (Dynamic semantics coincides with auxiliary local distance)**
*If* $\langle S'_{\mathrm{i}}\ \mathsf{wfwrt}\ e_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n}$, *and* $\bullet; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; \_$,
*then* $\langle S'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \boxminus^{\mathsf{perm}} T' = d$ *iff* $\langle S'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d$.
*If* $\langle S'_{\mathrm{i}}\ \mathsf{wfwrt}\ e_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n}$ (*where* $e_1 = e$), *and* $\bullet; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; \_$,
*then* $\langle S'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \ominus^{\mathsf{perm}} T' = d$ *iff* $\langle S'_{\mathrm{i}} \rangle_{\mathrm{i} \in 2..n}; S'_1; \sigma \curvearrowright T'; \sigma'; v'; d$ ($i > 1$).
*If* $\langle S'_{\mathrm{i}}\ \mathsf{wfwrt}\ e_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n}$, *and* $\bullet; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; \_$,
*then* $\langle S'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \boxminus^{\mathsf{perm}} T' = d$ *iff* $\overline{S}'_{\mathrm{i}}; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; d$.
*If* $\langle S'_{\mathrm{i}}\ \mathsf{wfwrt}\ e_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n}$ (*where* $e_1 = \kappa$), *and* $\bullet; \sigma; \kappa \Downarrow_{\mathrm{K}} T'; \sigma'; v'; \_$,
*then* $\langle S'_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \ominus^{\mathsf{perm}} T' = d$ *iff* $\langle S'_{\mathrm{i}} \rangle_{\mathrm{i} \in 2..n}; S'_1; \sigma \curvearrowright T'; \sigma'; v'; d$ ($i > 1$).

**Proof:** By simultaneous induction on the second derivation of each statement.

Case $\Downarrow_{\mathrm{E}}\Downarrow_{\mathrm{E}}$.

Subcase $\Downarrow_{\mathrm{K}}\Downarrow_{\mathrm{E}}$. By i.h.(3).

Case $\Downarrow_{\mathrm{E}}\curvearrowright$.

Subcase $\Downarrow_{\mathrm{K}}\Downarrow_{\mathrm{E}}$. By i.h.(3).

Case $\Downarrow_{\mathrm{K}}\Downarrow_{\mathrm{K}}$.

Subcase $\Downarrow_{\mathrm{K}}$ **halt**.
($\Rightarrow$) By rule $\Downarrow_{\mathrm{K}}$ **halt**.
($\Leftarrow$) $\langle S'_{\mathrm{i}}\rangle_{\mathrm{i}\in 1..n} \boxminus^{\mathsf{perm}} \mathtt{halt}^{v} = \langle c, 1\rangle$.

Subcase $\Downarrow_{\mathrm{K}}$ **memo/miss**.
($\Rightarrow$)

Subsubcase **search/memo/R**.By i.h.(1) and rule $\Downarrow_{\mathrm{K}}$ **memo/miss**.

Subsubcase **synch/search**.By i.h.(2) and rule $\Downarrow_{\mathrm{K}}$ **memo/hit**.

($\Leftarrow$)

Subsubcase $\Downarrow_{\mathrm{K}}$ **memo/miss**. By i.h.(1) and rule **search/memo/R**.

Subsubcase $\Downarrow_{\mathrm{K}}$ **memo/hit**. By i.h.(2), Lemma 29, and rule **synch/search**.

Subcase $\Downarrow_{\mathrm{K}}$ **put** ($\Downarrow_{\mathrm{K}}$ **get**, $\Downarrow_{\mathrm{K}}$ **set** are analogous).
($\Rightarrow$) By i.h.(1) and rule $\Downarrow_{\mathrm{K}}$ **put**.
($\Leftarrow$) By i.h.(1) and rule **search/a/R**.

Case $\Downarrow_{\mathrm{K}}\curvearrowright$.

Subcase $\Downarrow_{\mathrm{K}}$ **halt**.
($\Rightarrow$)

Subsubcase **synch/h**.By rule $\curvearrowright$ **halt**.

Subsubcase **search/synch**.By i.h.(3) and rule $\curvearrowright$ **change**.

($\Leftarrow$)

Subsubcase $\curvearrowright$ **halt**. By rule **synch/h**.

Subsubcase $\curvearrowright$ **change**. By i.h.(3) and rule **search/synch**.

Subcase $\Downarrow_{\mathrm{K}}$ **memo/miss**.
($\Rightarrow$)

Subsubcase **synch/a**.By i.h.(4) and rule $\curvearrowright$ **memo**.

Subsubcase **search/synch**.By i.h.(3) and rule $\curvearrowright$ **change**.

($\Leftarrow$)

Subsubcase $\curvearrowright$ **memo**. By i.h.(4) and rule **synch/a**.

Subsubcase $\curvearrowright$ **change**. By i.h.(3) and rule **search/synch**.

Subcase $\Downarrow_{\mathrm{K}}$ **put** ($\Downarrow_{\mathrm{K}}$ **get**, $\Downarrow_{\mathrm{K}}$ **set** are analogous).

($\Rightarrow$)

Subsubcase **synch/a**.By i.h.(4) and rule $\curvearrowright$ **put**.

Subsubcase **search/synch**.By i.h.(3) and rule $\curvearrowright$ **change**.

($\Leftarrow$)

Subsubcase $\curvearrowright$ **put**. By i.h.(4) and rule **synch/a**.

Subsubcase $\curvearrowright$ **change**. By i.h.(3) and rule **search/synch**.

■

**Theorem 32 (Dynamic semantics coincides with local distance)**
*If* $\bullet; \sigma_1; e_1 \Downarrow_{\mathrm{E}} T_1'; \sigma_1'; v_1'; \_$ *and* $\bullet; \sigma_2; e_2 \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$,
*then* $T_1' \boxminus T_2' = d$ *iff* $T_1'; \sigma_2; e_2 \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; d$.
*If* $\bullet; \sigma_1; e \Downarrow_{\mathrm{E}} T_1'; \sigma_1'; v_1'; \_$ *and* $\bullet; \sigma_2; e \Downarrow_{\mathrm{E}} T_2'; \sigma_2'; v_2'; \_$,
*then* $T_1' \ominus T_2' = d$ *iff* $\bullet; T_1'; \sigma_2 \curvearrowright T_2'; \sigma_2'; v_2'; d$.

**Proof:** By Lemma 31, choosing $T_1'$ and $T_2'$ to be sliced into themselves. ■

**Theorem 33 (Dynamic semantics coincides with global distance)**
*If* $\overline{S}$ $\mathsf{wf}$, *and* $\bullet; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; \_$,
*then* $\overline{S} \boxminus^{\gg} T' = d$ *iff* $\overline{S}; \sigma; e \Downarrow_{\mathrm{E}} T'; \sigma'; v'; d$.

**Proof:** By Lemma 31, noting that permuting both sides in global distance is equivalent to permuting only the left-hand side in asymmetric global distance. ■

# Chapter 5

# Translation

This chapter is based on work on a translation from a direct style Src language to a CPS-based Tgt language for self-adjusting computation [Ley-Wild et al., 2008b] and the preservation of cost semantics and trace distance for in-order reuse [Ley-Wild et al., 2009].

## 5.1 Overview

In this chapter, we present the *adaptive continuation-passing style* (ACPS) translation from $\mathsf{Src}(T)$ to $\mathsf{Tgt}(T, \mathsf{Pure})$. The translation uses Src-level TDTs and memoizing functions to produce Tgt-level self-adjusting programs with equivalent static semantics, and asymptotically equivalent dynamic and cost semantics and distance. Src-level direct style TDTs are translated to Tgt-level TDTs with a continuation-passing discipline. Src-level memoizing functions are translated to continuation-passing Tgt-level functions with explicit memoization at the function call and return points. Furthermore, the continuation is threaded through the store using Pure single-write modrefs to enable a function to memo-match on the argument even if the continuation—i.e., calling context—is different.

In Section 5.2, we give the ACPS program translation and show that it preserves the static semantics and dynamic and cost semantics of Src programs with TDTs. In Section 5.3, we extend the translation to trace slices and show that it preserves the global and local trace distance of Src programs with single- and multi-write modrefs.

## 5.2 Program Translation

The adaptive primitives of Src programs are used to guide an *adaptive continuation-passing style* (ACPS) transformation into equivalent Tgt programs (given in Figure 5.1). The ACPS transformation (given in Figure 5.1) uses the continuation to identify the scope of a TDT operation, so change-propagating an inconsistent TDT action will re-execute the tail of the trace. The translation is a selective CPS transformation [Danvy and Hatcliff, 1993a,b, Nielsen, 2001, Kim and Yi, 2001] that CPS-converts adaptive code—i.e., code that typechecks at mode $\$$—but keeps non-adaptive code in direct style—i.e., code that typechecks at mode $\flat$. Thus only adaptive functions and their bodies take an explicit continuation argument. The Src TDT operations are converted into equivalent Tgt TDT operations with explicit continuations. The translation of memoizing adaptive functions inserts explicit **memo** commands at the function call and return points and threads the continuation through the store, which are crucial for efficient change-propagation.

The type translation $[\![\tau^{\mathsf{src}}]\!] = \tau^{\mathsf{tgt}}$ converts the adaptive function type to take a continuation argument and is the straightforward structural translation for other types—e.g., it recursively translates the normal function type without introducing continuations. There are two term translations: $[\![e^{\mathsf{src}}]\!]^{\flat} = e^{\mathsf{tgt}}$ translates values and expressions appearing as the body of normal Src functions, and $[\![e^{\mathsf{src}}]\!]^{\$}\ v_k^{\mathsf{tgt}} = e^{\mathsf{tgt}}$ translates expressions appearing as the body of adaptive Src functions using the Tgt value $v_k^{\mathsf{tgt}}$ as the explicit continuation term. The metavariables $y$ and $k$ are used to distinguish identifiers introduced by the translation.

The $[\![e]\!]^{\flat}$ translation recursively translates sub-expressions and appropriately translates the body of normal and adaptive functions, introducing continuation arguments for adaptive functions; the translation of **mfun** is explained in more detail below. Note that $[\![e]\!]^{\flat}$ is not defined for adaptive applications or TDT primitives, as these expressions may not appear in the body of a well-typed normal function. For Src expressions translated to Tgt expressions that are evaluated in direct style (e.g., $e_1\ e_2$), the translation delivers the result to the continuation $v_k$. The type translation is extended pointwise to Src store and variable typing contexts $\Sigma$ and $\Gamma$; the value translation is extended pointwise to Src stores $\sigma$.

The expression translation $[\![e^{\mathsf{src}}]\!]^{\$}\ v_k^{\mathsf{tgt}} = e^{\mathsf{tgt}}$ takes an explicit Tgt value $v_k^{\mathsf{tgt}}$ continuation. It is a standard CPS conversion except that each Src TDT primitive is converted into an equivalent Tgt version with an explicit continuation $v_k$, and the memoizing adaptive functions is translated to use Tgt-level memoization and single-write modrefs.

The **halt** expression is not in the image of the translation, but it can be used as an initial identity continuation $\mathbf{id} = \lambda x.\mathbf{halt}\ x$ for evaluating a CPS-converted program.

$$
\begin{aligned}
[\![\mathbf{nat}]\!] &= \mathbf{nat} \\
\left[\!\!\left[\tau_{\mathrm{x}} \xrightarrow{\$} \tau\right]\!\!\right] &= [\![\tau_{\mathrm{x}}]\!] \to ([\![\tau]\!] \to \mathbf{res}) \to \mathbf{res} \\
[\![\tau\ \mathbf{tdt}]\!] &= [\![\tau]\!]\ \mathbf{tdt}
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]^\flat &= x \\
[\![\mathbf{zero}]\!]^\flat &= \mathbf{zero} \\
[\![\mathbf{succ}\, v]\!]^\flat &= \mathbf{succ}\ [\![v]\!]^\flat \\
[\![\ell]\!]^\flat &= \ell \\
[\![\mathbf{fun}\, f.x.e]\!]^\flat &= \mathbf{fun}\, f.x.([\![e]\!]^\flat) \\
[\![\mathbf{afun}\, f.x.e]\!]^\flat &= \mathbf{fun}\, f.x.\lambda k.([\![e]\!]^\$\ k) \\
[\![\mathbf{mfun}\, f.x.e]\!]^\flat &= \\
&\quad \mathbf{fun}\, f.x.\lambda k_1. && 1 \\
&\quad\ \ \mathbf{let}\ k_2\ =\ \lambda y_{\mathrm{r}}.\mathbf{memo}\,(k_1\ y_{\mathrm{r}})\ \mathbf{in} && 2 \\
&\quad\ \ \mathbf{put\_k}\ k_2\ (\lambda y_{\mathrm{k}}. && 3 \\
&\quad\ \ \ \ \mathbf{let}\ k'\ =\ \lambda y_{\mathrm{r}}.\mathbf{get\_k}\ y_{\mathrm{k}}\,(\lambda k_3.k_3\ y_{\mathrm{r}})\ \mathbf{in} && 4 \\
&\quad\ \ \ \ \mathbf{memo}\,([\![e]\!]^\$\ k')) && 5 \\
[\![\mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, x.e_{\mathrm{s}}]\!]^\flat &= \mathbf{caseN}\ [\![v_{\mathrm{n}}]\!]^\flat\ ([\![e_{\mathrm{z}}]\!]^\flat)\,(x.\,[\![e_{\mathrm{s}}]\!]^\flat) \\
[\![e_{\mathrm{f}}\, e_{\mathrm{x}}]\!]^\flat &= [\![e_{\mathrm{f}}]\!]^\flat\ [\![e_{\mathrm{x}}]\!]^\flat
\end{aligned}
$$

$$
\begin{aligned}
[\![v]\!]^\$\ v_{\mathrm{k}} &= v_{\mathrm{k}}\ [\![v]\!]^\flat \\
[\![\mathbf{caseN}\, v_{\mathrm{n}}\, e_{\mathrm{z}}\, x.e_{\mathrm{s}}]\!]^\$\ v_{\mathrm{k}} &= \mathbf{caseN}\ [\![v_{\mathrm{n}}]\!]^\flat\ ([\![e_{\mathrm{z}}]\!]^\$\ v_{\mathrm{k}})\,(x.\,[\![e_{\mathrm{s}}]\!]^\$\ v_{\mathrm{k}}) \\
[\![e_{\mathrm{f}}\, e_{\mathrm{x}}]\!]^\$\ v_{\mathrm{k}} &= [\![e_{\mathrm{f}}]\!]^\$\ (\lambda y_{\mathrm{f}}.\,[\![e_{\mathrm{x}}]\!]^\$\ (\lambda y_{\mathrm{x}}.v_{\mathrm{k}}\,(y_{\mathrm{f}}\ y_{\mathrm{x}}))) \\
[\![e_{\mathrm{f}}\,\$\,e_{\mathrm{x}}]\!]^\$\ v_{\mathrm{k}} &= [\![e_{\mathrm{f}}]\!]^\$\ (\lambda y_{\mathrm{f}}.\,[\![e_{\mathrm{x}}]\!]^\$\ (\lambda y_{\mathrm{x}}.(y_{\mathrm{f}}\ y_{\mathrm{x}})\ v_{\mathrm{k}})) \\
[\![\mathbf{mk}\, v]\!]^\$\ v_{\mathrm{k}} &= \mathbf{mk\_k}\ [\![v]\!]^\flat\ v_{\mathrm{k}} \\
[\![\mathbf{op}\, v_{\mathrm{l}}\, v_{\mathrm{arg}}]\!]^\$\ v_{\mathrm{k}} &= \mathbf{op\_k}\ [\![v_{\mathrm{l}}]\!]^\flat\ [\![v_{\mathrm{arg}}]\!]^\flat\ v_{\mathrm{k}}
\end{aligned}
$$

Figure 5.1: ACPS type translation $[\![\tau^{\mathsf{src}}]\!] = \tau^{\mathsf{tgt}}$ (top) and term translations $[\![e^{\mathsf{src}}]\!]^\flat = e^{\mathsf{tgt}}$ (middle) and $[\![e^{\mathsf{src}}]\!]^\$\ v_k^{\mathsf{tgt}} = e^{\mathsf{tgt}}$ (bottom).

The CPS discipline in Tgt facilitates identifying the scope of an adaptive store action as the rest of the computation, so change-propagating an inconsistent store action will re-execute the tail of the trace. The translation of memoizing functions ($\mathbf{mfun}$) is central to producing effective self-adjusting programs through compilation. Memoizing a function, however, in the presence of (possibly different) continuations and store mutation is subtle and crucially relies on two ideas: threading continuations through the store with single-write modrefs, and using explicit Tgt-level $\mathbf{memo}$ operations before and after the function body, which serves to isolate a function call from the rest of the computation. Note that a naïve translation:

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

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 Src function into a Tgt function consists of two parts, which we explain by referring to the line numbers in the translation of $\mathbf{mfun}$. The translation inserts memo commands at the function call *and* return points in an attempt to isolate reuse of the function body separately from reuse of the rest of the computation. Note that, unlike the translation of a non-memoizing adaptive function ($\mathbf{afun}$), 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, $\mathbf{putk}$ in the function body and $\mathbf{getk}$ in the continuation). Therefore, if change-propagation starts re-executing due to an inconsistent operation *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 shifting the continuation to the store 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. After the function body is change-propagated without cost, the continuation $k'$ wrapper will resume the (new) continuation by reading it from the store and passing it the memoized 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 operation *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

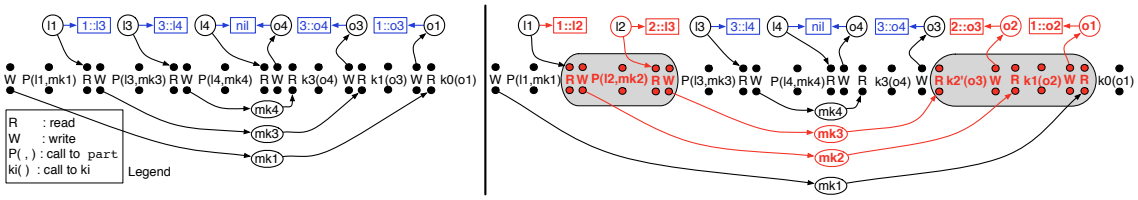



Figure 5.2: Execution of `partition` on lists `[1,3]` (left) and `[1,2,3]` (right).

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 Tgt 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.

**Example 34**

To illustrate how translated programs execute, recall the `partition` function (Figure 2.2) and its translation (Figure 2.3). 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 5.2 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`—stealing allows the write to reuse `mk3`—and `part` is called with `l3` and `mk3`. This call is in the reuse trace, so there is a memoization hit,

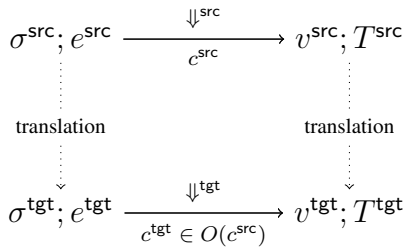



Figure 5.3: The correspondence between the source and target from-scratch runs.

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`—again, stealing allows the write to reuse `o1`—, 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.

### 5.2.1 Meta-Theory

The correctness and efficiency of the translation is captured by the fact that well-typed Src programs are compiled into statically-, dynamically-, and cost-equivalent well-typed Tgt programs with the same asymptotic complexity for from-scratch runs—i.e., Tgt evaluation with an empty reuse trace. The type preservation result is standard from type-directed compilation. More importantly, the evaluation and asymptotic cost of from-scratch runs of Src programs is preserved by compilation. This correspondence is intuitively captured in Figure 5.3.

**Theorem 35 (Static preservation)**
*If* $\Sigma; \Gamma \vdash^{\flat} e : \tau$,
*then* $[\![\Sigma]\!]\,; [\![\Gamma]\!] \vdash [\![e]\!]^{\flat} : [\![\tau]\!]$.
*If* $\Sigma; \Gamma \vdash^{\flat} v : \tau$,
*then* $[\![\Sigma]\!]\,; [\![\Gamma]\!] \vdash [\![v]\!]^{\flat} : [\![\tau]\!]$.
*If* $\Sigma; \Gamma \vdash^{\$} e : \tau$
*and* $[\![\Sigma]\!]\,; [\![\Gamma]\!]\,, \Gamma' \vdash v_{\mathrm{k}} : [\![\tau]\!] \to \mathbf{res}$,
*then* $[\![\Sigma]\!]\,; [\![\Gamma]\!]\,, \Gamma' \vdash [\![e]\!]^{\$}\ v_{\mathrm{k}} : \mathbf{res}$.

**Proof:** By simultaneous induction on the first derivation.

Case $\mathbf{mfun}\, f.x.e$.

$$\llbracket\Sigma\rrbracket\,;\llbracket\Gamma\rrbracket\,,f:\left\llbracket\tau_{\mathrm{x}}\xrightarrow{\$}\tau\right\rrbracket,x:\llbracket\tau\rrbracket,k_1:\llbracket\tau\rrbracket\to\mathbf{res}\vdash\lambda y_{\mathrm{r}}.\mathbf{memo}\,(k_1\;y_{\mathrm{r}}):\llbracket\tau\rrbracket\to\mathbf{res}$$ typing

$$\llbracket\Sigma\rrbracket\,;\llbracket\Gamma\rrbracket\,,y_{\mathrm{k}}:(\llbracket\tau\rrbracket\to\mathbf{res})\,\mathbf{modref}\vdash\lambda y_{\mathrm{r}}.\mathbf{get\_k}\;y_{\mathrm{k}}\;(\lambda k_3.k_3\;y_{\mathrm{r}}):\llbracket\tau\rrbracket\to\mathbf{res}$$ typing

$$\llbracket\Sigma\rrbracket\,;\llbracket\Gamma\rrbracket\,,f:\left\llbracket\tau_{\mathrm{x}}\xrightarrow{\$}\tau\right\rrbracket,x:\llbracket\tau\rrbracket,k':\llbracket\tau\rrbracket\to\mathbf{res}\vdash\llbracket e\rrbracket^{\$}\;k':\llbracket\tau\rrbracket$$ i.h.

$$\llbracket\Sigma\rrbracket\,;\llbracket\Gamma\rrbracket\,,f:\left\llbracket\tau_{\mathrm{x}}\xrightarrow{\$}\tau\right\rrbracket,x:\llbracket\tau\rrbracket,k':\llbracket\tau\rrbracket\to\mathbf{res}\vdash\mathbf{memo}\,(\llbracket e\rrbracket^{\$}\;k'):\mathbf{res}$$ typing

$$\llbracket\Sigma\rrbracket\,;\llbracket\Gamma\rrbracket\,,f:\left\llbracket\tau_{\mathrm{x}}\xrightarrow{\$}\tau\right\rrbracket,x:\llbracket\tau\rrbracket,k_2:\llbracket\tau\rrbracket\to\mathbf{res}\vdash\mathbf{put\_k}\;k_2\;(\ldots):\mathbf{res}$$ typing

$$\llbracket\Sigma\rrbracket\,;\llbracket\Gamma\rrbracket\vdash\llbracket\mathbf{mfun}\,f.x.e\rrbracket^{\flat}:\left\llbracket\tau_{\mathrm{x}}\xrightarrow{\$}\tau\right\rrbracket$$ typing

The remaining cases are analogous.

■

**Theorem 36 (Dynamic and cost preservation)**
*If* $\mathcal{E};\ \sigma_0; e_0\ \Downarrow\ \sigma_1; v_1\ ; T; c_0$,
*and* $\bullet; \llbracket\sigma_1\rrbracket \uplus \sigma_{\mathrm{k}}; v_{\mathrm{k}}\ \llbracket v_1\rrbracket^{\flat} \Downarrow_{\mathrm{E}} T_{\mathrm{k}}; \sigma_2; v_2; \langle \_, c_1\rangle$,
*then* $\bullet; \llbracket\sigma_0\rrbracket \uplus \sigma_{\mathrm{k}}; \llbracket e_0\rrbracket^{\$}\ v_{\mathrm{k}} \Downarrow_{\mathrm{E}} T'; \sigma_2 \uplus \sigma_{\mathrm{e}}; v_2; \langle \_, c_2\rangle$
*and* $c_0 + c_1 \le c_2 \le 4 \cdot c_0 + c_1$ *whence* $c_2 \in \Theta(c_0 + c_1)$.

**Proof:** By induction on the first derivation.

The cost bounds are elided in the proof, they can be obtained by inspecting the trace translation. We show the interesting case of **app**, the remaining cases are straightforward.

Case $D_1$ is **app**.

$D_1 :: \_;\ \sigma_0; e_1\,\$\,e_2\ \Downarrow\ \sigma_0'''; v\ ; \_; \_$ assumption
$v_1 := \mathbf{mfun}\, f.x.e$ abbreviation
$e' := [\,v_1\,/\,f\,]\,[\,v_2\,/\,x\,]\,e$ abbreviation
$D_{11} :: \_;\ \sigma_0; e_1\ \Downarrow\ \sigma_0'; v_1\ ; \_; \_$ subderivation
$D_{12} :: \_;\ \sigma_0'; e_2\ \Downarrow\ \sigma_0''; v_2\ ; \_; \_$ subderivation

$D_{13} :: \_;\ \sigma_0'';e'\ \Downarrow\ \sigma_0''';v\ ;\_;\_$ subderivation

$D_2 :: \bullet;\sigma_\mathrm{k} \uplus [\![\sigma_0''']\!]\,;v_\mathrm{k}\ [\![v]\!]^\flat \Downarrow_\mathrm{E} \_;\sigma';v';\_$ assumption

$D_\ell :: \ell \notin \mathrm{dom}\,\sigma' \supseteq \mathrm{dom}\,\sigma_\mathrm{k} \uplus [\![\sigma_0''']\!]$ fresh location, lemma

$k_w' := \lambda y_\mathrm{r}.\mathbf{memo}\,(v_\mathrm{k}\ y_\mathrm{r})$ abbreviation

$\sigma_1 := [\ell \mapsto k_w']$ abbreviation

$D_2' :: \bullet;(\sigma_\mathrm{k} \uplus [\![\sigma_0''']\!]) \uplus \sigma_1;v_\mathrm{k}\ [\![v]\!]^\flat \Downarrow_\mathrm{E} \_;\sigma' \uplus \sigma_1;v';\_$ frame lemma on $D_2$, $D_\ell$

$k_r' := \lambda y_\mathrm{r}.\mathbf{get\_k}\,\ell\,(\lambda k_3.k_3\ y_\mathrm{r})$ abbreviation

$D_2'' :: \bullet;(\sigma_\mathrm{k} \uplus \sigma_1) \uplus [\![\sigma_0''']\!]\,;k_\mathrm{r}\ [\![v]\!]^\flat \Downarrow_\mathrm{E} \_;\sigma' \uplus \sigma_1;v';\_$ rules **getk**, **memo** on $D_2'$

$E_3 :: \bullet;(\sigma_\mathrm{k} \uplus \sigma_1) \uplus [\![\sigma_0'']\!]\,;[\![e']\!]^\$\ k_\mathrm{r}' \Downarrow_\mathrm{E} \_;(\sigma' \uplus \sigma_1) \uplus \sigma_\mathrm{e};v;\_$ i.h. on $D_{13}$, $D_2''$

$k_w := \lambda y_\mathrm{r}.\mathbf{memo}\,(k_1\ y_\mathrm{r})$ abbreviation

$k_r := \lambda y_\mathrm{r}.\mathbf{get\_k}\ y_\mathrm{k}\,(\lambda k_3.k_3\ y_\mathrm{r})$ abbreviation

$E_3' :: \bullet;\sigma_\mathrm{k} \uplus [\![\sigma_0'']\!]\,;\mathbf{put\_k}\ k_\mathrm{w}'\,(\lambda y_\mathrm{k}.\mathbf{memo}\,([\![e']\!]^\$\ k_\mathrm{r})) \Downarrow_\mathrm{K} \_;\sigma' \uplus (\sigma_1 \uplus \sigma_\mathrm{e});v';\_$
rules **putk**, **memo** on $E_3$

$k_2' := \lambda y_\mathrm{x}.([\![v_1]\!]^\flat\ y_\mathrm{x})\,v_\mathrm{k}$ abbreviation

$E_3'' :: \bullet;\sigma_\mathrm{k} \uplus [\![\sigma_0'']\!]\,;k_2'\ [\![v_2]\!]^\flat \Downarrow_\mathrm{E} \_;\sigma' \uplus (\sigma_1 \uplus \sigma_\mathrm{e});v';\_$ rule **red** on $E_3'$

$E_2 :: \bullet;\sigma_\mathrm{k} \uplus [\![\sigma_0']\!]\,;[\![e_2]\!]^\$\ k_2' \Downarrow_\mathrm{E} \_;\sigma' \uplus (\sigma_1 \uplus \sigma_\mathrm{e}) \uplus \sigma_2;v';\_$ i.h. on $D_{12}$, $E_3''$

$k_2 := \lambda y_\mathrm{x}.(y_\mathrm{f}\ y_\mathrm{x})\,v_\mathrm{k}$ abbreviation

$k_1 := \lambda y_\mathrm{f}.\,[\![e_2]\!]^\$\ k_2$ abbreviation

$E_2' :: \bullet;\sigma_\mathrm{k} \uplus [\![\sigma_0']\!]\,;k_1\ [\![v_1]\!]^\flat \Downarrow_\mathrm{E} \_;\sigma' \uplus (\sigma_1 \uplus \sigma_\mathrm{e} \uplus \sigma_2);v';\_$ rule **red** on $E_2$

$E_1 :: \bullet;\sigma_\mathrm{k} \uplus [\![\sigma_0]\!]\,;[\![e_1]\!]^\$\ k_1 \Downarrow_\mathrm{E} \_;\sigma' \uplus (\sigma_1 \uplus \sigma_\mathrm{e} \uplus \sigma_2 \uplus \sigma_1);v';\_$ i.h. on $D_{11}$, $E_2'$

$E_1' :: \bullet;\sigma_\mathrm{k} \uplus [\![\sigma_0]\!]\,;[\![e]\!]^\$\ v_\mathrm{k} \Downarrow_\mathrm{E} \_;\sigma' \uplus (\sigma_1 \uplus \sigma_\mathrm{e} \uplus \sigma_2 \uplus \sigma_1);v';\_$ rule **red** on $E_1$

■

The store $\sigma_k$ accounts for locations free in the continuation $v_k$, while the store $\sigma_e$ accounts for locations allocated for (the continuations of) memoizing functions. Instantiating this theorem with the identity continuation $v_k = \mathbf{id}$, we have that evaluation of a Src program coincides with (from-scratch) Tgt evaluation of its ACPS-translation. Furthermore, the adaptive work $c_2 \in \Theta(c_0)$ in Tgt is proportional to the active work $c_0$ in Src, because the work of the identity continuation is constant. This means that the translation preserves the asymptotic complexity of from-scratch runs.

## 5.3 Trace Translation

The Tgt trace of an ACPS-compiled Src program is richer than its Src counterpart because Tgt traces have explicit continuations and the ACPS translation introduces administrative redices, threads continuations through the store, and inserts memoization at function call and return points. Trace translation is syntax-directed, except for the choice of locations for continuations of memoizing functions; below we specify how these locations are chosen.

**Context Translation.** Trace translation requires annotating Src actions with an evaluation context $\mathcal{E} ::= \Box \mid \mathcal{E}\, e_x \mid v_f\, \mathcal{E}$ and instrumenting the dynamic semantics to determine the evaluation context of each action. The Src dynamic semantics and distance, however, are sufficiently instrumented to translate Src traces into equivalent Tgt traces. An explicit Src evaluation context $\mathcal{E}$ is sufficient to reify the current continuation $[\![\mathcal{E}]\!]\ v_k^{\mathsf{tgt}}$ relative to an initial Tgt continuation $v_k^{\mathsf{tgt}}$:

$$\begin{aligned}
[\![\Box]\!]\ v_{\mathrm{k}} &= v_{\mathrm{k}} \\
[\![\mathcal{E}\, e_{\mathrm{x}}]\!]\ v_{\mathrm{k}} &= [\![\mathcal{E}]\!]\ (\lambda y_{\mathrm{f}}.\, [\![e_{\mathrm{x}}]\!]^{\$}\ (\lambda y_{\mathrm{x}}.(y_{\mathrm{f}}\ y_{\mathrm{x}})\, v_{\mathrm{k}})) \\
[\![v_{\mathrm{f}}\, \mathcal{E}]\!]\ v_{\mathrm{k}} &= [\![\mathcal{E}]\!]\ (\lambda y_{\mathrm{x}}.([\![v_{\mathrm{f}}]\!]^{\flat}\ y_{\mathrm{x}})\, v_{\mathrm{k}})
\end{aligned}$$

**Local Trace Translation.** Moreover, since active Src actions are instrumented with their local evaluation context, we can give a *trace slice translation* $[\![S^{\mathsf{imp}}]\!]\ v_k^{\mathsf{tgt}}\, U_k^{\mathsf{tgt}} = U^{\mathsf{tgt}}$ of a Src trace slice $S^{\mathsf{imp}}$ using $v_k^{\mathsf{tgt}}$ as an initial continuation (to extend the local context $\mathcal{E}$ of actions) and suffix slice list $U_k^{\mathsf{tgt}}$ to produce a Tgt slice list $U^{\mathsf{tgt}}$ corresponding to the original computation (with explicit holes). The trace slice translation also induces a *trace translation* $[\![T^{\mathsf{imp}}]\!]\ v_k^{\mathsf{tgt}}\, (T_k^{\mathsf{tgt}}, \bullet) = (T^{\mathsf{tgt}}, \bullet)$.

The translation of the empty trace and TDT actions is straightforward:

$$\begin{aligned}
[\![\varepsilon]\!]\ v_{\mathrm{k}}\, U_{\mathrm{k}} &= U_{\mathrm{k}} \\
\left[\!\!\left[\mathtt{mk}_{\mathcal{E}}^{v\uparrow\ell}\cdot S\right]\!\!\right]\ v_{\mathrm{k}}\, U_{\mathrm{k}} &= \mathtt{mk}_{[\![\mathcal{E}]\!]\ v_{\mathrm{k}}}^{[\![v]\!]^{\flat}\uparrow\ell}\cdot([\![S]\!]\ v_{\mathrm{k}}\, U_{\mathrm{k}}) \\
\left[\!\!\left[\mathtt{op}_{\mathcal{E}}^{\ell, v_{\mathrm{arg}}\downarrow v_{\mathrm{res}}}\cdot S\right]\!\!\right]\ v_{\mathrm{k}}\, U_{\mathrm{k}} &= \mathtt{op}_{[\![\mathcal{E}]\!]\ v_{\mathrm{k}}\checkmark}^{\ell, [\![v_{\mathrm{arg}}]\!]^{\flat}\downarrow[\![v_{\mathrm{res}}]\!]^{\flat}}\cdot([\![S]\!]\ v_{\mathrm{k}}\, U_{\mathrm{k}})
\end{aligned}$$

Note that the Src evaluation context $\mathcal{E}$ is used to extend the continuation $v_k$ in the Tgt trace.

Since a failure action is inserted at a function's return point, it is translated to the trace

that follows the evaluation of a function body (*cf.*, ACPS function translation):

$$\begin{aligned}\left[\!\left[\mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}\cdot S'\right]\!\right]\ v_{\mathrm{k}}\ U_{\mathrm{k}} &= \mathtt{get}^{\ell\downarrow k_{\mathrm{w}}}_{k_{\mathrm{a}}\checkmark}\cdot\mathtt{memo}^{(([\![\mathcal{E}]\!]\ v_{\mathrm{k}})\ [\![v]\!]^{\flat})}\cdot([\![S']\!]\ v_{\mathrm{k}}\ U_{\mathrm{k}})\\ \text{where } k_{\mathrm{w}} &= \lambda y_{\mathrm{r}}.\mathbf{memo}\,((\![\mathcal{E}]\!]\ v_{\mathrm{k}})\ y_{\mathrm{r}})\\ k_{\mathrm{a}} &= \lambda y_{\mathrm{k}}.y_{\mathrm{k}}\ [\![v]\!]^{\flat}\end{aligned}$$

Note that $k_{\mathrm{w}}$ is the memoizing version of the original continuation that was written to the store before the evaluation of the body and $k_{\mathrm{a}}$ is the continuation of the $\mathbf{getk}$ command that fetches the memoizing version of original continuation.

The translation of a memoizing function action must account for writing the memoizing version of the original continuation to the store before memoizing on the evaluation of the body:

$$\begin{aligned}\left[\!\left[\mathtt{app}^{(\mathbf{mfun}\,f.x.e)\$v_{\mathrm{x}}\Downarrow v}_{\mathcal{E}(\ell)}(S)\cdot S'\right]\!\right]\ v_{\mathrm{k}}\ U_{\mathrm{k}} &= \mathtt{put}^{k_{\mathrm{w}}\uparrow\ell}_{k_{\mathrm{m}}}\cdot\mathtt{memo}^{([\![e']\!]^{\$}\ k_{\mathrm{r}})}\cdot([\![S]\!]\ k_{\mathrm{r}}\ U_{\mathrm{r}})\\ \text{where } k_{\mathrm{w}} &= \lambda y_{\mathrm{r}}.\mathbf{memo}\,((\![\mathcal{E}]\!]\ v_{\mathrm{k}})\ y_{\mathrm{r}})\\ k_{\mathrm{m}} &= \lambda y_{\mathrm{l}}.\mathbf{memo}\,([\![e']\!]^{\$}\ (\lambda y_{\mathrm{r}}.\mathbf{get\_k}\ y_{\mathrm{l}}\ (\lambda y_{\mathrm{k}}.y_{\mathrm{k}}\ y_{\mathrm{r}})))\\ e' &= [\,\mathbf{mfun}\,f.x.e\,/\,f\,]\,[\,v_{\mathrm{x}}\,/\,x\,]\,e\\ k_{\mathrm{r}} &= \lambda y_{\mathrm{r}}.\mathbf{get\_k}\,\ell\,(\lambda y_{\mathrm{k}}.y_{\mathrm{k}}\ y_{\mathrm{r}})\\ U_{\mathrm{r}} &= \left[\!\left[\mathtt{fail}^{\Downarrow v}_{\mathcal{E}(\ell)}\cdot S'\right]\!\right]\ v_{\mathrm{k}}\ U_{\mathrm{k}}\end{aligned}$$

Memoizing function actions are instrumented with a location to indicate where the continuation is threaded through the store, and their translation accounts for memoizing at the function call and return points. If the trace of the function body is present, a memoization action precedes the trace; otherwise a hole marker indicates the body was removed by slicing.

$$\begin{aligned}\left[\!\left[\mathtt{app}^{(\mathbf{mfun}\,f.x.e)\$v_{\mathrm{x}}\Downarrow v}_{\mathcal{E}(\ell)}(\dot{S})\cdot S'\right]\!\right]\ v_{\mathrm{k}}\ U_{\mathrm{k}} &= \begin{cases}\mathtt{put}^{k_{\mathrm{w}}\uparrow\ell}_{k_{\mathrm{m}}}\cdot\mathtt{memo}^{([\![e']\!]^{\$}\ k_{\mathrm{r}})}\cdot([\![S]\!]\ k_{\mathrm{r}}\ (\mathtt{get}^{\ell\downarrow k_{\mathrm{w}}}_{k_{\mathrm{a}}-}\cdot U_{\mathrm{t}})) & \text{if } \dot{S}=S\\ \mathtt{put}^{k_{\mathrm{w}}\uparrow\ell}_{k_{\mathrm{m}}}\cdot\mathtt{hole}^{([\![e']\!]^{\$}\ k_{\mathrm{r}})},U_{\mathrm{t}} & \text{if } \dot{S}=\circ\end{cases}\\ \text{where } k_{\mathrm{w}} &= \lambda y_{\mathrm{r}}.\mathbf{memo}\,((\![\mathcal{E}]\!]\ v_{\mathrm{k}})\ y_{\mathrm{r}})\\ k_{\mathrm{m}} &= \lambda y_{\mathrm{l}}.\mathbf{memo}\,([\![e']\!]^{\$}\ (\lambda y_{\mathrm{r}}.\mathbf{get\_k}\ y_{\mathrm{l}}\ (\lambda y_{\mathrm{k}}.y_{\mathrm{k}}\ y_{\mathrm{r}})))\\ e' &= [\,\mathbf{mfun}\,f.x.e\,/\,f\,]\,[\,v_{\mathrm{x}}\,/\,x\,]\,e\\ k_{\mathrm{r}} &= \lambda y_{\mathrm{r}}.\mathbf{get\_k}\,\ell\,(\lambda y_{\mathrm{k}}.y_{\mathrm{k}}\ y_{\mathrm{r}})\\ U_{\mathrm{t}} &= \mathtt{memo}^{(([\![\mathcal{E}]\!]\ v_{\mathrm{k}})\ [\![v]\!]^{\flat})}\cdot([\![S']\!]\ v_{\mathrm{k}}\ U_{\mathrm{k}})\\ k_{\mathrm{a}} &= \lambda y_{\mathrm{k}}.y_{\mathrm{k}}\ [\![v]\!]^{\flat}\end{aligned}$$

Note that $k_{\mathrm{r}}$ is the continuation that fetches and invokes the memoizing version of the original continuation; this is the continuation that is passed to the body. The body of the memoizing function action is translated with respect to $k_{\mathrm{r}}$ and $T_{\mathrm{r}}$, which is the translation of a failure action.

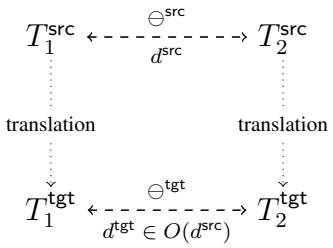



Figure 5.4: The correspondence between distance in the source and target.

**Global Trace Translation.** The *extracted trace translation* $\langle\!\langle S^{\mathsf{imp}} \rangle\!\rangle = U^{\mathsf{tgt}}$ translates a slice $S^{\mathsf{imp}}$ of the form $\mathsf{app}^{(\mathbf{mfun}\, f.x.e)\$v_{\mathrm{x}} \Downarrow v}_{\mathcal{E}(\ell)}(S)\cdot\varepsilon$ extracted from a larger computation. The translation shares the auxiliary definitions of the memoizing function translation, it begins with the memoized evaluation of the application and ends in a hole marker returning to the continuation:

$$\langle\!\langle \mathsf{app}^{(\mathbf{mfun}\, f.x.e)\$v_{\mathrm{x}} \Downarrow v}_{\mathcal{E}(\ell)}(S)\cdot\varepsilon \rangle\!\rangle = \mathtt{memo}^{(\llbracket e' \rrbracket^{\$}\, k_{\mathrm{r}})}\cdot(\llbracket S \rrbracket\ k_{\mathrm{r}}\, (\mathtt{get}^{\ell \downarrow k_{\mathrm{w}}}_{k_{\mathrm{a}}}\cdot \mathtt{hole}^{((\llbracket \mathcal{E} \rrbracket\, v_{\mathrm{k}})\, \llbracket v \rrbracket^{\flat})}, \bullet))$$

Note that the translations $\llbracket M(\circ)\cdot S' \rrbracket\ v_k\, T_k$ and $\langle\!\langle M(S) \rangle\!\rangle$ are equivalent (modulo permutation) to slicing the translation $\llbracket M(S)\cdot S' \rrbracket\ v_k\, T_k$ at the function call and return points.

Finally, the translation $\llbracket U^{\mathsf{imp}} \rrbracket\ v_k^{\mathsf{tgt}}\, U_k^{\mathsf{tgt}} = U^{\mathsf{tgt}}$ of a non-empty Src$*$ slice list concatenates the translation of the skeleton and the extracted translation of the subcomputations:

$$\llbracket S, \langle S_{\mathrm{i}} \rangle_{\mathrm{i} \in 1..n} \rrbracket\ v_{\mathrm{k}}\, U_{\mathrm{k}} = (\llbracket S \rrbracket\ v_{\mathrm{k}}\, U_{\mathrm{k}}), \langle\!\langle S_{\mathrm{i}} \rangle\!\rangle_{\mathrm{i} \in 1..n}$$

Given the trace translation, Theorem 36 can be strengthened to show that the if the continuation $v_k$ is of the form $\llbracket \mathcal{E} \rrbracket\ v'_k$, then the Tgt evaluation trace $T'$ is related to $\llbracket T \rrbracket\ v_k\, (T_k, \bullet) = (T', \bullet)$.

### 5.3.1 Meta-Theory

Finally, Src distance may be related to Tgt distance by trace translation. This correspondence is intuitively captured in Figure 5.4.

**Theorem 37 (Translation preserves Src precise/Tgt local distance)**
*Assume* $U_{k1} \boxminus U_{k2} = \langle \_, c'_1 \rangle$ *and* $U_{k1} \ominus U_{k2} = \langle \_, c'_2 \rangle$.
*If* $S_1 \boxminus S_2 = \_, b, \langle \_, c \rangle$ *(precise),*
*then* $(\llbracket S_1 \rrbracket\ v_{k1}\, U_{k1}) \boxminus (\llbracket S_2 \rrbracket\ v_{k2}\, U_{k2}) = \langle \_, c'' \rangle$
*and* $c'' = c + \mathsf{if}\ b\ \mathsf{then}\ c'_1\ \mathsf{else}\ c'_2$.

*If* $S_1 \ominus S_2 = \_, b, \langle \_, c \rangle$ *(precise),*
*then* $(\llbracket S_1 \rrbracket\ v_{k1}\, U_{k1}) \ominus (\llbracket S_2 \rrbracket\ v_{k2}\, U_{k2}) = \langle \_, c'' \rangle$
*and* $c'' = c + \mathsf{if}\ b\ \mathsf{then}\ c'_1\ \mathsf{else}\ c'_2$.

**Proof:** We preprocess the precise Src distance derivation by assigning matching fresh locations to memoization actions that synchronize, these are used by the trace translation for continuations (this is always possible because stores and traces are finite). Next, we proceed by induction on the (instrumented) precise Src distance derivation, using the trace translation to build an equivalent Tgt distance derivation. ■

**Corollary 38 (Translation preserves Src simple/Tgt local distance)**
*Assume* $U_{k1} \boxminus U_{k2} = \langle \_, c'_1 \rangle$ *and* $U_{k1} \ominus U_{k2} = \langle \_, c'_2 \rangle$.
*If* $S_1 \boxminus S_2 = \langle \_, c \rangle$ *(simple),*
*then* $(\llbracket S_1 \rrbracket\ v_{k1}\, U_{k1}) \boxminus (\llbracket S_2 \rrbracket\ v_{k2}\, U_{k2}) = \langle \_, c'' \rangle$
*and* $c \leq c'' \leq 6 \cdot c + \max\{c'_1, c'_2\}$.
*If* $S_1 \ominus S_2 = \langle \_, c \rangle$ *(simple),*
*then* $(\llbracket S_1 \rrbracket\ v_{k1}\, U_{k1}) \ominus (\llbracket S_2 \rrbracket\ v_{k2}\, U_{k2}) = \langle \_, c'' \rangle$
*and* $c \leq c'' \leq 6 \cdot c + \max\{c'_1, c'_2\}$.

**Proof:** By Theorems 24 and 37. ■

**Corollary 39 (Src/Tgt local distance soundness)**
*Let* $U_{\mathbf{id}i}$ *be the identity continuation trace for* $S_i$ *(*$i \in \{1, 2\}$*).*
*If* $S_1 \boxminus S_2 = \langle \_, c \rangle$ *(simple),*
*then* $(\llbracket S_1 \rrbracket\ v_{k1}\, U_{\mathbf{id}1}) \boxminus (\llbracket S_2 \rrbracket\ v_{k2}\, U_{\mathbf{id}2}) = \langle \_, c'' \rangle$ *and* $c'' \in \Theta(c)$.
*If* $S_1 \ominus S_2 = \langle \_, c \rangle$ *(simple),*
*then* $(\llbracket S_1 \rrbracket\ v_{k1}\, U_{\mathbf{id}1}) \ominus (\llbracket S_2 \rrbracket\ v_{k2}\, U_{\mathbf{id}2}) = \langle \_, c'' \rangle$ *and* $c'' \in \Theta(c)$.

**Proof:** The search distance $T^{\mathsf{tgt}}_{\mathbf{id}1} \boxminus T^{\mathsf{tgt}}_{\mathbf{id}2}$ and synchronization distance $T^{\mathsf{tgt}}_{\mathbf{id}1} \ominus T^{\mathsf{tgt}}_{\mathbf{id}2}$ between the identity continuation traces are constant, therefore the asymptotic bound $c'' \in \Theta(c)$ follows by Corollary 38. ■

**Theorem 40 (Translation preserves Src/Tgt global distance)**
*Assume* $U_{k1} \boxminus U_{k2} = \langle \_, c'_1 \rangle$ *and* $U_{k1} \ominus U_{k2} = \langle \_, c'_2 \rangle$.
*If* $S_1 \boxminus^{\gg} S_2 = \langle \_, c \rangle$,
*then* $\llbracket S_1 \rrbracket\ v_{k1}\, U_{k1} \boxminus^{\gg} \llbracket S_2 \rrbracket\ v_{k2}\, U_{k2} = \langle \_, c'' \rangle$,
*and* $c \leq c'' \leq 6 \cdot c + \max\{c'_1, c'_2\}$.

**Proof:** We define an equivalent variant of $\boxminus^{\gg}$ for SrcImp and Tgt that decomposes both traces but only permutes the left-hand trace slices. By induction on the subderivation $S_2 \gg U_2$ of $\boxminus^{\gg}$, there is a permutation of $[\![U_{\mathrm{i}}]\!]\ v_{k\mathrm{i}}\, U_{k\mathrm{i}}$ (which itself is a slicing of $[\![S_{\mathrm{i}}]\!]\ v_{k\mathrm{i}}\, U_{k\mathrm{i}}$) into $U_{p\mathrm{i}}$ ($i \in 1, 2$), such that $U_{p1} \boxminus U_{p2} = \langle \_, c'' \rangle$ using Corollary 38. ■

**Corollary 41** (Src/Tgt **global distance soundness)**
*Let* $U_{\mathbf{id}\mathrm{i}}$ *be the identity continuation trace for* $S_i$ *(*$i \in \{1, 2\}$*).*
*If* $S_1 \boxminus^{\gg} S_2 = \langle \_, c \rangle$,
*then* $[\![S_1]\!]\ \mathbf{id}_1\, U_{\mathbf{id}1} \boxminus^{\gg} [\![S_2]\!]\ \mathbf{id}_2\, U_{\mathbf{id}2} = \langle \_, c'' \rangle$
*and* $c'' \in \Theta(c)$.

**Proof:** The search distance $T^{\mathsf{tgt}}_{\mathbf{id}1} \boxminus T^{\mathsf{tgt}}_{\mathbf{id}2}$ and synchronization distance $T^{\mathsf{tgt}}_{\mathbf{id}1} \ominus T^{\mathsf{tgt}}_{\mathbf{id}2}$ between the identity continuation traces are constant, therefore the asymptotic bound $c'' \in \Theta(c)$ follows by Theorem 40. ■

Note that since Src and Tgt distance are quasi-symmetric, analogous results hold of the left component of distance. This means that change-propagation has the same asymptotic time-complexity as trace distance. The converse of the theorem does not hold: a distance may be derivable of Tgt traces which does not correspond to any derivable Src distance. This incompleteness is to be expected because memoization of a function call and return in Tgt need not match in lock-step, whereas the **synch/memo** (resp. **synch/search**) Src rule requires both (resp. neither) to match in lock-step.

### 5.3.2 Discussion

We briefly remark on some limitations of our approach.

**Incompleteness.** Soundness of the translation guarantees that any distance derivable in Src is also (up to a constant factor) derivable in Tgt. However, the Tgt proof system exhibits more possible distances: in Src, memoization requires matching both the function call and return points, while the ACPS translation into Tgt distinguishes memoization at the call and the return. Therefore, there are more opportunities for switching between search and synchronization in Tgt and there may be more distance values derivable in Tgt than in Src. For example, in Tgt a function call memoization can miss (i.e., remain in search mode) while the return can match (i.e., switch from search to synchronization mode), which is not possible in Src. Consequently, any upper bound found using Src

distance is preserved by compilation, but lower bound arguments on a Src program are not necessarily lower bounds on the Tgt distance. In particular, there may be a lower bound using Src distance, but an asymptotically smaller Tgt distance may be derivable for the ACPS-translated program.

**Nondeterminism.** The dynamic semantics and distance of Src and Tgt programs are nondeterministic due to the freedom in choosing locations as well as deciding when memoization matches. This avoids having to commit to a particular implementation, but also means that any upper bound derived using the nondeterministic semantics may not be realized by a particular implementation. In order for an implementation to realize an upper bound on distance, the allocation and memoization policies used in deriving the distance must coincide with those of the implementation. In previous work Ley-Wild et al. [2008b], we proposed both user-specified and compiler-generated mechanisms for defining allocation and memoization policies, which suffice for realizing the bounds derived in our examples. Ultimately, it would be useful to develop compilation and run-time techniques to automatically minimize the distance between the computations by considering all possible policies.

The analysis of our examples required manually computing distance and picking allocation and memoization policies that minimize distance. It would be useful to have automatic support for computing distance according to particular allocation and memoization policies as well as for automatically minimizing distance by considering all possible policies. Such a mechanism could be used to find the distance for concrete executions under particular inputs, as well as for reasoning abstractly about distance relative to a class of input changes.

# Chapter 6

# Implementation

This chapter is based on work on the implementation of the ΔML language and compiler [Ley-Wild et al., 2008b] and its extension to support traceable data types [Acar et al., 2010a].

## 6.1 Overview

To validate the compilation-based approach to self-adjusting computation, we developed the ΔML language and compiler with modifiable references and extensibility by traceable data types. The ΔML language extends Standard ML [Milner et al., 1997] with the Src language features for direct self-adjusting programming. A user interested in writing self-adjusting programs only needs to be familiar with the language extensions (Section 6.2) and the library interface for adaptive and meta-level operations (Section 6.3). The associated compiler is a modification the MLton[MLt] compiler with syntactic and static typing support (Section 6.4), together with a runtime library (Section 6.5) for self-adjusting computation that provides change-propagation based on adaptive dependence tracking and computation memoization (Subsection 6.5.2). 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]. We briefly discuss the implementation of TDTs (Subsection 6.5.3) and their integration into the ΔML compiler (Subsection 6.5.4).

## 6.2 Language Extensions

We extend Standard ML [Milner et al., 1997] with the Src feature for self-adjusting computation. We introduce an adaptive function type, written "*ty* `-$>` *ty*". An adaptive function is introduced via either the expression form "**afn** *match*" or the declaration forms "**afun** *tyvarseq fvalbind*" and "**mfun** *tyvarseq fvalbind*". All of these forms simply change the introductory keyword of existing Standard ML forms for function introduction (e.g., `fn` and `fun`). Hence, adaptive functions enjoy the same syntactic support for mutually recursive function definitions, clausal function definitions, and pattern matching as (normal) functions. An adaptive function is eliminated via the expression form "*exp* **$** *exp*"; as in Src, adaptive applications may only appear within the body of an adaptive function.

Note that while these language extensions introduce a new type, they do not (significantly) change the type system of Standard ML. 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 Src, a mode component in the typing rules is used to preclude adaptive applications from the body of normal functions. 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.

## 6.3 Library Interface

Figure 6.1 gives the interface of the `Adaptive` library. Controlling the nondeterministic stealing of locations, adaptive identification of invalid work, and computation memoization during change-propagation relies on equality and hashing functions. This is the most complex and subtle aspect of using the adaptive library, although these mechanisms are only necessary to improve the efficiency of change-propagation, not to enforce its correctness. Immutable modifiable references are the basic primitive for adaptivity to identify data and control dependencies. Memoizing functions identify opportunities for computation memoization at function call and return points. Meta-level primitives serve create a program's initial inputs, perform an initial from-scratch execution, and inspect the output. Furthermore, they also serve to change the inputs and propagate them through the computation to obtain an updated output.

```
signature EQ_HASH = sig
 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
end

signature MODREF = sig
 type 'a modref
 val put :  'a -$> 'a modref
 val get :  'a modref -$> 'a

 (** Stealing and Reuse **)
 val mkPutEq :  'k key * 'a eq -$> ('k * 'a -$> 'a modref)
 val mkPut :  unit -$> ('k * 'a -$> 'a modref)
 val putTh :  (unit -$> 'a) -$> 'a modref

 (** Meta operations **)
 structure Meta :  sig
  val new :  'a eq * 'a -> 'a modref
  val change :  'a modref * 'a -> unit
  val deref :  'a modref -> 'a
 end
end

signature ADAPTIVE = sig
 structure EqHash :  EQ_HASH = ...
 structure Modref :  MODREF = ...
 (** Memoization **)
 val memoFix :  ('a key * 'r eq *
                 (('a -$> 'r) -> ('a -$> 'r)))
               -> ('a -$> 'r)
 val memoCont :  ('r eq * ('a -$> 'r))
                -> ('a -$> 'r)

 (** Meta operations **)
 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 = ...
```

Figure 6.1: Signature for the `Adaptive` library.

**Equality and Hashing.** 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. In contrast to Standard ML's polymorphic equality, which may only be instantiated at equality types, the MLton-generated equality and hashing functions may be instantiated at any type; on values of function type, they operate on the structure of the function closure. The default versions make it easy to migrate an ordinary program to a self-adjusting version and avoid the need for awkward tagged values used in previous work [Acar et al., 2006a].

**Modifiable References.** The immutable modifiable reference is realized by the `modref`[1] type with the `get` and `put` adaptive functions.

To control allocation nondeterminism, the adaptive library provides *keyed allocation* to associate each location with a key. During change-propagation, an allocation with a matching key can steal the corresponding location used in the previous run. The `mkPutEq` operation takes a key and an equality predicate and returns an *allocator*, a function for allocating modrefs. 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 modref is reused, the equality predicate determines whether the contents of the modref have changed. This is the implementation realization of the **op/reuse** rule of Figure 4.6 when the result of the operation hasn't changed. Since there is a degree of nondeterminism between the **op/reuse** and **change** rules, the equality predicate need only be conservative.

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

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

[1] Due to historic reasons, the type is called `box` in the implementation.

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.

**Computation Memoization.** The next group of types and values are mechanisms to control the nondeterministic memoization that appears in the dynamic semantics of Chapter 4.

Although Tgt memoization can match whenever identical expressions are evaluated, this semantics is hard to implement because it would require comparing arbitrary expressions and maintaining a global memoization table. Previous work has shown how to use local memo tables indexed by a function's argument, which corresponds to the memo actions of source traces.

The `memoFix` operation is a fixed-point combinator used to create recursive memoizing functions. The operation takes a key tuple on the argument, an equality predicate on the result, and the function to memoize using open recursion. Recall the translation of **mfun** from Figure 4.6. In order to reuse the location returned by $\mathbf{put\_k}\ k_2\ (\lambda y_k.\cdots)$ at line 3, the equality predicate and hash function on the function argument are used to match the corresponding $\mathbf{put\_k}\ k_2\ (\lambda y_k.\cdots)$ 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.[2] 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 memoizing 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 provides additional operations with different memoization properties. For example, the `memoCont` operation implements only the

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

$\mathbf{memo}\ (k_1\ y_r)$ at line 2 and only for non-recursive calls of the memoizing function.

**Meta Operations.** The *meta operations* are used by a host mutator program to control a self-adjusting computation. The meta-level operations for modrefs are used by the host mutator program to create and modify inputs for and inspect outputs of a self-adjusting computation. They include the `new`, `change`, and `deref` operations for creating, updating, and dereferencing modrefs. Other TDTs provide meta-level operations according to their functionality.

The `call` operation is used to perform the initial from-scratch execution of a self-adjusting program. 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 updated output may be observed as the updated contents of the mutable cell. Correctness demands that self-adjusting computations not perform any computation with untracked effects because otherwise change-propagation may not produce an execution equivalent to a from-scratch run. Since the meta operations are effectful but not tracked by the self-adjusting library, they should not be used within adaptive functions.

## 6.4 Compiler Modifications

To implement ΔML, 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. 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 computation. Indeed, the compiler provides no direct support for tracking the dynamic data dependencies of self-adjusting computations or for re-executing computations during change-propagation. That support comes from the self-adjusting-computation libraries (Section 6.5). This approach minimizes 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.

**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 memoizing 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.

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 Chapter 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.

## 6.5 Self-Adjusting Computation Library

**Overview.** The high-level `Adaptive` library (Section 6.5) is implemented as a wrapper around a low-level library for self-adjusting computation. The low-level library effectively implements the semantics of Tgt, providing operations for creating and manipulating TDTs (Subsection 6.5.3), for memoizing functions, and for performing change-propagation (Subsection 6.5.2). 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 compiler's `afn_to_xfn` and `xfn_to_afn` primitives to convert between the two representations. The implementation is based on the previously proposed monadic libraries [Acar et al., 2006b,a] adapted for CPS, which simplifies some key internal data structures and algorithms.

### 6.5.1 Traces and Time Stamps

We represent a trace as a time line data structure TDS where each action—i.e., TDT operations and memoization—is represented by a time stamp. Since change-propagation may identify work for reuse, discard trace segments of stale work and produce new trace segments for re-executed work, the time line must support efficient time stamp comparison, disposal of time stamp intervals, and insertion of new time stamps between existing time stamps.

Simple data structures such as integers and fixed-precision floating-point numbers do not work because they do not allow insertions of new time stamps between two adjacent numbers; arbitrary precision real numbers would work but are not efficient. Instead, we represent time stamps with an order-maintenance data structure [Dietz and Sleator, 1987]

for total orders. We assume all time stamps are taken from the interval $[t_{start}, t_{end}]$ where $t_{start}$ is the beginning and $t_{end}$ is the end of time.

During a from-scratch execution and when re-running invalidated computation, the adaptive and memoization actions generate new time stamps for the new work. During change-propagation, stale trace segments are discarded and existing trace segments that can be reused are spliced into the trace for the updated run. In any case, the insertion or deletion of an action at some point in the trace must identify any subsequent work in the trace—i.e., with later time stamps—that becomes inconsistent.

**Traceable Data Types.** Each TDT instance maintains a local time line to record the time stamp of every action associated with that instance. Each query operation (e.g., modref dereferences) also records its continuation in case it needs to be rerun if the operation becomes inconsistent.

**Allocation and Memoization.** Each keyed allocator and memoizing function (*cf.* Section 6.3) maintains a hash table to map keys (e.g., allocation key or argument hash) to the relevant resource (e.g., location or memoized computation) together with its associated time stamp.

**Remark.** The monadic approach requires two time stamps to delimit the beginning and end of each read operation and memoized computation. By taking advantage of the fact that continuations are explicit in the compilation-based approach, the representation of an action only requires one time stamp. Furthermore, change-propagation becomes a standalone iterative process that re-executes the continuation of inconsistent operations, whereas the monadic approach required a mutual recursion between change-propagation and memoization.

**Out-of-Order Memoization.** To support out-of-order memoization, trace segments may be reordered. Therefore, the time line needs to allow extraction and insertion of chunks of trace. As discussed below, this can be implemented reasonably efficiently.

### 6.5.2 Change Propagation

The semantics of change-propagation in the Tgt language (Chapter 4) explicitly traverses the trace replaying every consistent action and falls back to evaluation when it encounters

Algorithm CPA ($S_1$, $T_2$, $Q$, $t_{now}$)
    let
        $t_{inc}$ = find the next element in $Q$ greater than $t_{now}$
        $T_r$ = $S_1$ [$t_{now}$,$t_{inc}$)
    in if $t_{inc}$ is $t_{end}$
        then return $T_2$ ++ $T_r$
        else let
            $S_1$' = $S_1 \setminus T_r$
            ($t_{memo}$, $Q_n$, $T_n$) = run continuation of $t_{inc}$ until memo match in $S_1$'
                $t_{memo}$ is the time stamp of the memo match
                $T_n$ is the new trace for [$t_{inc}$,$t_{memo}$)
                $Q_n$ is $Q$ extended with inconsistencies from running $T_n$
            $T_2$' = $T_2$ ++ $T_r$ ++ $T_n$
            –for in-order reuse, revoke unusable trace and update propagation queue:
            $T_{stale}$ = $S_1$' [$t_{inc}$,$t_{memo}$)
            $S_1$' updated with $T_{stale}$ removed
            $Q_n$ updated with $T_{stale}$ revoked
        in if $t_{memo}$ is $t_{end}$
            then return $T_2$'
            else return CPA ($S_1$', $T_2$', $Q_n$, $t_{memo}$)
        end
    end

Figure 6.2: The change-propagation algorithm.

an inconsistent action. Here we describe a concrete algorithm CPA and associated data structures to implement the abstract change-propagation semantics of Tgt. This goes into a level more detail than the Tgt semantics allowing an analysis of runtime.

The main idea of CPA is to traverse the trace of the previous run in execution order while identifying the parts of the trace that need to be rerun (the **change** rule transitions from $\curvearrowright$ to $\Downarrow$ in Figure 4.2) and the parts that can be reused (the **memo/hit** rule transitions from $\Downarrow$ to $\curvearrowright$ in Figure 4.6). In particular, it is important to skip over the part that can be reused without incurring any cost. An important aspect, therefore, is to identify the next inconsistent time stamp following a memo match. Once this is identified, the CPA also can splice the trace segment between the memo match and the inconsistency from the previous time line and append it to the new time line. Moreover, the CPA must insert a fresh trace segment for the computation that must be rerun because it wasn't available for reuse from the previous run.

Figure 6.2 gives the pseudo-code for $\mathsf{CPA}(S_1, T_2, Q, t_{now})$ that takes an input trace $S_1$ of the previous run and builds an accumulator trace $T_2$ for the updated run; the algorithm also maintains a *propagation queue* $Q$ of inconsistent time stamps and a *finger* (position) $t_{now}$ in the reuse trace $S_1$ that indicate where computation reuse from $S_1$ begins. $T_2$ corresponds to the output trace in the change-propagation judgement of Tgt and the input trace $S_1$ corresponds to the list of trace slices $\overline{S}$—here we leave the fragmentation into trace segments implicit—from which computation can be reused by memoization. In the actual implementation, $T_2$ is produced by editing $S_1$ in place, but here we distinguish the old and new trace for clarity.

**Propagation Queue.** The CPA algorithm maintains a *propagation queue* $Q$ of time stamps for inconsistent actions that must be replayed. The queue is ordered by time for the algorithm to efficiently find the next inconsistent computation (e.g., TDT query) that must be rerun.

Under in-order memoization, work from the previous run must be reused in execution order so the reuse trace $S_1$ is an integral trace—i.e., without fragmentation. Therefore the propagation queue only contains inconsistent queries with time stamps greater than $t_{now}$. In a simple implementation with only modrefs, the propagation queue contains time stamps for *every* inconsistent `get` action—i.e., for which the contents of the reference has changed with respect to the previous run. In a more sophisticated implementation with TDTs, the propagation queue *only* contains the time stamp of the next earliest inconsistent query (instead of all inconsistent queries) of each TDT instance—i.e., for which the result of the operation has changed. In the latter case, the propagation queue must include support for a TDT instance's time stamp to be updated according to the time stamp returned

by each invoke and revoke operation.

Under out-of-order memoization, the reuse trace $S_1$ may contain holes due to stolen work. Therefore the propagation queue contains entries that occur anywhere in the trace—i.e., with time stamps between $t_{start}$ and $t_{end}$—that correspond to inconsistent queries as well as holes where trace segments have been stolen. For out-of-order memoization with pure TDTs (e.g., immutable modrefs), the propagation queue contains time stamps for *every* inconsistent action. This identification is correct since the only way a `get` of a pure modref from the source language can become inconsistent (read a different value) is if the original `put` has changed. This relies on the fact that any reordering among the reads does not affect the values read and the initialization write always precedes the read actions. For mutable modrefs (and other effectful TDTs), reads and writes may be interleaved and reordering traces can swap the relative order between a read and write. Below we discuss how to extend CPA to support out-of-order memoization with effectful TDTs.

**Initial Call.** CPA is initially called with $\mathsf{CPA}(T_1, \varepsilon, Q_0, t_{start})$ where $T_1$ is the full (unfragmented) trace of the previous run and $\varepsilon$ is an empty trace to be built into the trace $T_2$ of the updated run. The propagation queue $Q_0$ is initialized to include all the TDT query actions that are immediately inconsistent due to input changes—e.g., `get` actions of input modrefs that have changed. The finger $t_{now}$ initially points to the beginning of the time line $t_{start}$.

**Reusing Work.** At each step, CPA uses the propagation queue to find the next inconsistent query $t_{inc}$ following $t_{now}$. If there are no more inconsistencies, then $t_{inc}$ is $t_{end}$ and the algorithm only needs to append the trace $T_r$—from the finger $t_{now}$ to the end of time $t_{end}$ in $S_1$—to the end of the accumulator trace $T_2$. If there is an inconsistent query at time $t_{inc}$, CPA extracts the trace segment $T_r$—from the finger $t_{now}$ to the inconsistency at $t_{inc}$ in the previous run $S_1$—that isn't affected by input changes and can thus be reused by simply appending it to the output trace. This corresponds to the $\curvearrowright$ **reuse** rules that fast-forward past the consistent actions that do not need to be replayed. Since the trace is a reified representation of the computation, this segment is removed from the input trace—because it can't be reused multiple times—and transferred to the output trace.

**New Work.** Change-propagation fast-forwards to the next inconsistent operation at $t_{inc}$ and reruns the continuation for that operation. For modrefs, the operation is a `get` that fetches a different value from the store compared to the previous run. More generally, for TDTs, the operation is a query that returns a different result.

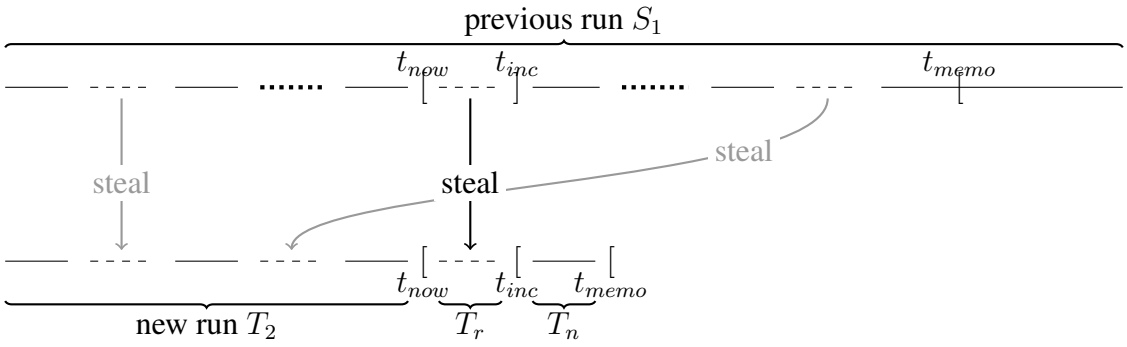



Figure 6.3: CPA constructing new trace from old trace.

While the continuation runs, it looks for a memo match in $S_1$ and stops when it finds one. While the continuation runs, whenever a change is made to a TDT that existed in the previous run (e.g., updating a modref with a different value), the inconsistent time stamps affected by that operation are added to $Q$'. For in-order memoization, the memo match time $t_{memo}$ must occur after $t_{now}$; therefore memo matching discards a prefix of the trace (from $t_{inc}$ to $t_{memo}$) by explicitly revoking the corresponding actions. For out-of-order memoization, $t_{memo}$ may occur before $t_{now}$ earlier in the trace—i.e., earlier than a memo match from an earlier iteration of CPA.

Thus when the fresh rerun $T_n$ is done, all inconsistent reads caused by the run are properly marked in $Q$'. Also while running all memoizing function calls are placed in the memo table for future reference. The rerun returns the time stamp $t_{memo}$ of the memo match, as well as the modified queue $Q$' and the new trace segment $T_n$ for the computation that has just run. Now CPA can extend the original output trace $T_2$ with the reused segment $T_r$ and the new segment $T_n$. Thus on every step (except perhaps the last), the algorithm adds one reused chunk of trace and one new chunk of trace to the output trace. Only the new chunks require work.

Figure 6.3 gives a pictorial representation of CPA constructing the time line $T_2$ for the updated run from chunks of the time line $S_1$ from the previous run. In one iteration of the algorithm, the trace chunk $T_r$ from the interval $[t_{now}, t_{inc})$ is stolen from the previous run, and the new trace chunk $T_n$ is executed in the interval $[t_{inc}, t_{memo})$, and reusable computation is found at $t_{memo}$ for the next iteration. Under out-of-order reuse, reused computation in $T_2$ may come from earlier than $t_{now}$ or later than $t_{inc}$.

**Runtime.** Now lets consider the runtime of CPA. Certainly all new computation needs to be run but this is accounted for in the trace distance. The other costs of the algorithm include the time for extracting and appending chunks of the trace, the cost for the queue operations, and the cost for memo lookup and associated insertion into the memo table. We use $T_{splice}$ to indicate the time to append or extract a chunk of trace for a trace of size $n$. Using balanced trees this can easily be implemented in $O(\log n)$ time, and with some work comparisons between time stamps in the trace can be made to work in $O(1)$ time. We use $T_{queue}(n)$ to indicate the time to insert or delete in the queue of size $n$. This is easy to implement in $O(\log n)$ time per operation as long as the comparison of time stamps is $O(1)$ time. We assume the memo lookup uses standard hash tables and therefore takes constant expected time per operation (either lookup or insertion). Consider a computation in which the total new computation in $c$, the total number of recursive calls of the CPA is $l$, the total trace distance just counting reads is $r$, and the maximum of the sizes of the input and output traces is $n$. The runtime is then $O(c + lT_{splice}(n) + (r + l)T_{queue}(n))$. Relating this to the trace distance measured by the semantics, change-propagation for two traces $S_1$ and $S_2$ such that $S_1 \ominus^{\gg} S_2 = \langle c_1, c_2 \rangle$ will run in time $O((c_1+c_2)(1+T_{splice}(n)+T_{queue}(n))) = O((c_1 + c_2) \log n)$.

**Out-of-Order Memoization with Effectful TDTs.** To make the CPA work with out-of-order memoization and effectful TDTs (e.g., mutable modrefs) requires some modification. The problem, as mentioned above, is that reordering of the operations on an mutable reference can change the value on any read even though no write has occurred. For example, suppose an initial run has a `set` with value $v_1$, a `set` with value $v_2$ followed by a `get` with value $v_2$. Next, suppose that in the updated run the `get` moves ahead in the trace and now is between both `set` actions, so that its value should be $v_1$. We call a move of a `get` that changes its value an *inversion*. Since we only update $Q$ on updates (e.g., `put` s and `set` s), and since we allow reordering, the queue will not contain all inconsistencies. To modify the algorithm we need to change the line that finds $t_{inc}$ so that it not only looks for the first element in $Q$ but also scans the trace from $t_{now}$ looking for any inversions. This can be implemented in time proportional to the number of queries (e.g., `get` actions) in that trace segment by simply keeping a linked list of queries.

### 6.5.3 Implementing Traceable Data Types

We describe how to implement the traceable version of various data types that conform to the invoke/revoke interface (such as `PRIORITY_QUEUE_TRACEABLE` from Section 2.3). A more complete description of the data types and how to implement others can be found

elsewhere [Acar et al., 2007a]. The basic idea behind the implementations is to keep an augmented version of the operation trace $T$. In particular, most of our TDTs maintain a data structure for the trace that is ordered by time stamps and supports insert$(T, v, t)$ (insert $v$ at time $t$), delete$(T, t)$ (delete the element at time $t$), findPrev$(T, t)$ (returns the greatest element in $T$ that is less that $t$) and findNext$(T, t)$. For a trace with $n$ entries, all these can be implemented in $O(\log n)$ time using balanced trees. Some of our traceable data types also maintain balanced trees ordered by keys (e.g., modular modifiables and priority queues).

**Modifiable References.** The traceable implementation of a modifiable reference maintains a time-ordered sequence of operations. Each operation is tagged with the value it has read or written. To invoke a $\mathbf{get}$ (read) or $\mathbf{put}$ (write) at time $t$, we insert the operation into the trace data structure at $t$. If the operation is a $\mathbf{get}$, then we also use findPrev$(T, t)$ to access the value returned by the read—the previous element in the trace might either be a $\mathbf{get}$ or $\mathbf{put}$, but both types of operation are stored with values. Note that a revision sequence requires that all operations before time $t$ are consistent; therefore, the value of this previous element contains the correct value for time $t$. To revoke a $\mathbf{get}$ or $\mathbf{put}$ at time $t$, we simply delete the operation from the trace. For all revisions (invokes or revokes) we can use findNext$(T, t)$ to return the earliest inconsistent operation, if any. In particular, if the next operation is a $\mathbf{get}$ and has a different value, then it is inconsistent and is returned, otherwise nothing is returned. All operations on a trace with $n$ elements take $O(\log n)$ time.

**Accumulator Modrefs.** An accumulator modifiable is implemented simply by "adding" to the sum using the commutative operator on an invoke and subtracting from the sum on a revoke. For any value other than the identity, this will return the next read as the earliest inconsistent operation.

**Modular Modrefs.** A modular modifiable is implemented by keeping all the boundary elements $c_i$ for all $\mathbf{mget}$ operations on a modular modifiable $m$ in a data structure $S_m$ sorted by their ordering. Invoking or revoking a $\mathbf{mget}$ operation on $m$ corresponds to inserting or deleting the partitioning elements from $S_m$. Changing the initial value will identify all partitions that are crossed by the change of value and return the earliest as inconsistent.

**Queues and Priority Queues.** The implementation of queues and priority queues is beyond the scope of this work. We note that a priority queue can be done with two balanced trees one ordered by time for the trace and the other by key. In addition, during an update sequence, the implementation maintains two additional balanced trees, one for insertions invoked during the current update sequence and the other for insertions revoked during the sequence. All operations take $O(\log n)$ time.

**Dictionaries.** The implementation of dictionaries is based on modifiables. Basically, we create a standard hash table, where each entry in the table is a modifiable with its own trace. The first time an operation is invoked on a particular key $k$, we create a new modifiable $m_k$ for that key with its own trace. Any insert of a key-value pair $(k, v)$ into the dictionary at time $t$ will correspond to a $\mathbf{put}$ of value $v$ into $m_k$ at $t$. Any delete of a key $k$ from the dictionary at time $t$ will correspond to a $\mathbf{put}$ of value $\mathbf{NONE}$ into $m_k$ at $t$, where $\mathbf{NONE}$ is a special value indicating that the dictionary has no entry at that key. Any search of a key $k$ at time $t$ corresponds to a $\mathbf{get}$ $m_k$ at $t$. Finally, if a revoke of an operation on key $k$ removes the last operation from the trace of $m_k$, then we can delete $m_k$ from the dictionary (this avoids a memory leak).

### 6.5.4 Integrating Traceable Data Types

Invoking and revoking TDT operations in the semantics uses trace reparation as a uniform mechanism to find and mark the next inconsistent operation. Each TDT is implemented as a standalone Standard ML module that maintains a local time line of operations on that instance. The implementation of a TDT operation $\mathbf{op} : \tau_{arg} \rightarrow \tau_{res}$ must provide functions

```
invoke_op : ts * τ_arg -> τ_res * ts option
revoke_op : ts -> ts~option
```

where `invoke_op` takes an argument value and the time stamp at which the operation is invoked and returns the result and the time stamp of the next inconsistent operation, and `revoke_op` takes the time stamp at which the operation was invoked and returns the time stamp of the next inconsistent operation. Each TDT supports efficient change-propagation by taking advantage of problem-specific structure so that each invoke or revoke operation can quickly identify the next inconsistent operation in the trace without having to perform a linear traversal of the trace.

In order to integrate the operation into the change-propagation implementation, we generate boilerplate code to invoke and revoke the operation and maintain an entry in the

propagation queue with the earliest inconsistent operation. For each invoke operation, the boilerplate code creates a new time stamp for the operation invokes the TDT operation, and passes the result to the continuation. Moreover, it sets suitable hooks up in the time line for the operation to be reinvoked if the action needs to be re-executed (i.e., if change-propagation re-runs it) and for the operation to be revoked (i.e., when memoization discards it). Therefore, if the change-propagation algorithm deletes a time stamp, we revoke the operation that is associated with that time stamp.

During change-propagation, trace elements that need re-evaluation are stored in the propagation queue prioritized by their time stamps, including the inconsistent operations of all TDTs. Since the set of inconsistent operations dynamically changes over time as a result of invokes and revokes, we adjust the priority queue dynamically to maintain the correct set of inconsistent operations.

# Chapter 7

# Evaluation

This chapter is based on the experimental evaluation of the ΔML compiler with modifiable references [Ley-Wild et al., 2008b] and traceable data types [Acar et al., 2010a], and formal analysis of self-adjusting computation [Ley-Wild et al., 2009].

## 7.1 Overview

We describe an experimental evaluation of the ΔML compiler with modifiable references (Section 7.2) and modref-based data structures against traceable data types (Section 7.3). The experiments for modref applications indicate that 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.

Furthermore, we use a set of diverse benchmarks to compare the space usage and time performance of programs using TDTs to that of programs using standard, modref-based implementations. The results show that traceable data structures significantly help improve speed and reduce memory consumption. To understand the source of this performance improvement, we study how tracking at the granularity of data type operations affects the trace size and stability. Our findings suggest that tracking operations on data structures helps reduce the trace size and improve stability by asymptotic factors.

Finally, we use the cost semantics to formally reason about the efficiency of change-propagation with respect to the semantics with in-order reuse (Section 7.4) and out-of-order reuse (Section 7.5).

## 7.2 △ML with Modifiable References

### 7.2.1 Synthetic 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.

### 7.2.2 Input Generation

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 $10n$.

### 7.2.3 Measurements

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:

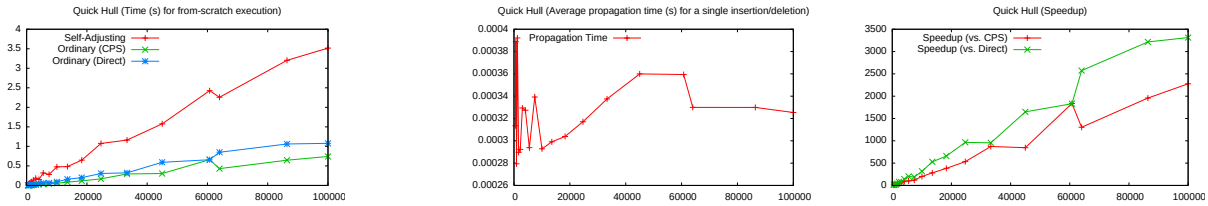
- **Time for from-scratch execution:** The time for the from-scratch execution of the ordinary or self-adjusting version.
- **Average propagation time for a single insertion/deletion:** For each element in the input list, we delete the element, run change-propagation, insert the element (at the same 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.

**Technical Setup.** 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.

### 7.2.4 Results

Table 7.1 gives summaries for the benchmarks at fixed input sizes. Each columns show the measurements for the quantities described above. The overheads range between 3 and 33. The integer benchmarks have higher overheads because the perform trivial work between




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

Figure 7.1: 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 7.1: Summary of benchmark timings.

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 7.1(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 7.1(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 7.1(c) shows the average speedup, which increases linearly with the input size to exceed three orders of magnitude.

### 7.2.5 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.

For experiments, we render an input scene (shown on the right) of 3 light sources and 19 objects with an output image size of $512 \times 512$ 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 and 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.

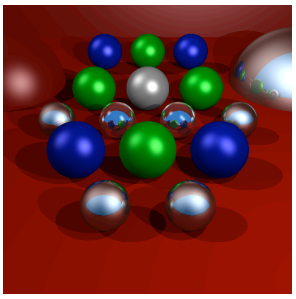


Figure 7.2: Ray-tracer output.

Table 7.2 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.

## 7.3 $\Delta$ML with Traceable Data Types

### 7.3.1 Benchmarks

We developed a set of benchmark to study the performance characteristics of the $\Delta$ML with traceable data types. Each benchmark is specified by a static algorithm's description. Based on this description, we implemented three versions: (1) a static program ("static"), (2) a self-adjusting program that does not utilize TDTs ("modref-based"), and (3) a self-adjusting program that makes use of TDTs whenever appropriate ("traceable"). In developing the test suite, we first implemented the static program and transformed it into a self-adjusting program by inserting modrefs. The traceable version is identical to the modref-based version, except the traceable version makes calls to traceable data structures whereas the modref-based version makes calls to modref-based implementations of data structures. In Table 7.3.1, we summarize the data types used in each benchmark.

- **heapsort** (hsort-int): sort an integer list using the standard heapsort algorithm.
- **Dot product** (dot-product): compute the dot product of two real-number vectors represented as a list of ordered pairs, by first computing the product for each component and using an accumulator to compute the sum.
- **List intersection** (intersection): compute the intersection of lists $\ell_1$ and $\ell_2$, by inserting the elements of $\ell_1$ into a dictionary and selecting the elements of $\ell_2$ that are present in the dictionary.
- **Huffman code** (huffman): construct a Huffman tree for a list of keys and frequencies using the standard Huffman algorithm.
- **Interval stabbing** (stabbing): take as input a list of intervals $I = \{[a_i, b_i)\}_{i=1}^{n}$ and a list of queries $Q = \{q_j\}_{j=1}^{m}$, and report for each query $q_j$ how many intervals this query "stabs" (i.e., the size of the set $\{(a_i, b_i) \in I : a_i \leq q_j < b_i\}$). We present a plane-sweep algorithm: First, insert into a priority queue the endpoints of all the intervals and the query values, known as events, and set initialize a counter $c$ to $0$. Then, to answer queries, consider the events in an increasing order of their values, incrementing the counter on a left endpoint, decrementing it on a right endpoint, and outputting the counter value on a query.
- **Graham Scan** (graham-scan): compute the convex hull of a set of points in 2D using the Graham's scan algorithm (more in Section 7.3.8).

- **Dijkstra** (dijkstra): compute the shortest-path distances in a weighted graph from a specified source node using Dijkstra's algorithm and output a dictionary mapping each node to its distance to the source.

- **Breadth-First Search** (bfs): perform a breadth-first search, which computes the shortest paths in an unweighted graph from a specified source node and outputs a dictionary mapping each node to its distance to the source.

### 7.3.2 Modref-based Data Structures

We implemented modref-based data structures for every data type used in the benchmarks. These implementations may not be the best one can obtain using modifiables alone, but they are reasonable baselines because we believe they are representative of what a programmer with significant background in self-adjusting computation would produce after some optimization. The accumulator data structure is implemented by maintaining a modifiable list and running a self-adjusting `fold` operation to obtain the solution. Both the dictionary and priority queue data structures are implemented using the treap data structure. For priority queues, we found that treaps are more stable than common alternatives (e.g., leftist heaps or binary heaps). The queue data structure is obtained by essentially transforming a standard purely functional implementation of a queue that maintains two lists; however, we are especially careful about when the front list is reserved to enhance stability.

### 7.3.3 Input Generation

We use randomly generated data sets for all the experiments. Let $n$ be the target input size. For the sorting benchmarks, we generate a random permutation of the integers from 1 to $n$. For dot-product, we generate random vectors by picking floating-point numbers uniformly at random from $[0.0, 10.0]$ (with 5 significant digits). For intersection, We generate a pair of lists of lengths $n$ and $m$ by picking integers uniformly at random from the set $\{0, \ldots, t\}$, where $t = \frac{1}{4}\min\{n, m\}$; this choice of $t$ ensures that the two lists have a common element with high probability. For huffman, the alphabets are simply the numbers 1 to $n$, and the frequencies are random integers drawn from the range $[1, 10n]$. For stabbing, the endpoints and query values are random numbers in the range $[0, n/10]$ chosen uniformly at random. For convex hulls, we generate inputs by drawing points uniformly from the circumference of a unit-radius circle. This arrangement is known to be a challenging pattern for many convex-hull algorithms. For our graph benchmarks,

we generate random, connected graphs with approximately $\sqrt{n}$-separators, mimicking the fact that many real-world graphs have small separators (e.g., $n^{1-\varepsilon}$).

### 7.3.4 Metrics and Measurements

The metrics for this study are (1) the time to run a program from scratch, denoted by $T_i$ (2) the average update time after a modification, denoted by $T_u$, and (3) the space consumption, denoted by $S$. To measure the second metric, for example, in list-based experiments, we apply a delete-propagate-insert-propagate step to each element (i.e., in each step, delete an element, run change-propagation, insert the element back, and run change-propagation) and divide the end-to-end time by $2n$, where $n$ is the list's length. This quantity represents the expected running time of change-propagation if a random update to the input is performed. We can use this measurement in graph experiments, where here the delete-propagate-insert-propagate is applied to each edge in turn.

**Technical Setup.** Our experiments were conducted on a $2.0$Ghz Intel Xeon E5405 with 32 GB of memory running Ubuntu 8.04 (kernel 2.6.24-19). Programs were compiled using the $\Delta$ML compiler with the options "`-runtime ram-slop 0.9 gc-summary`", which direct the runtime system to make 90% of the physical memory available to the benchmark and report garbage collection (GC) statistics.

We measure the space consumption by noting the maximum amount of live data as reported by $\Delta$ML's garbage collector. This is an approximation of the actual space usage because garbage collection may miss the high-water mark.

When measuring time, we carefully break down the execution time into application time and garbage collection (GC) time. In these experiments, we have found that GC is at most $20\%$ of the execution time. For this reason, we only report the application time to isolate the GC effects and highlight the asymptotic performance.

| Image Size $512 \times 512$ | | 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 7.2: Summary of raytracer timings.

| **Benchmark** | **Data Types Used** |
|---|---|
| hsort-int | priority queue |
| dot-product | accumulator |
| intersection | dictionary |
| huffman | priority queue |
| stabbing | priority queue, counter |
| graham-scan | priority queue |
| dijkstra | priority queue, dictionary |
| bfs | queue, dictionary |
| Motion Simulation | modular modref |

Table 7.3: Summary of data types used in our benchmarks. Every self-adjusting program also uses the modref data type.

| Experiment | Size | Traceable | | | Modref-based | | | Modref-based ÷ Traceable | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $N$ | $T_i$ (ms) | $T_u$ ($\mu$s) | $S$ (MB) | $T_i$ (ms) | $T_u$ ($\mu$s) | $S$ (MB) | $T_i$ | $T_u$ | $S$ |
| hsort-int | $10^3$ | 7.50 | 35.00 | 0.61 | 85.00 | 27695.00 | 14.04 | 11.33 | 791.28 | 23.02 |
| dot-product | $10^5$ | 280.00 | 6.75 | 52.88 | 872.50 | 121.55 | 223.80 | 3.11 | 18.00 | 4.23 |
| intersection | $10^5$ | 1372.50 | 82.00 | 382.78 | 11207.50 | 1948.45 | 1509.17 | 8.16 | 23.53 | 3.94 |
| huffman | $10^4$ | 157.50 | 492.00 | 22.13 | 2575.00 | 2530000.00 | 707.61 | 16.34 | 5142.28 | 31.98 |
| stabbing | $10^3$ | 17.50 | 115.00 | 1.92 | 240.00 | 98195.00 | 23.56 | 13.71 | 853.87 | 12.27 |
| graham-scan | $10^4$ | 375.00 | 265.50 | 24.90 | 1542.50 | 1105.50 | 277.24 | 4.11 | 4.16 | 11.13 |
| bfs | $10^3$ | 37.50 | 845.56 | 2.74 | 717.50 | 23784.07 | 139.39 | 19.13 | 28.12 | 50.82 |
| dijkstra | $10^3$ | 42.50 | 1160.03 | 2.74 | 725.00 | 34528.30 | 72.41 | 17.05 | 29.76 | 26.42 |

Table 7.4: Traceable vs. modref-based implementations: $T_i$ (in ms) is the from-scratch execution time, $T_u$ (in $\mu s$) is the average time per update, and $S$ (in MB) is the maximum space usage as measured at garbage collection.



| Experiment | Size | Traceable | | | Static | Overhead | Speedup |
|---|---|---|---|---|---|---|---|
| | $N$ | $T_i$ (ms) | $T_u$ ($\mu$s) | $S$ (GB) | $T_i$ (ms) | (SAC $T_i$)/(static $T_i$) | (static $T_i$)/(SAC $T_u$) |
| hsort-int | $10^6$ | 14390.00 | 59.02 | 1.75 | 2599.75 | 5.5 | $4.4 \times 10^4$ |
| dot-product | $10^6$ | 2787.50 | 7.45 | 0.44 | 100.25 | 27.80 | $1.3 \times 10^4$ |
| intersection | $10^6$ | 12820.00 | 74.91 | 2.19 | 1091.50 | 11.74 | $1.5 \times 10^4$ |
| huffman | $10^6$ | 22975.00 | 1021.04 | 1.08 | 6447.25 | 3.56 | $6.3 \times 10^3$ |
| stabbing | $10^6$ | 38832.50 | 202.11 | 1.70 | 10609.75 | 3.60 | $5.2 \times 10^4$ |
| graham-scan | $10^5$ | 4307.50 | 297.30 | 0.70 | 547.75 | 7.86 | $1.8 \times 10^3$ |
| bfs | $10^4$ | 445.00 | 1310.59 | 0.12 | 47.50 | 9.36 | 36.2 |
| dijkstra | $10^4$ | 490.00 | 1783.68 | 0.12 | 52.50 | 9.33 | 29.4 |

Table 7.5: Traceable SAC versus static: $T_i$ (in ms) is the from-scratch execution time, and $T_u$ (in $\mu$s) is the average time per update.

### 7.3.5 Modref-based Programs vs. Traceable Programs

The first set of experiments studies how TDTs provide the performance benefits over traditional, modref-based implementations. Recall that $T_i$ is the time to run a program from scratch and $T_u$ is the average time that change propagation takes to perform an update. Table 7.4 shows the performance of our benchmark programs, comparing the traceable versions to their modref-based counterparts. Note that for graham-scan, the modref-based program uses merge sort whereas the traceable program uses heapsort; the modref-based version of heap sort is too slow except for extremely small inputs. We explore this in more detail in Sections 7.3.8 and 7.3.9.

We find that compared to the modref-based programs, the traceable versions are $3$–$20$ times faster to run from scratch and $4$–$5000$ times faster to perform an update. Moreover, traceable versions consume $4$–$50$ times less space than the modref-based ones. We remark that these experiments involve relatively small input sizes because with larger inputs our experiments with some modref-based applications require too much time to complete.

### 7.3.6 Traceable Programs vs. Static Programs

Our second set of experiments, shown in Table 7.5, draws a comparison between traceable programs and static programs, quantifying the effectiveness of the approach in more absolute terms. First, consider the overhead column, calculated as the ratio of the from-scratch run of the traceable implementation to that of the static implementation. This quantity represents the overhead due to dependence tracking and the runtime system. We find the overhead to relatively small: the traceable versions are about a factor of $10$ slower than their static counterparts, except for dot-product, which is about $30$ times slower. We believe this is because the benchmark dot-product is relatively lightweight computationally.

Second, consider the speedup column, calculated as the ratio of the static from-scratch run time to the update time. Results show that the traceable versions can perform updates many orders of magnitude faster. One exception is our graph algorithms, which are output-sensitive and may need to update the results at many nodes even after a small modification, e.g., deleting a single edge can change the shortest distance of many nodes. We discuss this in greater depth next.

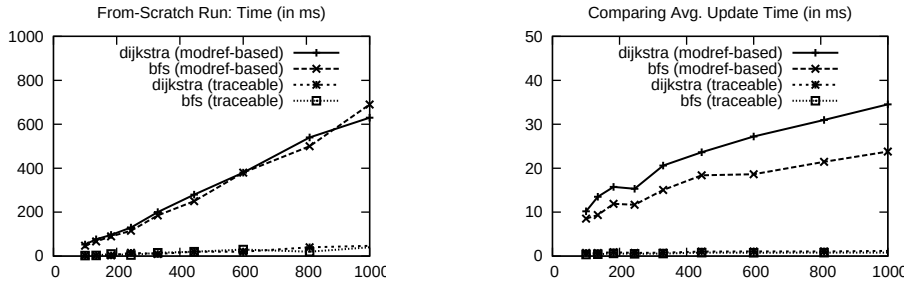



Figure 7.3: Measurements for from-scratch runs (left) and updates (right) with our graph benchmarks; timing (vertical axis in ms) as input size (horizonal axis) varies.

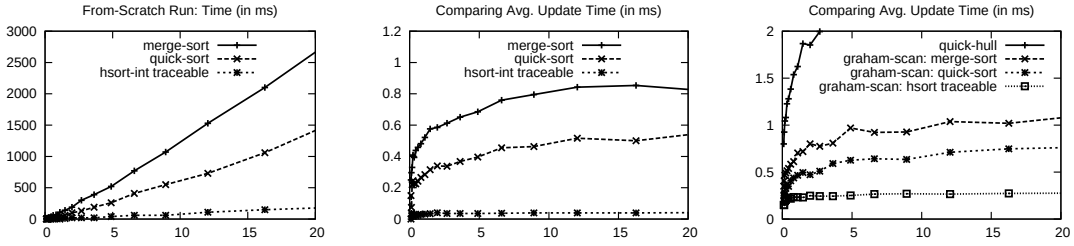



Figure 7.4: Detailed measurements for the sorting and graham-scan experiments: timing (vertical axis in ms) as input size (horizontal axis in thousands of elements) is varied.

### 7.3.7 Graph Algorithms

Self-adjusting computation with modrefs works well for problems with structured data (e.g., lists and trees). Computations involving unstructured data (e.g., graphs), however, often require dynamic data structures whose traditional self-adjusting versions can require the tracking and updating of large amount of dependencies. Traceable data types address this problem by reducing the amount of required tracking and exploiting problem-specific structures, thereby dramatically decreasing the update time.

We consider two algorithms: the Dijkstra's single-source shortest path algorithm (dijkstra) and the classic breath-first-search algorithm (bfs). Our implementations follow the standard textbook descriptions (Figure 2.7 shows the pseudo-code for dijkstra). Both algorithms use a dictionary to represent a graph.

Figure 7.3 contrasts the performance of traceable versions of shortest-path algorithms with that of the traditional, modref-based versions. The left side shows from-scratch execution times of dijkstra and bfs. Both perform similarly, and their traceable versions are

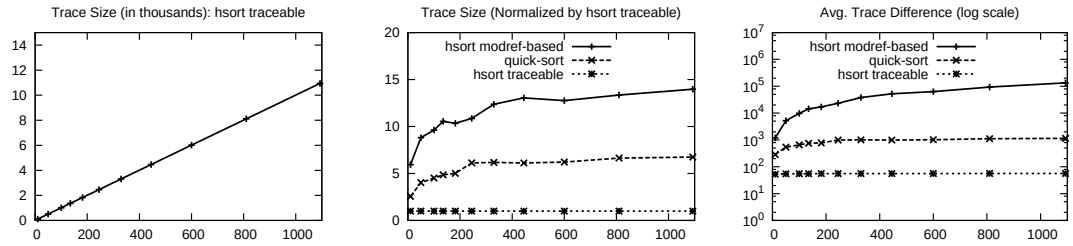



Figure 7.5: Trace size (in thousands of trace elements) and average trace difference (in trace elements on a log scale) of sorting benchmarks as input size is varied: trace size of traceable heapsort (left), trace size of quicksort and modref-based heapsort as normalized by the trace size of traceable heapsort (center), and average trace difference (right).

significantly faster than their traditional versions, by more than an order of magnitude at peak. The right side shows the average update times for an edge deletion/insertion. Again, both benchmarks perform similarly and the traceable versions are significantly faster than the traditional, by approximately an order of magnitude at $N = 1,000$.

We note that both dijkstra and bfs are highly output sensitive algorithms. Since inserting/deleting an edge can change the shortest-path distances on a large number of nodes, these benchmarks are highly output sensitive. Specifically, if the shortest-path distances change on $t$ nodes, both benchmarks will need to update all $t$ nodes, requiring at least $\Omega(t)$ time.

### 7.3.8 Sorting and Convex Hulls

Another noteworthy feature of the TDT framework is modularity, specifically the fact that we can often enjoy substantial performance improvements by simply replacing the modref-based implementations of data structures with the compatible traceable versions. As an example, consider the problem of computing the convex hull of 2D data points. Given a set of 2D points, Graham's scan algorithm first orders the points by the $x$ coordinates and computes the convex hull by scanning the sorted points. Here we compare the traceable version of heapsort (hsort) and graham-scan benchmarks against other modref-based algorithms. The fastest version turns out to be identical to the old graham-scan code, except the sort routine is now a traceable heapsort.

As shown in Figure 7.4 (left and center plots), traceable heapsort outperforms the quick-sort and merge-sort algorithms by nearly an order of magnitude for both from-scratch runs and updates. Since graham-scan uses sorting as a substep, it shows the

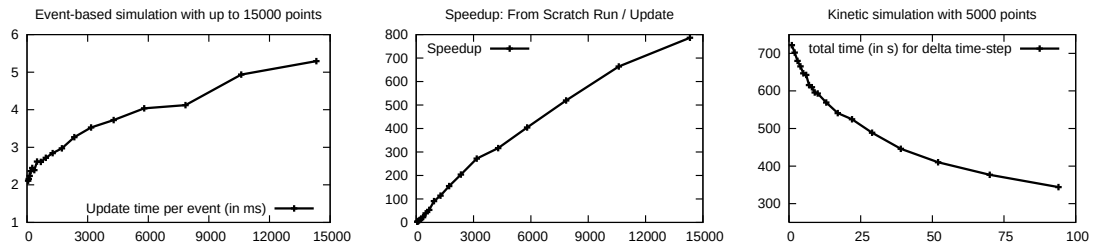



Figure 7.6: Time per kinetic event (left), speedup for an update (center), and total simulation time (seconds) with time-slicing (right).

same performance trends (rightmost plot). Compared to the modref-based implementation of the quick-hull algorithm [Barber et al., 1996], graham-scan is extremely fast.

### 7.3.9 Trace Size and Stability

Our empirical measurements thus far illustrate the performance benefits of TDTs, both in running time and space consumption. Here we investigate the question of whether these improvements are related to potential constant factor improvements in the run-time systems or to the benefits of TDTs as we expect them to be. Our measurements suggest the latter and indicate asymptotic improvements in performance. To this end we consider two abstract measurements: trace size and trace stability. Trace size measures the size of the memory consumed. Trace stability measures how much the trace changes as a result of an input modification—this ultimately determines how fast the program can respond to modifications. In our experiments, we measure the trace size by the number of trace elements, and the trace stability by counting the average number of trace elements created and deleted during change propagation after a single insertion/deletion. These measures are independent of the specifics of the hardware as well as the specifics of the data structures used for change propagation—they only depend on the abstract representation of the trace. They are, however, specific to the particular self-adjusting program and the class of input changes considered. As an example, we consider here sorting with integers, specifically hsort-int, with traceable and modref-based priority queues, and a self-adjusting implementation of quicksort.

Figure 7.5 (leftmost) shows the trace size for traceable heapsort as the input size increases. Regression analysis shows a perfect fit with $10n + 12$ ($n$ is the input size), providing strong evidence that the trace size of traceable heapsort is $O(n)$. This is consistent with a simple analytical reasoning on traces: since we record dependencies at the level of

priority-queue operations and since heap-sort performs linear number of such operations, the trace has linear size.

Figure 7.5 (center) shows the trace size of hsort-int using both traceable and modref-based priority queues normalized to the trace size of the traceable heapsort. The figure suggests that the traces of traceable heapsort are by a factor of $\Theta(\log n)$ smaller than those of the modref-based. This explains why traceable heapsort has a significantly smaller memory footprint than the modref-based counterpart.

Figure 7.5 (right) shows our measurements of average trace difference on a vertical log scale for a single insertion/deletion. Trace difference is constant for traceable heapsort, because a single insertion/deletion requires inserting/deleting a single priority queue operation from the trace. The modref-based implementation heapsort appears to have super-logarithmically larger trace difference. The reason for this is the internal comparisons traced by the modref-based priority queues. This finding explains the difference in the runtime performance between the two implementations of heapsort.

Figure 7.5 also compares the traceable heapsort to our self-adjusting quicksort implementation, which, until now, has been the most efficient self-adjusting sorter. Traceable heapsort appears faster by at least a moderate constant factor.

## 7.4 Cost Semantics with In-Order Memoization

We consider several examples to show how trace distance can be used to analyze the sensitivity of programs to small changes in their input. We say that a program is $O(f(n))$-*sensitive* or $O(f(n))$-*stable* for an input change if the distance between the traces of that program is $O(f(n))$ for inputs related by that change. In our analysis, we consider two kinds of changes: insertions/deletions that relate lists that differ by the existence of an element (e.g., `[1,3]` and `[1,2,3]`) and replacements that relate inputs that differ by the value of one element (e.g., `[1,2,3]` and `[1,7,3]`). When convenient, we visualize traces as derivations and analyze their relative distance under a replacement.

In our analysis, we consider two kinds of bounds: upper bounds and lower bounds. Our upper bounds state that the distance between the traces of a program with inputs related by some change can be asymptotically bounded by some function of the input size under the assumption that locations allocated in the computation (or mentioned in the trace) can be chosen to match nicely. Without the ability to match locations, it is not possible to prove interesting upper bounds, because two runs of the program can differ by as much as the size of the traces if their locations are chosen from disjoint sets. An implementation can

often match locations, sometimes with programmer guidance. Our lower bounds state that the distance between traces of a program with inputs related by some change cannot be asymptotically smaller than a function of input size regardless of how we choose locations. Such lower bounds suggest but do not prove a lower bound on the running time for change-propagation.

Our analyses fit into one of the following patterns. Sometimes, we start with two concrete inputs and show a bound on the distance between traces with these inputs. We then generalize this bound to arbitrary inputs using the identity and substitution theorems (Theorems 20 and 22). Other times, using the identity and the substitution theorems, we write a recursive relation for the distance between the traces of the program with inputs related by some change, and solve this relation to establish the bound. When analyzing our examples and using the identity and the substitution theorems, we ignore contexts, because, as noted in Chapter 3, they are not needed for analysis. We use the distance and the composition theorems in the informal style of traditional algorithmic analysis, because we have no meta-logical framework for reasoning about asymptotic properties of self-adjusting programs (Section 8.2).

Figure 7.7 shows the code for map, list reduction, and merge sort. The list reduce and merge sort implementations use several helper functions, whose code we omit for brevity. The `lenLT $ (l,i)` function returns a modref containing `true` iff the length of the list `l` is less than the integer `i`. The `partition` function evenly splits a list into two and `merge` combines two sorted lists. All of these functions are $O(1)$-sensitive to replacements on average (for `merge`, we need to average over all permutations of the input to obtain this bound). To focus on the main ideas, we omit the analysis of these utility functions here, which are similar to that of the `map` function discussed below.

```
  datatype 'a cell = nil | ::  of 'a * 'a list
  withtype 'a list = 'a cell modref

  mfun mapA l =
    case get $ l of
      nil => put $ nil
    | h::t =>
      let val t' = mapA $ t
      in put $ ((i2c h)::t') end


afun reduce f id l =
let mfun red r l =
      case get $ l of
        nil => put $ r
      | h::t => red (f(h,r)) $ t
in red id $ l end

afun reducePair f id l =
let mfun comp l =
      case get $ l of
        nil => put $ nil
      | a::t => case get $ t of
                  nil => put $ (a::put $ nil)
                | b::u => put $ (f(a,b)::(comp $ u))
    mfun rec l =
      if get $ (lenLT $ (l,2))
      then
        case get $ l of
          nil => id
        | h::_ => h
      else rec $ (comp $ l)
in rec l end

mfun msort l =
  if get $ (lenLT $ (l,2)) then l
  else let val (a,b) = partition l
           val sa = msort a
           val sb = msort b
       in merge (sa,sb) end

mfun filter f l =
  case get $ l of
    nil => put $ nil
  | h::t => if (f h) then h::(filter f $ t)
            else filter f $ t
```

Figure 7.7: Code for the examples.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{T_0 \ominus T_0 = 0 \qquad \ell_b \overset{\texttt{put}}{\leftarrow} \texttt{b}{::}\ell_c{\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_b \boxminus \ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_b = \langle 2,1\rangle}{\texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_b \overset{\texttt{put}}{\leftarrow} \texttt{b}{::}\ell_c{\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_b \boxminus \texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_c = \langle 3,2\rangle}}{\ell_2 \overset{\texttt{get}}{\rightarrow} 2{::}\ell_3{\cdot}2 \Downarrow \texttt{b}{\cdot}\texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_b \overset{\texttt{put}}{\leftarrow} \texttt{b}{::}\ell_c{\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_b \boxminus \texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_c = \langle 5,2\rangle}}{\texttt{M}^{\ell_2 \Downarrow \ell_b}(\ell_2 \overset{\texttt{get}}{\rightarrow} 2{::}\ell_3{\cdot}2 \Downarrow \texttt{b}{\cdot}\texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_b \overset{\texttt{put}}{\leftarrow} \texttt{b}{::}\ell_c){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_b \boxminus \texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_c = \langle 7,3\rangle}}{\ell_1 \overset{\texttt{get}}{\rightarrow} 1{::}\ell_2{\cdot}1 \Downarrow \texttt{a}{\cdot}\texttt{M}^{\ell_2 \Downarrow \ell_b}(\ell_2 \overset{\texttt{get}}{\rightarrow} 2{::}\ell_3{\cdot}2 \Downarrow \texttt{b}{\cdot}\texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_b \overset{\texttt{put}}{\leftarrow} \texttt{b}{::}\ell_c){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_b \ominus \ell_1 \overset{\texttt{get}}{\rightarrow} 1{::}\ell_3{\cdot}1 \Downarrow \texttt{a}{\cdot}\texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_c = \langle 8,4\rangle}}{\texttt{M}^{\ell_1 \Downarrow \ell_a}(\ell_1 \overset{\texttt{get}}{\rightarrow} 1{::}\ell_2{\cdot}1 \Downarrow \texttt{a}{\cdot}\texttt{M}^{\ell_2 \Downarrow \ell_b}(\ell_2 \overset{\texttt{get}}{\rightarrow} 2{::}\ell_3{\cdot}2 \Downarrow \texttt{b}{\cdot}\texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_b \overset{\texttt{put}}{\leftarrow} \texttt{b}{::}\ell_c){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_b) \boxminus \texttt{M}^{\ell_1 \Downarrow \ell_a}(\ell_1 \overset{\texttt{get}}{\rightarrow} 1{::}\ell_3{\cdot}1 \Downarrow \texttt{a}{\cdot}\texttt{M}^{\ell_3 \Downarrow \ell_c}(T_0){\cdot}\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{a}{::}\ell_c) = \langle 9,5\rangle}$$

Figure 7.8: Trace distance between `mapA $ [1,2,3]` and `mapA $ [1,3]`.

### 7.4.1 Map

We analyze the sensitivity of the `mapA`—for mapping a list of integers to a list of characters—to an insertion/deletion more precisely by using trace distance. Figure 7.8 shows the derivation of the trace distance for `mapA` with inputs `[1,2,3]` and `[1,3]`. We consider derivations where the input locations are $\ell_1, \ell_2, \ell_3, \ell_4$ and the output locations are $\ell_a, \ell_b, \ell_c, \ell_n$. In the derivations we use the notation $\mathtt{M}^{\ell \Downarrow \ell'}(T)$ as a shorthand for the memoization action $\mathtt{M}^{\mathtt{mapA\$}\ell \Downarrow \ell'}(T)$. Similarly we write $\mathtt{x} \Downarrow \mathtt{y}$ as a shorthand for the memoization action $\mathtt{M}^{\mathtt{i2c\$}x \Downarrow y}(\_)$ of the function `i2c` mapping integer $x$ to letter $y$, whose subtrace (body) we leave unspecified and assume to be of length constant (it contributes one to the distance). We define the tail trace $T_0$ common to both executions as:

$$\ell_3 \overset{\mathtt{get}}{\rightarrow} 3 :: \ell_4 \cdot 3 \Downarrow \mathtt{c} \cdot \mathtt{M}^{\ell_4 \Downarrow \ell_n}(\ell_4 \overset{\mathtt{get}}{\rightarrow} \mathtt{nil} \cdot \ell_n \overset{\mathtt{put}}{\leftarrow} \mathtt{nil}) \cdot \ell_c \overset{\mathtt{put}}{\leftarrow} c :: \ell_d.$$

When deriving the distance, we combine consecutive applications of the same rule and use the fact that the synchronization distance between a trace and itself is $\langle 0, 0 \rangle$.

Having derived a constant bound for this example, we can generalize the result to obtain an asymptotic bound for a change in one element in the middle of an arbitrary list. Consider the traces $T_1$ and $T_2$ for `mapA $` $L_1$ and `mapA $` $L_2$ where $L_1$ is `[`$x$`]` and $L_2$ is `nil`. The distance between them is trivially constant for any $x$. We will now use the substitution theorem to generalize this result to arbitrary lists by showing how to extend the inputs lists with identical prefixes and suffixes without affecting the constant bound.

We consider extending the input with the same suffix. We start by replacing each of the sub-traces of the form $\mathtt{M}^{\text{-}\Downarrow\text{-}}(\_)$ for the rightmost call to `mapA` in $T_1$ and $T_2$ with a hole to obtain the trace contexts $\mathscr{T}_1$ and $\mathscr{T}_2$. Let $L_3$ be any list and let $T_3$ be the trace for `mapA $` $L_3$. Note that the traces $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$ are the traces for `mapA $ (`$L_1$`@`$L_3$`)` and `mapA $ (`$L_2$`@`$L_3$`)`. By the identity theorem, the distance between $T_3$ and itself is $\langle 0, 0 \rangle$. Since $T_3$ starts with memoization action of the form $\mathtt{M}^{\ell_i \Downarrow \ell_\alpha}(\ldots)$, we can apply the substitution theorem, so the distance between $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$ is equal to the distance between $\mathscr{T}_1[\mathtt{M}^{\ell_i \Downarrow \ell_\alpha}(\square)]$ and $\mathscr{T}_3[\mathtt{M}^{\ell_i \Downarrow \ell_\alpha}(\square)]$, which is constant. Thus, we are able to append any suffix to $L_1$ and $L_2$ without increasing their distance.

Symmetrically, we can extend these lists with the same prefix. To see this, let $L_0$ be a list and consider its trace $T_0$ with `mapA`. Now define the trace context $\mathscr{T}_0$ as the context obtained by replacing the rightmost sub-trace in $T_0$ of the form $\mathtt{M}^{\text{-}\Downarrow\text{-}}(\_)$ with a hole. Now, substitute into this trace the traces $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$ (i.e., $\mathscr{T}_0[\mathscr{T}_1[T_3]]$ and $\mathscr{T}_0[\mathscr{T}_2[T_3]]$). By the identity and the substitution theorems, the distance is equal to distance between of $\mathscr{T}_1[T_3]$ and $\mathscr{T}_2[T_3]$, which is constant.

Thus, we can generalize concrete examples to other lists by prepending and appending arbitrary lists, essentially obtaining any two lists related by an insertion/deletion. We conclude that `mapA` is constant sensitive for an insertion into/deletion from its input.

### 7.4.2 Reduce

The reduce function reduces a list to a value by applying a given binary operator with a specified identity element to the elements of the list. The standard accumulator-based implementation,

```
reduce:  ('a * 'a -> 'a) -> 'a -> 'a list -> 'a modref
```

shown in Figure 7.7, is not amenable to self-adjusting computation, because the distance can be as large as linear. To see this, note that all intermediate updates of the accumulator depend on the previously-seen elements. Thus replacing the first element will prevent all derivation steps from matching, causing the distance to be linear in the size of the input (in the worst case).

Figure 7.7 shows another implementation for list-reduce, called `reducePair`. This implementation applies the function `comp` repeatedly until the list is reduced to contain at most one element. Function `comp` pairs the elements of the input list from left to right and applies `f` to each pair reducing the size of the input list by half. Thus, `comp` is called a logarithmic number of times. Using the shorthand $\texttt{chk}\$\ell \Downarrow v$ for derivations of the form $\texttt{lenLT}\$\ell \Downarrow b{\cdot}b \overset{\texttt{get}}{\rightarrow} v$, the derivations for `reducePair` can be represented with the following derivation context.

$$\dfrac{\texttt{chk}\$\ell \Downarrow \texttt{false} \qquad \overset{\texttt{comp}\$\ell\Downarrow\ell_1}{\bigtriangledown} \qquad \overset{\texttt{rec}\$\ell_1\Downarrow r_1}{\bigtriangledown}}{\dfrac{\texttt{rec}\$\ell \Downarrow r_1}{\texttt{reducePair}\$(f, id, \ell) \Downarrow r_1}}$$

To analyze the sensitivity of `reducePair` for a replacement operation, consider evaluating `reducePair` with two lists related by a replacement. The recursive case for the derivations both fit the derivation context given above. Note that the derivations for `comp` are related by a replacement. Since a replacement in the input causes the output of `comp` to change by a replacement as well, the recursive calls to `rec` are related by a replacement as well. Furthermore, since the derivation for `comp` and `rec` both start with memoizing functions, we can apply the substitution theorem assuming that the `comp` returns its output

in the same location. More precisely, we can write the sensitivity of `rec` to a replacement for an input size of $n$ as:

$$\Delta_{\texttt{rec}}(n) = \begin{cases} \Delta_{\texttt{rec}}(n/2) + \Delta_{\texttt{comp}}(n/2) & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Since `comp` uses one element of the input to produce one element of the output, it is relatively easy to show that is is $O(1)$-sensitive to replacement when `f` is $O(1)$—i.e., $\Delta_{\texttt{comp}}(m) \in O(1)$ for any $m$. By straightforward arithmetic, we conclude that $\Delta_{\texttt{rec}}(n) \in O(\log n)$. Since `reducePair` simply calls `rec`, this implies that `reducePair` has logarithmic sensitivity to a replacement.

### 7.4.3 Merge Sort

We analyze the sensitivity of the merge sort algorithm to replacement operations. The recursive case for the derivations of `msort` with inputs that differ in one element, fit the following derivation context.

$$\frac{\texttt{chk}(\ell) \Downarrow \texttt{false} \qquad \texttt{partition}\$\ell\Downarrow(\ell_a,\ell_b)\ \bigtriangledown \qquad \texttt{msort}\$\ell_a\Downarrow\ell_c\ \bigtriangledown \qquad \texttt{msort}\$\ell_b\Downarrow\ell_d\ \bigtriangledown \qquad \texttt{merge}\$(\ell_c,\ell_d)\Downarrow\ell'\ \bigtriangledown}{\texttt{ms}\$(\ell) \Downarrow \ell'}$$

The derivation starts with a check for the length of the list being greater than one. In the recursive case, the list has more than one element so the `lenLT` function returns `false`. Thus, we `partition` the input lists into two lists $\ell_a$ and $\ell_b$ of half the length, sort them to obtain $\ell_c$ and $\ell_d$, and merge the sorted lists. Since both evaluations can be derived from this context, the distance between the derivations is the distance between the derivations substituted for the holes in the context.

Consider the derivations substituted for each hole. Since `lenLT` and `part` are called with the input, the derivations for $\texttt{lenLT}\$\ell_1$ (and $\texttt{part}\$\ell_1$) are related by replacement. As a result, one of $\ell_a$ or $\ell_b$ are also related by replacement. Thus only one of the derivations $\texttt{ms}\$\ell_a$ or $\texttt{ms}\$\ell_b$ are related by replacement and the other pair is identical. Consequently $\texttt{mg}\$(\ell_c, \ell_d)$ derivations are related by replacement. Since all contexts belong to memoized function calls, we can apply the substitution theorem by assuming that all related and identical functions calls in both evaluations return their results in the same locations. Thus, we can write the sensitivity of `msort` as $\Delta_{\texttt{msort}}(n) = 2\Delta_{\texttt{msort}}(n/2) + \Delta_{\texttt{partition}}(n) + \Delta\texttt{merge}(n)$. It is easy to show that `partition` and `lenLT` functions are $O(1)$ sensitive to replacements. Similarly, we can show that `merge` is $O(1)$

sensitive to replacements on average, if we take the average over all permutations of the input list. Thus, we obtain

$$\Delta_{\texttt{msort}}(n) = \begin{cases} \Delta_{\texttt{msort}}(n/2) + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

This recurrence trivially is bounded by $1+4c\log n$, so we conclude that `msort` is $O(\log n)$-sensitive to replacement operations.

### 7.4.4 Filter

As an example of another program that is not naturally stable we consider a standard list filter function `filter`, whose code is shown in Figure 7.7, for which we prove that there are inputs whose traces are separated by a linear distance in the size of the inputs regardless of the choice of locations. In other words, we will prove a lower bound for the sensitivity of `filter`. The reason for which `filter` is not stable is similar to that of the conventional implementation of `reduce` (Subsection 7.4.2), but more subtle because it is primarily determined by the choice of locations rather than the computation being performed.

To see why `filter` can be highly sensitive, it suffices to consider a specialization, which we call `filter0`, that only keeps the nonzero elements. For example, with input lists $L$ = `[0,0,0]` and $L'$ = `[0,0,1]`, `filter0` returns `nil` and `[1]`, respectively. Since we are interested in proving a lower bound only, we can summarize traces by including function calls and put operations only—the omitted parts of the trace will affect the bound by a constant factor assuming that the filtering functions takes constant time. In particular, using the shorthand $\mathtt{M}^{\ell\Downarrow\ell'}(T)$ for the memoization action $\mathtt{M}^{\texttt{filter0\$}\ell\Downarrow\ell'}(T)$, the traces for filter with $L$ and $L'$ are respectively:

$$\mathtt{M}^{\ell_1\Downarrow\ell_n}(\mathtt{M}^{\ell_2\Downarrow\ell_n}(\mathtt{M}^{\ell_3\Downarrow\ell_n}(\mathtt{M}^{\ell_4\Downarrow\ell_n}(\ell_n \overset{\texttt{put}}{\leftarrow} \texttt{nil})))), \text{and}$$

$$\mathtt{M}^{\ell_1\Downarrow\ell_a}(\mathtt{M}^{\ell_2\Downarrow\ell_a}(\mathtt{M}^{\ell_3\Downarrow\ell_a}(\mathtt{M}^{\ell_4\Downarrow\ell_n}(\ell_n \overset{\texttt{put}}{\leftarrow} \texttt{nil})\cdot\ell_a \overset{\texttt{put}}{\leftarrow} \texttt{1::}\ell_n))).$$

Note that the distance between these two traces is greater than 3—the length of the input—because in the second trace three memoization actions return the location $\ell_a$ holding `[1]`, whereas in the first trace $\ell_n$ is returned. Since these locations are different, the memoization actions do not match and contribute to the distance. This example does not lead to a lower bound, however, because we can give two traces for the considered inputs for which the distance is one, e.g.,:

$$\mathtt{M}^{\ell_1\Downarrow\ell_n}(\mathtt{M}^{\ell_2\Downarrow\ell_n}(\mathtt{M}^{\ell_3\Downarrow\ell_n}(\mathtt{M}^{\ell_4\Downarrow\ell_n}(\ell_n \overset{\texttt{put}}{\leftarrow} \texttt{nil})))), \text{and}$$

$$\mathtt{M}^{\ell_1 \Downarrow \ell_n}(\mathtt{M}^{\ell_2 \Downarrow \ell_n}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell'_n}(\ell'_n \overset{\mathtt{put}}{\leftarrow} \mathtt{nil}) \cdot \ell_n \overset{\mathtt{put}}{\leftarrow} \mathtt{1}::\ell'_n))).$$

The idea is to choose the locations in such a way that the traces overlap maximally. It is not difficult to generalize this example for arbitrary lists of the form `[0,...,0,0]` and `[0,...,0,1]`.

We obtain the worst-case inputs by modifying this example to prevent location choices from reducing the distance arbitrarily. Consider parameterized lists of the form $L_1(n)$ = `[(0)`$^n$`,0,(0)`$^n$`]` and $L_2(n)$ = `[(0)`$^n$`,1,(0)`$^n$`]`, where `(0)`$^n$ denotes $n$ repeated `0`'s. We will show that the distance between traces for any two such inputs is at least $n+1$ and thus linear in the size of the input, $2n+1$. For example, the traces for $L_1(1)$= `[0,0,0]` and $L_2(1)$ = `[0,1,0]` have the form:

$$\mathtt{M}^{\ell_1 \Downarrow \ell_n}(\mathtt{M}^{\ell_2 \Downarrow \ell_n}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell_n}(\ell_n \overset{\mathtt{put}}{\leftarrow} \mathtt{nil})))), \text{and}$$

$$\mathtt{M}^{\ell_1 \Downarrow \ell_a}(\mathtt{M}^{\ell_2 \Downarrow \ell_a}(\mathtt{M}^{\ell_3 \Downarrow \ell_n}(\mathtt{M}^{\ell_4 \Downarrow \ell_n}(\ell_n \overset{\mathtt{put}}{\leftarrow} \mathtt{nil})) \cdot \ell_a \overset{\mathtt{put}}{\leftarrow} \mathtt{1}::\ell_n)).$$

These traces have distance greater than 2. Regardless of how we change the locations this distance will not decrease because the return locations of $n$ memoization actions before and after the occurrence of `1` will have to differ. Thus, regardless of which location the other trace chooses to store the empty list, at least half the calls will have a differing location. We can generalize this example with $n = 3$ to arbitrary lists by using our identity and substitution theorems as we did for the `mapA` example. Thus, we conclude that `filter` is $\Omega(n)$-sensitive to a replacement.

This example implies that a self-adjusting computation can do poorly with this implementation of `filter`. As with `reduce`, however, we can give a stable implementation of `filter` by using a compress function similar to `comp` of `reducePair` that applies the filter function to half of the remaining unfiltered elements. We can show that this implementation of `filter` is $O(\log n)$ sensitive under suitable choice of locations.

## 7.5 Cost Semantics with Out-of-Order Memoization

We give several examples where a simple change to the input causes a reordering in the computation. For these applications, in-order computation reuse is asymptotically no more effective than re-computing from scratch but out-of-order reuse enables efficient change-propagation.

We consider a purely functional implementation of quicksort, an imperative depth-first search algorithm, and an incremental parser and evaluator for a lazy language. In

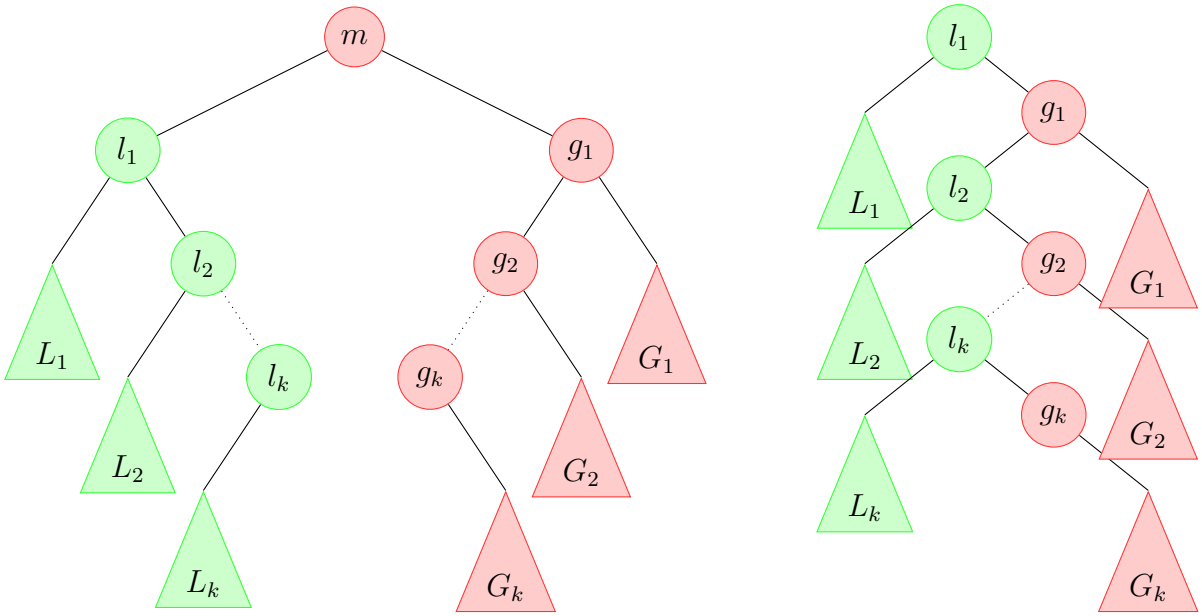



Figure 7.9: Deleting the key $m$ swaps the order in which quicksort performs a large number of subcomputations shown with triangles.

the various examples a non-invasive change can cause reordering in the computation. We compare how in-order and out-of-order computation reuse affect change-propagation in each case.

### 7.5.1 Quicksort

Consider a purely functional implementation of the standard, deterministic quicksort algorithm. The algorithm uses the first element $p$ of the input $I$ as pivot to partition $I$ into sublists $I_<$ and $I_>$, consisting of the keys less than and greater than $p$ respectively. It then sorts $I_>$ followed by $I_<$ recursively supplying the sorted greater-than sublist an accumulator to each recursive call, which returns the accumulator when the input list is empty.

As an example how a single insertion/deletion of a key can invert computation order, consider the partially specified input lists

$$[m, \ldots, l_1, \ldots, g_1, \ldots, l_2, \ldots, g_2, \ldots, l_k, \ldots, g_k, \ldots]$$
$$\text{and } [\ldots, l_1, \ldots, g_1, \ldots, l_2, \ldots, g_2, \ldots, l_k, \ldots, g_k, \ldots],$$

that differ by the key $m$ only. Assume that the elements are ordered as:

$$L_1 < l_1 < \cdots < L_k < l_k < m < g_k < G_k < \cdots < g_1 < G_1$$

Figure 7.9 illustrates the executions of the quicksort algorithm with both inputs. Each circle shows a recursive call and is labeled with the pivot for that call. Each triangle, represents an unspecified part of the computation; triangles labeled with the same letter represent the same computation. Note that the trees look similar: the second tree can be obtained from the first by "zipping" the leftmost spine of the right subtree of the root and the rightmost spine of the left subtree of the root.

To see the problem with in-order reuse, let's consider the traces, which can be viewed as a pre-order, right-to-left traversal of the trees shown in Figure 7.9 that visits the right subtree before the left:

$$m \cdot g_1 \cdot G_1 \cdot g_2 \cdot G_2 \cdot \cdots \cdot g_k \cdot G_k \cdot l_1 \cdot l_2 \cdots l_k \cdot L_k \cdots L_2 \cdot L_1$$

and

$$l_1 \cdot g_1 \cdot G_1 \cdot l_2 \cdot g_2 \cdot G_2 \cdot \cdots \cdot l_k \cdot g_k \cdot G_k \cdot L_k \cdots L_2 \cdot L_1$$

Since $l_1$ is visited earlier in the second run, greedy in-order reuse discards the intervening computation ($g_1 \cdot G_1 \cdots g_k \cdot G_k$) which must then be recomputed. Thus in-order reuse can make trace distance $O(n \log n)$, which asymptotically the same as recomputing from scratch (e.g., when the two subtree of the root are (approximately) the same size). Under out-of-order reuse, the initial trace can be reordered into the second trace by decomposing it into subtrees of computation ($L_1$, $L_2$, etc.) which can be reused in their entirety and only performing the computation for the nodes ($l_1$, $l_2$, etc.) afresh. Therefore the distance between the two runs is linear in the height of the tree which is a $O(\log n)$ speedup on average over all insertions and deletions. Some bounds can be achieved using in-order computation reuse by using extra annotations to prevent reuse. This requires a relatively detailed understanding of algorithmic properties of quicksort.

### 7.5.2 Depth-First Search on Graphs

Depth-first search (DFS) is a fundamental graph algorithm that constitutes the substep of many other algorithms (e.g., topological sorting). Starting at a specified node, depth-first search visits a node $u$ of the graph by visiting the targets of the outgoing edges of $u$, backtracking after all outgoing edges are visited. An implementation of DFS typically maintains a *visited* flag set when a node is first visited. As an example consider the graph shown in Figure 7.10 consisting of a root $a$ linked with two connected components (subgraphs) $B$ and $C$; here edges are ordered from left to right. Starting at root $a$, DFS visits $b_1$

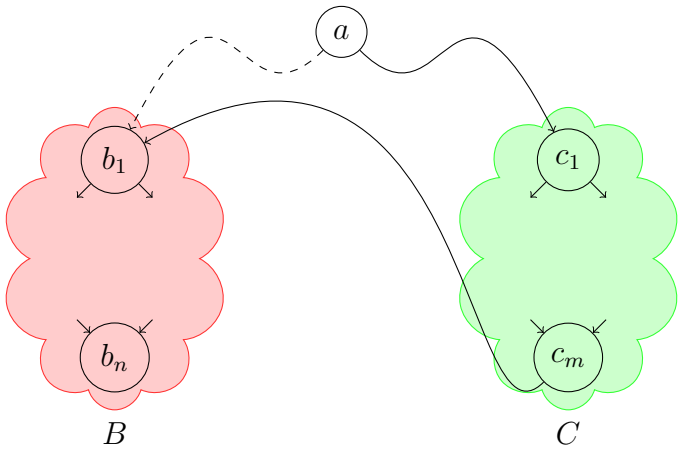



Figure 7.10: Deleting edge $(a, b_1)$ swaps the order in which a DFS visits components $B$ and $C$.

and the component $B$, followed by $c_1$ and the component $C$. Before completing $C$, DFS visits the edge $(c_m, b_1)$ backtracking to $a$ after finding $b_1$ is visited because its visited flag is set.

Consider now deleting the edge $(a, b_1)$ and performing a DFS. DFS will now visit $c_1$ and the component $C$, then take the edge $(c_m, b_1)$ and find $b_1$ is unvisited this time. It will then visit $b_1$ and the component $B$, then return to $c_1$ and $a$. We can visualize the self-adjusting-computation traces for DFS on the graph before and after the deletion of $(a, b_1)$ respectively as follows:

$$\boxed{\text{visit}(a)} \cdot \boxed{\text{visit}(b_1) \cdots \text{visit}(b_n)} \cdot \boxed{\text{visit}(c_1) \cdots \text{visit}(c_m)}$$

and

$$\boxed{\text{visit}(a)} \cdot \boxed{\text{visit}(c_1) \cdots \text{visit}(c_m)} \cdot \boxed{\text{visit}(b_1) \cdots \text{visit}(b_n)}$$

Here the boxed subtraces show the subtraces for components $B$ and $C$ respectively, which appear in opposite order in the two traces. The edit distance between two traces can therefore be asymptotically the same as the size of the whole trace. Since reordering the two subtraces easily yields one trace from the other, under out-of-order reuse their distance reduces to $O(1)$ since change-propagation only has to account for the reordering.

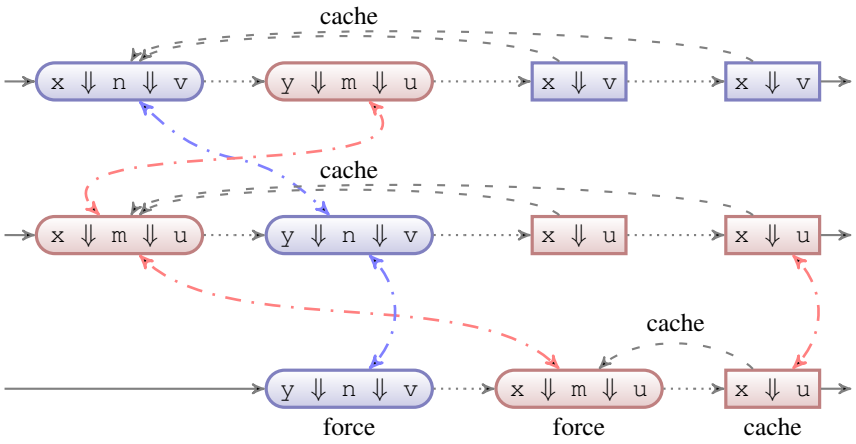



Figure 7.11: Evaluation with `(i,x,y)` bound to `(true,n,m)` (top), `(true,m,n)` (middle), `(false,m,n)` (bottom).

### 7.5.3 Incremental Parsing and Evaluation

Interactive program development can be done through incremental parsing and evaluation of a source program. Self-adjusting computation is a natural way to write an incremental parser and evaluator. A packrat parser is a top-down parser that uses laziness to cache intermediate parses and avoid unnecessary work [Ford, 2002], which can be implemented using the suspensions for laziness of the SrcLazy language (see Chapter 3). Moreover, the resulting abstract syntax tree from the parser can be fed to an evaluator. Reordering fragments of code, however, is problematic for incremental parsing and evaluation because in-order memoization prevents reuse of the work done on both blocks of rearranged code, and incremental evaluation of a lazy (call-by-need) language can be inefficient because of the order of evaluation is sensitive to the surrounding context.

Consider parsing and evaluating the following program, assuming short-circuiting evaluation of Boolean expressions and lazy evaluation of the bound variables. Note that `x` and `y` are bound to expressions `n` and `m`, and their order of evaluation depends on `i`.

```
let val (i,x,y) = (true,n,m)
in (i andalso x) ... y ... x ... x end
```

Packrat parsing creates a trace traversing the string in left-to-right order and evaluation traverses the corresponding abstract syntax tree. Figure 7.11 compares three runs of the evaluator with different bindings for `(i,x,y)` and shows that the order of computation

depends on the value of the inputs. Under the initial binding `(true,n,m)`, `x` first evaluates `n` to `v`, next `y` evaluates `m` to `u`, and finally, due to the lazy evaluation semantics, the second and third mentions of `x` use the cached result `v` (indicated by the dashed arrow back to the first evaluation).

Swapping the expressions `n` and `m` forces the re-parsing of the expression `(true,m,n)`. Under in-order reuse, the parse and evaluation of `m` may be reused at the expense of discarding the parse and evaluation of `n`, which must then be re-computed. Thus change-propagation takes time at least proportional to re-parsing and re-evaluating the expression `m`. Under out-of-order reuse, the parsing and evaluation of `m` and `n` can be swapped, so the trace distance between the two runs is constant and change-propagation takes $O(1)$. Under both in-order and out-of-order reuse, the second and third mentions of `x` must be updated with the new cached result `u`, and the rest of the evaluation may also need to be updated according to the new values of `x` and `y`.

Changing the binding of `i` from `true` to `false` doesn't cause a reordering that affects the parser, but the evaluation is affected due to short-circuiting evaluation of Booleans and laziness. Since the **`andalso`** short-circuits evaluation, the first mention of `x` isn't demanded and the evaluation of `n` and `m` is swapped relative to the previous run. Therefore `y` evaluates `m` to `u` before the second mention of `x` evaluates `n` to `v`; the third mention of `x` still uses the cached value of `u` but now references the evaluation of the second (instead of the first) mention of `x`. In-order reuse would discard the evaluation of `m` to reuse the previous evaluation of `n`, and thus the distance between the two runs as well as the time for update would be proportional to the evaluation to `m`. On the other hand, out-of-order reuse allows reusing both evaluations and update only needs to deal with the possibly different result of evaluating `i` **`andalso`** `x`.

# Chapter 8

# Conclusion

In this chapter, we discuss related work and directions for future work.

The problem of having computation respond to input changes has been studied in the algorithms community under the name of dynamic and kinetic algorithms and in the programming languages community under the name of incremental computation. Self-adjusting computation generalizes and extends these earlier approaches to improve their effectiveness.

## 8.1 Related Work

### 8.1.1 Dynamic and Kinetic Algorithms and Data Structures

Algorithms researchers devise *dynamic* and *kinetic* algorithms and data structures with an *explicit* mechanism for updating the computation optimized for particular classes of small discrete or continuous changes. Dynamic algorithms and data structures have been developed for trees [Sleator and Tarjan, 1985], graphs [Eppstein et al., 1999], and computational geometry applications [Chiang and Tamassia, 1992, Guibas and Russel, 2004]. For motion simulation, kinetic algorithms and data structures maintain some property of geometric objects undergoing continuous motion by tracking the discrete events that affect the property [Agarwal et al., 2002, Guibas, 2004, Basch et al., 1999, Russel et al., 2007, Russel, 2007].

Since discrete or continuous changes to data often cause small changes to the output in these applications, it is often possible to update the computation and output faster—often

asymptotically—than re-computing from scratch. By taking advantage of the structure of the particular problem being considered, the algorithmic approach facilitates designing efficient, often optimal algorithms.

Designing and analyzing these algorithms, however, can be quite difficult even for problems that are simple in the absence of data changes, such as many of the simple graph and computational geometry algorithms. Since these algorithms are specialized to support a particular class of changes, an algorithm may be efficient for some changes but not others, and difficult to adapt to a slightly different problem or compose into larger a larger algorithm. Furthermore, due to their inherent complexity, it can be difficult to implement and use such algorithms in practice. Kinetic algorithms pose additional implementation challenges due to difficulties with motion modeling and handling of numerical errors.

### 8.1.2 Incremental Computation

Programming languages researchers have developed general-purpose compile- and run-time techniques to transform static programs into an *incremental* versions that can respond to input changes [Ramalingam and Reps, 1993]. The common idea is to maintain information about the computation that can be used to efficiently update the output after changes to the input. The most effective incremental computation techniques are based on dependence graphs [Demers et al., 1981], memoization [Pugh and Teitelbaum, 1989], and partial evaluation [Field and Teitelbaum, 1990].

#### Dependence Graphs

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 et al. [1981] introduced *static dependence graphs* for incremental evaluation of attribute grammars, and Reps [1982] showed an optimal change-propagation algorithm. Hoover [1987] generalized the approach beyond the domain of attribute grammars. Yellin and Strom's INC language [Yellin and Strom, 1991] applies dependence graph techniques to a purely functional language with bags but without recursion. The fixed nature of static dependence graphs prevents the change-propagation algorithm from updating the dependence structure, thus significantly restricting the class of computations that can be incrementalized.

### Memoization

*Memoization* (*a.k.a. function caching*) [Bellman, 1957, McCarthy, 1963, Michie, 1968] is a classical optimization technique for pure programs to cache the result of a subcomputation (e.g., a function call) that may be performed *multiple times* during a *single run*. When a subcomputation is invoked for the first time, its result is recorded; thereafter when the same subcomputation is invoked, the recorded result is immediately available without having to execute the body of the subcomputation. This technique is sometimes coupled with a lookup table that maps keys to cached answers

Pugh and Teitelbaum [Pugh, 1988, Pugh and Teitelbaum, 1989] were the first to apply memoization 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].

Memoization can improve the efficiency of incremental computation when executions of a program with similar inputs perform similar function calls. Unfortunately, a small input change can cause some function call to receive modified arguments, which in turn causes all ancestors in the call tree to be re-executed thus preventing their reuse via memoization.

In self-adjusting computation, *computation memoization* uses the trace of a program's execution to identify and reuse of work *across runs* subject to the restriction that a trace fragment is reused *at most once* during change-propagation. Since changeable data is explicitly handled with an indirection through the store, only the computation that is affected by input changes must be re-executed. Furthermore, SrcLazy suspensions combine the features of multi-write modrefs and out-of-order computation memoization to provide the functionality of classical memoization in the context of self-adjusting computation.

### Partial Evaluation

Sundaresh and Hudak [Sundaresh and Hudak, 1991] use partial evaluation to optimize incremental evaluation. This approach requires the user to specify a subset of the inputs to be fixed and the program is then partially evaluated with respect to the fixed inputs. An incremental evaluation can then process input changes faster by evaluating the remainder of the specialized program. The main limitation of this approach is that it only allows input changes within the predetermined subset. Field and Teitelbaum [Field, 1991, Field and Teitelbaum, 1990] present techniques for incremental computation in the context of lambda calculus. Their approach is similar to Hudak and Sundaresh's, but they present formal reduction systems that optimally use partially evaluated results. The main limitation

of the partial evaluation approach is that input changes are restricted to a predetermined partition.

### 8.1.3 Self-Adjusting Computation

**Adaptivity and Computation Memoization.** Self-adjusting computation aims to bridge the gap between algorithmic approaches—which tend to be difficult to use in practice—and programming language techniques—which tend to be inefficient—with a general-purpose language and a generic change-propagation mechanism to respond to input changes efficiently.

Acar, Blelloch, and Harper [Acar et al., 2006c] proposed Adaptive Functional Programming (AFP) with the modal language AFL to stratify self-adjusting programs according to whether the data and computation are *stable* or *changeable*—i.e., whether they are affected by input changes. That work introduced *dynamic dependence graphs* (DDGs) as a means for adaptivity to represent data and control dependencies. The execution of an AFL program builds a DDG of the computation. After the inputs change, a change-propagation algorithm updates both the structure of the DDG—by inserting and deleting dependencies as necessary—and the output by re-executing the parts of the computation affected by the changes as necessary. Change-propagating and AFL program 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.

Separate work by Acar, Blelloch, and Harper [Acar et al., 2003] proposed selective memoization with the modal language MFL to identify data and control dependencies the input and result of memoizing functions. Subsequent work by Acar et al. [Acar, 2005, Acar et al., 2009] combined the complementary features of adaptivity and computation memoization in the modal language SLf for self-adjusting computation. That approach employs *memoized* DDGs to identify data and control dependencies as well as opportunities for reusing subgraphs of deleted DDGs. The dynamic semantics combines a standard evaluation relation for from-scratch runs as well as a change-propagation relation for updating a previous run to different inputs. Adaptivity re-executes the parts of the program affected by input changes while computation memoization reuses the unaffected parts of a previous run.

**Previous Languages.** Self-adjusting computation has been realized through several formal languages with corresponding implementations. The modal languages AFL, MFL,

and SLf were implemented as Standard ML libraries, although the modal type discipline was only simulated through a monadic interface with explicit destination-passing. The monadic discipline required all changeable data to be threaded through the monad and made the scope of reads explicit, thus identifying data and control dependencies. The library imposed certain proper-usage guidelines such as ending a sequence of reads with a write, only writing once into a destination, and not having any reads after a write. Those restrictions, however, could only be enforced manually without changing the Standard ML's type system to support the modal type discipline.

Carlsson [Carlsson, 2002] recast the modal language as a monadic interface in Haskell that statically enforced the proper-usage guidelines of the SML library for AFL. By using monads, the Haskell library ensures some correct-usage properties that the AFP library did not enforce. A later version of the SML library for SLf 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.

The proposed monadic approaches required the programmer to substantially rewrite the source program to obtain a self-adjusting version. Furthermore, they could not guarantee safety of memoization primitives, which are crucial to the effectiveness of the approach, making it difficult to write correct self-adjusting programs. Compilation support was thus suggested as necessary for scaling to large programs and for enforcing the numerous invariants required for self-adjusting computation to work correctly [Acar et al., 2006a].

**Our Languages.** Our work [Ley-Wild et al., 2008b] proposed a considerably simpler and safe interface that offers a more natural programming model for self-adjusting computation. An ordinary program can be instrumented with simple annotations in the direct style Src language with first-class modifiable references and memoizing functions. These primitives can be inserted anywhere in the code subject to some simple typing constraints, which can be statically checked using simple extensions to a ML-style type system.

Whereas the monadic approach included a `lift` primitive that conflated keyed allocation from memoization, the Src primitives provide orthogonal means for adaptivity and memoization. Since modifiable references are ML-style references, their allocation and initialization are combined, which avoids the safety problems of the destination-passing discipline. Since modrefs identify changeable data directly, programs don't need to be explicitly stratified into stable and changeable modes. The compiler-generated equality and hashing functions guarantee the safety of memoization. Thus well-typed programs are guaranteed to respond to changes correctly via change-propagation.

The adaptive continuation-passing style (ACPS) transformation compiles Src programs into equivalent self-adjusting versions in the CPS Tgt language, which subsumes the need to rewrite a program into a modal or monadic language The translation identifies sufficient information about the program to employ the CPS primitives inspired by the monadic primitives. In particular, the translation uses continuations as a coarse approximation to programmer-supplied fine-grain dependencies in the modal/monadic setting. Since modref operations are in CPS, a write can explicitly reuse an allocation from a previous run during change-propagation to maximize computation reuse. To ensure that change-propagation remains efficient with the compiler-inferred dependencies, the translation generates memoizing CPS functions that may be reused even when their continuations differ. This is achieved by memoizing continuations and by treating continuations as changeable data by threading them through the store in modrefs indexed by the function's arguments.

The $\Delta$ML language and compiler shows that the proposal is realistic by extending the SML language and the MLton optimizing compiler. The experimental evaluation of $\Delta$ML indicates that the approach is consistent with the previous proposal based on manual re-writing in terms of practical performance and asymptotic complexity.

**Laziness and Out-of-Order Computation Memoization.** Existing designs for self-adjusting computation focus on *strict* languages with *call-by-value* functions that *eagerly* evaluate function arguments and none of them supported efficient reordering. In *non-strict* languages with *call-by-name* functions use normal-order evaluation to postpone evaluation of arguments until actually needed. Lazy languages refine call-by-name with *call-by-need* functions that delay evaluation of a suspended computation until it is actually needed and thereafter keep a cached version of the result for subsequent uses in the classical sense memoization [Michie, 1968]. Although laziness is a standard optimization for non-strict languages, it can also be added as an orthogonal construct to strict languages, as done in our SrcLazy language.

Laziness improves the expressivity of a language by giving the user control over the amount of computation performed and enabling the finite representation of infinite data structures. Furthermore, it improves performance by avoiding unnecessary computation of unused values and preventing redundant computation of values that are used multiple times. Vuillemin [1974] proposed call-by-name evaluation with a *delay rule* that avoids redundant computation through caching—essentially call-by-need evaluation—and proved its optimality. Friedman and Wise [1976] designed a LISP evaluator with a lazy list constructor that affords fine-grained laziness. The crucial characteristic of their language is that the cons constructor delays evaluation of its contents until they are needed and

thereafter caches their value. A similar LISP language with a lazy cons was studied by Henderson and Jr. [1976], although they gave a Scott-Strachey semantics. Hughes [1985] proposed *lazy memo-functions* that explicitly integrate memoization into a call-by-name language, considered various applications, and discussed practical implementation issues. The formal semantics of a call-by-need $\lambda$-calculus was formulated by Ariola et al. [1995] using evaluation contexts, whereas our SrcLazy language explicitly uses state to record the suspended thunk and cached value.

Although Carlsson [Carlsson, 2002] targets Haskell—a non-strict language with lazy evaluation—, the use of monads forces the eager evaluation of the self-adjusting computation primitives. Since the first use of a lazy computation depends dynamically and non-locally on the surrounding program, the ordering of computation in lazy programs isn't stable under data changes and therefore laziness is problematic for self-adjusting computation with in-order memoization. Since previous approaches maintain a time line with insertions and deletions, they do not work well when swapping the order of operations. Out-of-order memoization allows large changes in the ordering of computations, and therefore SrcLazy can provide suspensions with call-by-need behavior in a self-adjusting language.

**Other Languages.** Hammer et al. retargeted self-adjusting computation for a low-level imperative C-like language CEAL. That work extends change-propagation to support efficient garbage collection [Hammer and Acar, 2008] and provides a compiler for C [Hammer et al., 2009].

**Consistency of Change-Propagation.** The work of Acar et al. on the modal languages AFL [Acar et al., 2006c], MFL [Acar et al., 2003], and SLf [Acar, 2005] showed the consistency of change-propagation—i.e., that change-propagation is extensionally equivalent to a from-scratch run. Acar, Blume, and Donham [Acar et al., 2007b] formalized the consistency of the pure language AML with single-write modrefs and memoization in Twelf [Pfenning and Schürmann, 1999]. Acar, Ahmed, and Blume [Acar et al., 2008a] proved the consistency of the monadic language language with multi-write modrefs and memoization SAIL. In their formulation of the semantics, change-propagation could not re-allocate locations allocated in a previous run. This limited their proof to show that the results obtained by change-propagation and evaluation are *isomorphic* and required using step-indexed logical relations.

In our formulation of self-adjusting computation, we support arbitrary TDTs as well as in-order and out-of-order memoization. Our proof of consistency [Ley-Wild et al., 2009]

is a straightforward structural induction, which is made possible by formulating change-propagation as a full replay mechanism that can re-allocate locations. Furthermore, we prove that the ACPS translation preserves the intensional and extensional semantics from Src to Tgt.

**Applications.** Self-adjusting computation has been shown to be effective for a reasonably broad range of problems including hardware verification [Santambrogio et al., 2007], machine learning [Acar et al., 2007c, 2008c], motion simulation [Acar et al., 2006d, 2008b], and other algorithmic problems [Acar et al., 2004, 2005, 2009].

Self-adjusting computation techniques have served to make progress on open problems in computational geometry: practically efficient motion simulation of convex hulls in three dimensions [Acar et al., 2008b] and dynamic maintenance of well-spaced point sets [Acar et al., 2010b].

Shankar and Bodik [Shankar and Bodik, 2007] applied self-adjusting computation techniques specialized for incremental invariant checking in Java. That work restricts the control flow of invariant checks to prevent return values from functions calls to affect loop conditionals or other function calls.

### 8.1.4 Evaluation

Previous evaluations of self-adjusting computation relied on empirical and algorithmic analysis techniques. Earlier work [Acar et al., 2004] analyzed the performance of change-propagation for the tree contraction problem. Most applications of self-adjusting computation, however, evaluated its effectiveness experimentally [Acar et al., 2005, 2009]. Those evaluations have experimentally observed that updating a computation is faster than computing from scratch, sometimes with an apparently asymptotic speedup. While empirical nature of those studies demonstrated the practical advantages of self-adjusting computation, they do not establish formal results relating the time for update and from-scratch evaluation.

**Cost Semantics.** The complexity of the semantics in modal/monadic approaches made it difficult to reason directly about the effectiveness of self-adjusting computation. Our work on cost semantics [Ley-Wild et al., 2009] provides a formal theory for reasoning about the running time of programs and determining how much work must be done to update one run into another. The semantics produces traces to capture the essential of structure of the computation consisting of function calls and individual store operations. The user can

determine the responsiveness of compiled self-adjusting programs by calculating an *edit distance* between traces of source programs, without having to reason about evaluation contexts or global state. These results are made possible by (1) a compilation mechanism that can translate ordinary program into a self-adjusting version while preserving its from-scratch efficiency, and (2) by techniques for matching evaluation contexts appropriately without exposing them to the user for source-level reasoning.

This work is inspired on previous work [Sands, 1990a, Rosendahl, 1989]. on profiling or cost semantics for reasoning about resource requirements of programs. The idea of instrumenting evaluations to generate cost information has been particularly important in high-level languages such as lazy [Sands, 1990a,b, Sansom and Jones, 1995] and parallel languages [Blelloch and Greiner, 1995, 1996, Spoonhower et al., 2008] where it is particularly difficult to relate execution time to the source code. The idea of having a cost semantics construct a trace resembles the techniques used for reasoning about resource-consumption in (pure) parallel programs, although we allow store effects and compare different runs with distance. In the context of incremental computation, we know of no other work that offers a source-level cost semantics for reasoning about effectiveness of incremental update mechanisms.

**Asymptotics.** A common limitation of cost semantics-based approaches to performance analysis is that they usually only apply to concrete evaluations. We show that this need not be the case by providing techniques for generalizing the trace distances of concrete evaluations to arbitrary inputs, composing trace distances, and reasoning with trace contexts. For illustrative purposes, we derive asymptotic bounds for several examples. We expect these results to lead to a more formal and precise reasoning of effectiveness of self-adjusting programs as well as profiling tools that can infer concrete and perhaps asymptotic complexity bounds.

### 8.1.5 Retroactive Data Structures and Traceable Data Types

The initial work on self-adjusting computation provided single-write modrefs to manipulate changeable data in purely functional programs. Subsequent work by Acar, Ahmed, and Blume [Acar et al., 2008a] extended the semantics to support multi-write modifiable references in imperative self-adjusting programs. Other work [Acar et al., 2008b] shows that self-adjusting computation can be effective in simulation of motion in three dimensions, but relies on unsafe modifications to the change-propagation mechanism to support efficient adaptivity for continuous data. We generalize self-adjusting computation to support traceable data types as containers for changeable data, which subsume modrefs and

provide a safe way to operate on continuously-changing values.

Demaine, Iacono, and Langerman [Demaine et al., 2004] developed *retroactive data structures* that maintain a time-ordered history of update operations (but not query operations) and allow changing (invoking or revoking) the history of operations while making subsequent operations consistent with the revised history. Since that work did not maintain the history of queries, the data structures would only work with update operations that were *oblivious* in the sense that they do not depend on earlier queries. Acar, Blelloch, and Tangwongsan [Acar et al., 2007a] developed *non-oblivious* retroactive data structures (*a.k.a.* traceable data types) that maintain a history of both updates and queries and thus support update operations that *do* depend on earlier queries. They designed several data structures with efficient algorithmic techniques to identify the next inconsistent action following an invocation and revocation.

**Traceable Data Types in Self-Adjusting Computation.** We support abstract data types operations in self-adjusting computation by generalizing trace actions to represent high-level TDT operations instead of individual modref operations. We show how to integrate the invoke/revoke TDT operations into change-propagation for programs to respond automatically to input changes, while the user has access to the standard data type operations. Since the number of accesses to an abstract data type can be asymptotically less than the number of memory accesses in a data structure implementation, our approach can asymptotically reduce the number of dependencies to be tracked. For self-adjusting computation, these techniques result in dramatic time and space improvements. Furthermore, in some cases the trace can be stable with respect to the data type operations even if it isn't at the level of memory cell operations.

**FrTime and Flapjax.** Cooper and Krishnamurthi [Cooper and Krishnamurthi, 2004, 2006] developed FrTime as an extension of Scheme with dynamic dataflow for writing interactive programs. In their setting, data can depend on *time* as a continuously-changing input and all dependencies are implicitly tracked by *signals*, which are akin to modrefs in self-adjusting computation. Any operation that depends on time—i.e., manipulates a signal—must return the result in a signal to register the dependence. Since automatic dependence-tracking creates redundant signals to to store intermediate values, in subsequent work [Burchett et al., 2007], they propose a static optimization technique to eliminate this redundancy. The Flapjax language [Meyerovich et al., 2009] adapts these ideas for writing web applications.

By contrast, self-adjusting computation provides primitives for controlling the gran-

ularity of dependence-tracking. Both self-adjusting computation and FrTime construct a dynamic dependence graph and use it to update the computation by a change-propagation algorithm. Self-adjusting computation represents DDGs using order-maintenance data structure and updates the computation in topological order. FrTime updates the computation using height information, which complicates the dynamic maintenance of the DDG and requires the programmer to use explicit delay statements to handle cyclic dependencies—which arise naturally in their setting. This can also be inefficient because deleting a DDG node can change the height of all other nodes, requiring linear time in the size of the DDG.

## 8.2 Future Work

### 8.2.1 Self-Adjusting Computation

**Out-of-Order Memoization.** The Tgt language gives a formal semantics for out-of-order memoization and the CPA algorithm accounts for handling mutable and immutable modrefs in the presence of reordering, but the ΔML implementation only supports in-order memoization. To make TDTs compatible with out-of-order memoization, the semantics would have to maintain consistency of multiple trace slices. To support out-of-order memoization in the ΔML implementation, the order-maintenance time line would have to be replaced with a version that supports reordering time segments.

**Trace Distance for TDTs.** The Src and Tgt trace distance only account for the cost of mutable and immutable modrefs. To account for the cost of TDTs, the semantics would have to include the cost of invoking and revoking operations as well as finding inversions (*cf.* Subsection 6.5.2).

**Meta-Logic for Cost Semantics.** Our proof system for trace distance applies to concrete traces and trace contexts, while in our examples we use it to reason schematically over classes of contexts and input changes. To fully formalize the examples, we would need a meta-logic that permits quantification over contexts and classes of input changes, and can express asymptotic bounds. Such a meta-logic could be extended with theorem-proving capabilities to help automate finding bounds on distance.

**Garbage Collection.** In self-adjusting computation, traces make up a large percentage of the memory footprint, have a long lifespan, and old parts of the trace can point to new data. This contravenes the assumptions of generational garbage collectors. Empirical studies have observed that garbage collection contributes significantly to the degradation of performance when trace data become large relative to available memory. A specialized garbage collector for self-adjusting computation could take advantage of the change-propagation semantics because it explicitly determines when a trace segment becomes garbage—i.e., when it's discarded.

**Annotation Inference.** The direct style Src primitives provide a more natural programming style than the modal and monadic primitives of previous work, but still require the programmer to instrument an ordinary program to obtain a self-adjusting version. Heuristics for adaptivity and computation memoization could be used to infer which data should be marked as changeable and which functions should be memoizing, thus automating the instrumentation of ordinary programs.

**Metalanguage.** A self-adjusting program can be regarded as a *core*-level computation, while a *meta*-level *host mutator* program examines the outputs, changes the inputs, and drives change-propagation. Existing proposals for self-adjusting computation give the dynamic semantics in terms of adjusting a previous execution with a different set of inputs to produce an updated execution. The associated implementations provide separate core- and meta-level primitives for manipulating data, and the mutator can additionally invoke change-propagation. Due to the lack of a formal semantics for the mutator, the behavior and usage guidelines of the mutator's primitives have only been described informally.

To formally capture the possible interactions between the meta- and core-levels, we need to develop a formal *metalanguage* that specifies the behavior of the host mutator and how it can manipulate changeable data. The design of the metalanguage should be informed by the restrictions imposed by the implementation, so that the informal proper-usage guidelines are statically enforced by the metalanguage. In particular, the efficient implementation of self-adjusting computation allows the mutator to change a self-adjusting program's inputs but not any intermediate data created during the run. Moreover, the mutator can inspect the result of a self-adjusting program and use it to determine how to change inputs. However, due to invariants of the implementation, the mutator cannot hold on to output data across runs or feed back the output data of one run as the input of a subsequent run. We believe it is possible to devise a type system that uses permissions to statically distinguish input and runtime data in order to identify how the mutator can manipulate that data.

### 8.2.2 Functional Reactive Programming

Functional Reactive Programming (FRP) is a declarative and compositional approach to describing hybrid continuous- and discrete-time systems. FRP programs are based on a primitive notion of (continuous) time and functions of time called *signals*. Signals that are continuously-varying *behaviors* or discretely-occurring *events*. An FRP language usually provides combinators to build *networks* of computation nodes that communicate through signals.

Elliott and Hudak [Elliott and Hudak, 1997] introduced FRP for describing animations, and it has since been applied to sound synthesis, robotics, and computer vision. The initial work on FRP [Elliott and Hudak, 1997, Wan and Hudak, 2000] gives a denotational semantics, making it difficult to establish time and space bounds. Elliott discusses some of the difficulties with providing an efficient implementation for FRP [Elliott, 1998]. Follow-up work gave an operational semantics for a strict subset of FRP, called Real-Time FRP [Wan et al., 2001]. Existing implementations typically simulate the evolution in time of an FRP network by periodically evaluating the entire network relative to the current values of the input signals. This sampling approach can be inefficient because the current inputs may only differ slightly (or not at all) from the previous inputs, so the current evaluation of the network may not differ much from the previous evaluation.

FRP remains largely orthogonal to incremental computation, except for Cooper and Krishnamurthi's FrTime [Cooper and Krishnamurthi, 2004] extends a functional language with a primitive notion of time and support for incrementality. Since computations performed at consecutive time steps are generally similar, FRP may benefit from incremental update techniques. Self-adjusting computation may be applied to FRP by representing the network as a syntax tree and running a self-adjusting evaluator program to sample the value of the network and evolve the network according to the combinator semantics. Change-propagating the evaluator program could reuse previous evaluations of the network and thus only perform the necessary computation to re-sample the value under input changes and update the structure of the network as necessary.

A self-adjusting approach presents several challenges which we believe can be resolved with new general-purpose primitives. First, an FRP network changes over time. By representing the network as changeable data, the evaluator would update the network in place and thus the updated network would have to become the input for the next sampling cycle. Existing self-adjusting computation implementations, however, only allow the inputs to be changed and disallow the outputs to be fed back as inputs for subsequent runs. We would remedy this by introducing a notion of *persistent* data that allows data (e.g., the FRP network) to be modified in place and have the updated value fed back as the initial value for the next run.

Second, FRP allows a signal to be used in multiple parts of the network. According to the semantics of FRP, a signal should only be evaluated once per sampling but the result can be used in multiple parts of the network. This presents another challenge in terms of how to correctly and efficiently only perform a computation once during the run of a self-adjusting program, but still allow the result to be used multiple times. We believe that our work on out-of-order memoization and suspensions provides a suitable form of classical memoization that can be used to evaluate a network once and cache the result for subsequent uses in the same run.

# Bibliography


MLton. http://mlton.org/. 2.1.2, 6.1

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. 8.1.2

Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005. 1.1, 2.2.1, 2.4.1, 8.1.3, 8.1.3

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective Memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003. 1.1, 1.1, 1.1, 2.1.1, 2.2.1, 8.1.2, 8.1.3, 8.1.3

Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing Static Algorithms with Applications to Dynamic Trees and History Independence. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 2004. 1.1, 2.2.1, 8.1.3, 8.1.4

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. 1.1, 8.1.3, 8.1.4

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. 2.1.1, 1, 2.1.2, 6.1, 6.3, 6.5, 7.1, 7.2.5, 8.1.3

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*, 2006b. 1.1, 1.1, 1.1, 2.1.1, 1, 2.1.2, 2.2.1, 2.4.1, 6.5, 7.1

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive Functional Programming. *ACM Transactions on Programming Languages and Systems*, 28(6):990–1034, 2006c. 1.1, 1.1, 1.1, 2.1.1, 2.2.1, 2.3.1, 8.1.3, 8.1.3

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*, pages 636–647, September 2006d. 1.1, 2.2.1, 8.1.3

Umut A. Acar, Guy E. Blelloch, and Kanat Tangwongsan. Non-oblivious Retroactive Data Structures. Technical report, Carnegie Mellon University, 2007a. 2.3.2, 2.3.3, 6.5.3, 8.1.5

Umut A. Acar, Matthias Blume, and Jacob Donham. A Consistent Semantics of Self-Adjusting Computation. In *Proceedings of the 16th Annual European Symposium on Programming (ESOP)*, 2007b. 8.1.3

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian Inference. In *Neural Information Processing Systems (NIPS)*, 2007c. 1.1, 8.1.3

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*, 2008a. 2.3.1, 8.1.3, 8.1.5

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 2008b. 1.1, 2.2.1, 10, 8.1.3, 8.1.5

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Inference on General Graphical Models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008c. 1.1, 8.1.3

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An Experimental Analysis of Self-Adjusting Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(1):3:1–3:53, 2009. 1.1, 8.1.3, 8.1.3, 8.1.4

Umut A. Acar, Guy E. Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Türkoğlu. Traceable Data Types for Self-Adjusting Computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010a. 3, 4, 6, 7

Umut A. Acar, Andrew Cotter, Benoit Hudson, and Duru Türkoğlu. Dynamic Well-Spaced Point Sets. In *SCG '10: Proceedings of the 26th Annual Symposium on Computational Geometry*, 2010b. 8.1.3

Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolefson. Algorithmic Issues in Modeling Motion. *ACM Comput. Surv.*, 34(4):550–572, 2002. ISSN 0360-0300. 8.1.1

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1991. 2.1.2, 4.1

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The Call-by-Need Lambda Calculus. In *Principles of Programming Languages*, pages 233–246, 1995. 8.1.3

C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996. 7.2.1, 7.3.8

Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999. 8.1.1

Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957. 8.1.2

Guy Blelloch and John Greiner. Parallelism in Sequential Functional Languages. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 226–237, 1995. ISBN 0-89791-719-7. 2.2.2, 8.1.4

Guy E. Blelloch and John Greiner. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 213–225. ACM, 1996. 2.2.2, 8.1.4

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. 8.1.5

Magnus Carlsson. Monads for Incremental Computing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming*, pages 26–35. ACM Press, 2002. 2.1.1, 8.1.3, 8.1.3

Y.-J. Chiang and R. Tamassia. Dynamic Algorithms in Computational Geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992. 8.1.1

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. 8.1.5, 8.2.2

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. 8.1.5

Olivier Danvy and John Hatcliff. CPS Transformation after Strictness Analysis. *Letters on Programming Languages and Systems (LOPLS)*, 1(3):195–212, 1993a. 5.2

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. 5.2

Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 281–290, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. ISBN 0-89871-558-X. 2.3.2, 8.1.5

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. 8.1.2, 8.1.2

P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 365–372, 1987. 6.5.1

Conal Elliott. Functional Implementations of Continuous Modeled Animation. *Lecture Notes in Computer Science*, 1490:284–299, 1998. 8.2.2

Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997. 8.2.2

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. 8.1.1

J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LISP and Functional Programming*, pages 307–322, June 1990. 8.1.2, 8.1.2

John Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems*. PhD thesis, Department of Computer Science, Cornell University, November 1991. 8.1.2

Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *ICFP*, pages 36–47, 2002. 7.5.3

Daniel P. Friedman and David S. Wise. CONS Should Not Evaluate its Arguments. In *ICALP*, pages 257–284, 1976. 8.1.3

L. Guibas. Modeling motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004. 8.1.1

Leonidas Guibas and Daniel Russel. An empirical comparison of techniques for updating Delaunay triangulations. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 170–179, New York, NY, USA, 2004. ACM Press. 8.1.1

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. ISBN 978-1-60558-134-7. 8.1.3

Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009. 8.1.3

Peter Henderson and James H. Morris Jr. A Lazy Evaluator. In *POPL*, pages 95–103, 1976. 8.1.3

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. 3.3

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*, pages 311–320, 2000. 8.1.2

Roger Hoover. *Incremental Graph Evaluation*. PhD thesis, Department of Computer Science, Cornell University, May 1987. 8.1.2

R. J. M. Hughes. Lazy memo-functions. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, 1985. 8.1.3

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. 5.2

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. 6.4

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A Cost Semantics for Self-Adjusting Computation. Technical Report CMU-CS-08-141, Department of Computer Science, Carnegie Mellon University, July 2008a. 2.3.2

Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling Self-Adjusting Programs with Continuations. In *Proceedings of the International Conference on Functional Programming*, 2008b. 3, 4, 5, 5.3.2, 6, 7, 8.1.3

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009. 2.3.2, 3, 4, 5, 7, 8.1.3, 8.1.4

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. 8.1.2

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. 8.1.2

Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a Programming Language for Ajax Applications. In *OOPSLA*, pages 1–20, 2009. 8.1.5

D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, 1968. 4.4.1, 8.1.2, 8.1.3

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised)*. The MIT Press, 1997. 6.1, 6.2

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. 5.2

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632. 8.1.3

F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag Inc., 1985. 7.2.1

William Pugh. *Incremental Computation via Function Caching*. PhD thesis, Department of Computer Science, Cornell University, August 1988. 8.1.2

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. 8.1.2, 8.1.2

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. 8.1.2

Thomas Reps. Optimal-time Incremental Semantic Analysis for Syntax-directed Editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 169–176, 1982. 8.1.2

Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, New York, NY, USA, 1989. ACM. 8.1.4

Daniel Russel. *Kinetic Data Structures in Practice*. PhD thesis, Department of Computer Science, Stanford University, March 2007. 8.1.1

Daniel Russel, Menelaos I. Karavelas, and Leonidas J. Guibas. A package for Exact Kinetic Data Structures and Sweepline Algorithms. *Comput. Geom. Theory Appl.*, 38 (1-2):111–127, 2007. ISSN 0925-7721. 8.1.1

David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, September 1990a. 2.2.1, 2.2.2, 8.1.4

David Sands. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*, pages 361–376, London, UK, 1990b. Springer-Verlag. 2.2.1, 2.2.2, 8.1.4

Patrick M. Sansom and Simon L. Peyton Jones. Time and Space Profiling for Non-strict, Higher-order Functional Languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 355–366, 1995. 2.2.2, 8.1.4

Marco D Santambrogio, Vincenzo Rana, Seda Ogrenci Memik, Umut A. Acar, and Donatella Sciuto. A novel SoC design methodology for combined adaptive software descripton and reconfigurable hardware. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2007. 1.1, 8.1.3

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. 1.1, 8.1.3

Daniel Dominic Sleator and Robert Endre Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985. 8.1.1

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space Profiling for Parallel Functional Programs. In *Proceedings of the International Conference on Functional Programming*, 2008. 2.2.2, 8.1.4

R. S. Sundaresh and Paul Hudak. Incremental Compilation via Partial Evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, 1991. 8.1.2

Hayo Thielecke. Comparing Control Constructs by Double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):367–412, 2002. 6.4

Jean Vuillemin. Correct and Optimal Implementations of Recursion in a Simple Programming Language. *J. Comput. Syst. Sci.*, 9(3):332–354, 1974. 8.1.3

Philip Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 385–407, London, UK, 1987. Springer-Verlag. 2.2.1

Zhanyong Wan and Paul Hudak. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252. ACM, 2000. 8.2.2

Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *SIGPLAN Not.*, 36(10): 146–156, 2001. ISSN 0362-1340. 8.2.2

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. 8.1.2