Self-Adjusting Computation

Umut A. Acar

May 2005
CMU-CS-05-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy Blelloch, co-chair
Robert Harper, co-chair
Daniel Dominic Kaplan Sleator
Simon Peyton Jones, Microsoft Research, Cambridge, UK
Robert Endre Tarjan, Princeton University

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

This research was sponsored in part by the National Science Foundation under grant CCR-0085982, CCR-0122581 and EIA-9706572, and the Department of Energy under contract no. DE-FG02-91ER40682. 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, dynamic algorithms, dynamic data structures, kinetic data structures, dynamic dependence graphs, memoization, change propagation, trace stability, functional programming, lambda calculus, type systems, operational semantics, modifiable references, selective memoization, sorting, convex hulls, parallel tree contraction, dynamic trees, rake-and-compress trees.

Abstract

This thesis investigates a model of computation, called *self-adjusting computation*, where computations adjust to any external change to their data (state) automatically. The external changes can change any data (*e.g.*, the input) or decisions made during the computation. For example, a self-adjusting program can compute a property of a dynamically changing set of objects, or a set of moving objects, *etc.* This thesis presents algorithmic and programming-language techniques for devising, analyzing, and implementing self-adjusting programs.

From the algorithmic perspective, we describe novel data structures for tracking the dependences in a computation and a change-propagation algorithm for adjusting computations to changes. We show that the overhead of our dependence tracking techniques is $O(1)$. To determine the effectiveness of change propagation, we present an analysis technique, called *trace stability*, and apply it to a number of applications.

From the languages perspective, we describe language facilities for writing self-adjusting programs in a type-safe and correct manner. The techniques make writing self-adjusting programs nearly as easy as ordinary (non-self-adjusting) programs. A key property of the techniques is that they enable the programmer to control the cost of dependence tracking by applying it selectively. Using language techniques, we also formalize the change-propagation algorithm and prove that it is correct.

We demonstrate that our techniques are efficient both in theory and in practice by considering a number of applications. Our applications include a random sampling algorithm on lists, the quick sort and merge sort algorithms, the Graham's Scan and the quick algorithm for planar convex hull, and the tree-contraction algorithm. From the theoretical perspective, we apply trace stability to our applications and show complexity bounds that are within an expected constant factor of the best bounds achieved by special-purpose algorithms. From the practical perspective, we implement a general purpose library for writing self-adjusting programs, and implement and evaluate self-adjusting versions of our applications. Our experiments show that our techniques dramatically simplify writing self-adjusting programs, and can yield very good performance even when compared to special-purpose algorithms, both in theory and in practice.

For my parents Dürdane and İsmail Acar

Acknowledgments

This thesis would not have been possible without the support and encouragement of my advisors Guy Blelloch and Robert Harper. Many of the results in this thesis have come out of long conversations and meetings with Bob and Guy. I thank you both very much.

I thank Simon Peyton Jones and Bob Tarjan for supporting my research endevours and for giving me feedback on the thesis. Simon's feedback was critical in tying the different parts of the thesis together. Bob's feedback helped simplify the first three parts of the thesis and started us thinking about some interesting questions.

My colleague Maverick Woo helped with the proofs in the third part of the thesis, especially on the stability of tree contraction. Maverick also uncovered a body of related work that we were not aware of.

I thank Renato Werneck (of Princeton) for giving us his code for the Link-Cut trees, and for many discussions about dynamic-trees data structures.

I thank Jernej Barbic for his help with the graphics library for visualizing kinetic convex hulls.

I have had the chance to work with bright young researchers at CMU. Jorge Vittes (now at Stanford) did a lot of the hard work in implementing our libraries for kinetic data structures and for dynamic trees. Kanat Tangwongsan helped implement key parts our SML library and helped with the experiments. I thank Jorge and Kanat for their enthusiasm and dedication.

Matthias Blume (of Toyota Technological Institute) helped with the implementation of the SML library. Matthias and John Reppy (of the University of Chicago) helped in figuring out the intricacies of the SML/NJ's garbage collector.

Many colleagues and friends from CMU enriched my years in graduate school. I thank them for their friendship: Konstantin Andreev, Jernej Barbic, Paul Bennett, Mihai Budiu, Armand Debruge, Terry Derosia, Derek Dreyer, Jun Gao, Mor Harchol-Balter, Stavros Harizopolous, Rose Hoberman, Laurie Hiyakumoto, Yiannis Koutis, Gary Miller, Aleks Nanevski, Alina Oprea, Florin Oprea, Lina Papadaki, Spriros Papadimitrou, Sanjay Rao, Daniel Spoonhover, Srinath Sridhar, Mike Vandeweghe, Virginia Vassilevska, Maverick Woo, Shuheng Zhou.

Outside CMU, I thank Yasemin Altun (of TTI), Görkem Çelik (of UBC), Thu Doan (of IBM Austin), Burak Erdogan (of UIUC), Ram Mettu (of Dartmouth College) for their friendship and for the great times biking, rock-climbing, snowboarding, or just enjoying. I had great fun hanging out with the RISD crowd; thanks to you Jenni Katajamäki, Celeste Mink, Quinn Shamlian, Amy Stein, and Jared Zimmerman.

I thank my family for their unwavering support. My parents Dürdane and İsmail, my brother Uğur, and my sister Aslı have always been there when I needed them. Finally, I thank my wife Melissa for her love and support over the years and for help in proofreading this thesis.

Contents

List of Figures

Chapter 1

Introduction

This thesis investigates a model of computation, called *self-adjusting computation*, where computations adjust to any external change to their data (state) automatically. The external changes can take any form. For example, they can change the input or the outcome of the decisions made during the computation. As an example, consider the convex-hull problem from computational geometry. This problem requires finding the convex hull of a given set of points, *i.e.* smallest polygon enclosing the points. A self-adjusting convex-hull computation, for example, enables the user to insert/delete points into/from the input while it adjusts the output to that change. Like a dynamic convex-hull algorithm [75, 69, 20], this computation can compute the convex hull of a dynamically changing set of points. Similarly a self-adjusting convex-hull computation enables the user to change the outcome of a computed predicate. Like a kinetic convex-hull data structure[13], this computation can be used to compute the convex-hull of a set of points in motion.

The problem of adjusting computations to external changes has been studied extensively in both the algorithms community and the programming-languages community.

The algorithms community has focused on devising so-called *dynamic algorithms (or data structures)* and *kinetic algorithms (or data structures)* that would give the fastest update times under external changes. Although dynamic/kinetic algorithms often achieve fast, even optimal, updates they have some important limitations that make them difficult to use in practice.

Inherent complexity: Dynamic algorithms can be very difficult to devise, even for problems that are easy in the static case (when the input does not change). Furthermore, dynamic algorithms are often very sensitive to the definition of the problem. For example, a problem can be easy for certain kinds of changes, but very difficult for others; or a slight extension to the problem can require substantial revisions to the algorithm.

As an example consider the problem of computing the minimum-spanning tree (MST) of a graph. In the static case (when the graph does not change) giving an algorithm for computing MSTs is easy. When the graph is changed by insertions only, it is also easy to give an algorithm for MSTs. When the graph is changed by deletions, however, the problem becomes very difficult. Indeed, it took nearly two decades [36, 33, 45, 46, 49] to devise an efficient solution to this problem. Many dynamic algorithms exhibit this property. Another example is the dynamic trees problem of Sleator and Tarjan [91] whose many variants have been studied extensively [91, 92, 23, 84, 46, 96, 10, 38, 11, 95, 6].

Lack of support for broad set of changes: Most dynamic algorithms are designed to support one insertion and/or one deletion at a time. More complex changes, such as *batch changes* where multiple insertions and deletions can take place simultaneously, must be handled separately. Similarly, kinetic data structures are often designed to support one *kinetic change* that changes the outcome of a predicate, but not insertions and deletions, or batch changes [43, 13]. As Guibas points out [43], these are important limitations. To address these limitations, Guibas suggests that "the interaction between kinetic and dynamic data structures needs to be better developed" [43].

Lack of support for composition: Dynamic and kinetic data structures are not composable—it is not possible to take two dynamic and/or kinetic algorithms and feed the output of one into the other. The key reason for this is that dynamic algorithms cannot detect change; rather, they assume that changes take place only through a predefined set of operations. Another reason is that through composition, a small change can propagate to more complex changes. For example, to be composable, kinetic data structures must support insertions and deletions [43]—not just kinetic changes.

These limitations make it difficult to use dynamic/kinetic algorithm in practice. In practice solving a problem often requires adapting an existing algorithm to the needs of the particular problem, extending the algorithm for different kinds of changes, and composing algorithms. Because of the limitations of existing dynamic/kinetic algorithms and the inherent complexity of these problems, it is not reasonable to expect the practitioner to employ dynamic/kinetic algorithms.

The programming language community has focused on developing general-purpose techniques for transforming static (non-dynamic) programs into dynamic programs. This is known as *incremental computation*. Previous work on incremental computation addresses some of the limitations of the approach preferred in the algorithms community. In particular, incremental computation helps reduce the inherent complexity problem by providing programming abstractions. The techniques often support a broad range of changes and support composition. However, all previously proposed incremental computation techniques have a critical limitation: they either apply only to certain restricted classes of computations, or they deliver suboptimal, often inadequate, performance.

The most successful incremental computation techniques are based on static dependence graphs and memoization (or function caching). Static dependence graphs, introduced by Demers, Reps, and Teitelbaum [29], can provide efficient updates. The key problem with the technique is that it is applicable only to certain class of computations [81]. Memoization, first applied to incremental computation by Pugh and Teitelbaum [83], is a general-purpose technique. The key problem with memoization is that it is not efficient. For example, the best bound for incremental list sorting with memoization is linear [57].

This thesis develops general-purpose techniques for self-adjusting computation. The techniques enable writing self-adjusting programs like ordinary programs. In particular, an ordinary program can be transformed into a self-adjusting program by applying a methodical transformation. Self-adjusting programs support any change to their state whatsoever, and they are composable. The techniques have $O(1)$-time overhead and yield self-adjusting programs that are efficient both in theory and in practice. We achieve our results by combining algorithmic and programming-languages techniques. We show that our techniques are effective in practice by implementing and evaluating them.

From the algorithmic perspective, we introduce efficient data structures for tracking dependences in a computation and a change-propagation algorithm for adjusting computations to external changes. We prove that the overhead of our dependence tracking techniques is $O(1)$. To bound time for change-propagation, we present an analysis technique, called *trace stability*. Trace stability enables bounding the time for change propagation in terms of the distance between computation traces. To determine the effectiveness of change-propagation, we consider a number of applications including

- a random-sampling algorithm for combining the elements of a list,
- the quick sort and the merge sort algorithms,
- the Graham's Scan [42] and the quick hull algorithms for planar convex hulls,
- the parallel tree-contraction algorithm of Miller and Reif [62, 63], and
- kinetic data structures.

These applications are chosen to span a number of problem solving paradigms such as random-sampling, divide-and-conquer, incremental result construction, and parallel, bottom-up result construction. Using trace stability, we show complexity bounds for these applications (except for kinetic data structures) under insertions and deletions to their input. The trace-stability results yield upper bounds that are within an expected constant factor of the best bounds achieved by special-purpose dynamic algorithms.

From the programming-languages perspective, we present techniques for writing self-adjusting programs. Using our techniques, the programmer can transform an ordinary program into a self-adjusting program by applying a methodical transformation technique. The key properties of the techniques are that

- they are general purpose,
- they enable applying dependence tracking selectively,
- they yield type safe programs,
- they yield correct self-adjusting programs, and
- they accept an efficient implementation.

Our language techniques apply to all programs both purely functional, and imperative. Selective dependence tracking enables the programmer to identify parts of the computation data as *changeable*, *i.e.*, can be affected by an external change, and other parts as *stable*. The techniques then track only the dependences pertaining to changeable data. This enables the programmer to control the cost of dependence tracking by controlling the granularity of dependence tracking. We employ type theory to ensure that type-safe programs written in the language are safe, *i.e.*, do not "go wrong" during execution.

A key property of our techniques is that they yield correct self-adjusting programs, *i.e.*, self-adjusting programs adjust to any permissible change correctly. To study correctness of our techniques, we formalize our change-propagation algorithm and memoization techniques, and show that they are correct. To prove that change propagation is correct, we show that the algorithm and a from-scratch execution are semantically equivalent (performance, of course, is different).

To study the practical effectiveness of self-adjusting computation, we perform an experimental evaluation in two contexts. On the one hand, we implement a general-purpose ML library for writing self-adjusting programs, and present an experimental evaluation based on our applications. Our experiments confirm our asymptotic complexity bounds and show that the constant factor hidden in our $O(1)$ overhead bound is small. For the input sizes that we consider, we measure up to three orders of magnitude time difference between self-adjusting programs and recomputing from scratch—since the gap is asymptotically significant, the gap grows with the input size. On the other hand, we implement a specialized version of our library for dynamic-trees problem of Sleator and Tarjan [91]. We solve the dynamic-trees problem by applying self-adjusting computation to the tree-contraction algorithm of Miller and Reif [62] and perform an extensive experimental evaluation by considering a broad range of applications. When applicable, we compare our implementation to Werneck's implementation of the fastest dynamic-trees data structure [98], known as Link-Cut Tress [91, 92]. Our experiments show that for certain operations, Link-Cut trees are faster; for other operations, self-adjusting tree contraction is faster. We note that self-adjusting tree contraction supports a broader set of changes and applications than Link-Cut trees. The experimental results show that self-adjusting computation performs very well even when compared to special-purpose algorithms.

1.1 Overview and Contributions of this Thesis

This section presents an overview of the thesis and describes its key contributions. The thesis consists of five parts: algorithms and data structures, trace stability, applications, language techniques, and implementation and experiments. Some of the work reported in this thesis has been published in conference proceedings and journals in the areas of algorithms [6, 5] and programming languages [2, 4, 3].

1.1.1 Part I: Algorithms and Data Structures

We introduce the key data structures and algorithms that self-adjusting computation relies on. The data structures, called *dynamic dependence graphs (DDGs)* and *memoized dynamic dependence graphs (MDDGs)*, are described in Chapters 4 and 5 respectively. In Chapter 6 we present complexity bounds for the overhead and the effectiveness of these data structures.

Dynamic Dependence Graphs (DDGs)

We introduce dynamic dependence graphs (DDGs) for representing computations and provide a change-propagation algorithm for them. Given the DDG of a computation and any change to the state of the computation, the change-propagation algorithm adjusts the computation to that change. In Chapter 6, we show that the DDG of a computation can be constructed with $O(1)$ overhead.

Memoized Dynamic Dependence Graphs (MDDGs)

We point out the limitations of DDGs and address these limitations by introducing Memoized Dynamic Dependence Graphs (MDDGs) and *the memoized change-propagation algorithm* for them. Given the MDDG of a computation, and any change to the state of the computation, the memoized change-propagation algorithm adjusts the computation to that change. A key property of memoized DDGs is that they enable memoization under side-effects (mutations) to the memory. In Chapter 6, we show that the MDDG of a computation can be constructed with $O(1)$ overhead and present a bound on the complexity of memoized change propagation.

1.1.2 Part II: Trace Stability

Although the complexity bound for change propagation developed in the first part of this thesis applies to all programs, it is somewhat unsatisfactory. This is because it relies on a low level understanding of memoized dynamic dependence graphs. In this part of the thesis, we develop a higher level analysis technique for determining the complexity of change propagation. The analysis technique called *trace stability*, relies on representing computations with traces. To determine the complexity of change propagation, it suffices to measure the distance between two traces. In particular, if the trace distance between two computations is $O(f(n))$ (for some measure n), then we show that change propagation can transform one computation into the other in $O(f(n)\log f(n))$ time, in general, and in $O(f(n))$ time in certain cases. Chapter 7 defines the notions of traces and trace distance. Chapter 8 defines and proves our trace stability theorems.

1.1.3 Part III: Applications

We consider a number of algorithms on lists (Chapter 9) and an algorithm on trees (Chapter 10), and show trace stability bounds for them. The list algorithms include a random-sampling algorithm for combining the values in a list, the merge sort and the quick sort algorithms, the Graham's Scan and quick hull algorithms for planar convex hulls. For trees, we consider the tree-contraction algorithm of Miller and Reif [62]. These algorithms are chosen to span a number of computing paradigms such as random sampling, divide-and-conquer, incremental results construction, and parallel, bottom-up result construction.

All our bounds are for insertions/deletions to the input and are parameterized by the size of the input. For combining values in a list, we show an expected $O(\log n)$ bound, where the expectation is taken over internal randomization of the algorithm. A self-adjusting version of this algorithm can compute the minimum, maximum, sums, *etc.*, of a list in expected $O(\log n)$ time under insertions/deletions. For the quick sort algorithm we show an expected $O(\log n)$ bound, where the expectation is taken over all possible positions of insertions/deletions into/from the input. For the randomized merge sort algorithm that uses random splitting, we show an expected $O(\log n)$ bound, where the expectation is taken over the internal randomization of the algorithm. For the Graham's Scan algorithm we show an expected $O(1)$ stability bound for line-side tests under insertions/deletions into the list—the expectations are taken over all points in the input. For the tree-contraction algorithm, we show an expected $O(\log n)$ stability bound under edge insertions/deletions. Using self-adjusting tree contraction, the dynamic-trees problem can be solved in expected $O(\log n)$ time.

All of the stability bounds directly yield complexity bounds for insertions/deletions. These bounds are within an expected constant factor of the best bounds achieved by special-purpose dynamic algorithms.

1.1.4 Part IV: Language Techniques

In this part of the thesis, we present language techniques for writing self-adjusting programs. The techniques make it possible to transform an ordinary (non-self-adjusting program) into a safe, correct, and efficient self-adjusting program by making small, methodical changes to the code.

Adaptive Functional Programming and the AFL Language

Adaptive functional programming (Chapter 11) enables writing self-adjusting programs based on (non-memoized) dynamic dependence graphs. A key property of the techniques is that it enables the programmer to apply DDG-based techniques selectively. We study the techniques in the context of a purely functional language called the *Adaptive Functional Language (AFL)*. We give a static and dynamic semantics for the language and show that the language is type safe. Based on the dynamic semantics, we prove that the change-propagation algorithm on dynamic dependence graphs is correct.

Selective Memoization and the MFL Language

Selective memoization (Chapter 12) enables the programmer to control the effectiveness of memoization by providing control over the cost of equality tests, the input-output dependences, and space consumption. The most important aspect of selective memoization is the mechanisms for supporting memoization with precise-input output dependences. We study the technique in the context of a function language, called *Memoizing Functional Language (MFL)*. We present the static and dynamic semantics for the language. We prove that the language is type-safe and the result re-use technique with precise dependences is correct.

Self-Adjusting Functional Programming and the SLf Language

Self-adjusting functional programming (Chapter 13) enables writing self-adjusting programs based memoized dynamic dependence graphs. We study the techniques in the context of a purely functional language, called *Self-adjusting functional Language (SLf)*. The key property of the SLf language is that it supports efficient change propagation. In particular, SLf enables matching the change-propagation bounds obtained in the algorithmic setting. Note that these bounds rely on an imperative computation model that provides explicit management of memory. To achieve this, the SLf language combines the AFL and the MFL languages and extends them with support for *non-strict* and *strict dependences*. We demonstrate the effectiveness of SLf by implementing it as an ML library and presenting an experimental evaluation based on our applications.

Imperative Self-Adjusting Programming and SLi language

We present an imperative language, called SLi, for writing self-adjusting programs (Chapter 14). The language extends the SLf language with constructs to enable memory locations to be written multiple times. This shows that the techniques presented here apply beyond the domain of single-assignment or purely functional programs.

1.1.5 Part V: Implementation and Experiments

In this part of thesis, we study the practical effectiveness of self-adjusting computation in the context of a general-purpose ML library (Chapter 15), and an implementation of self-adjusting tree contraction in the C++ language (Chapter 17). Based on our general-purpose library we implement a library for transforming ordinary programs into kinetic programs and show snapshots of a kinetic simulation based on our techniques (Chapter 16).

A Library for Self-Adjusting Computation

We show that our techniques are practical by implementing an ML library to support self-adjusting computation, implementing our list applications based on this library, and presenting an experimental evaluation (Chapter 15). Our experiments show that the overhead of our library over non-self-adjusting programs is between four and ten, and the overhead of change propagation is no more than six. The experiments confirm the theoretical bounds obtained by trace-stability analysis. In our experiments, we observe that self-adjusting programs are up to three orders of magnitude faster than recomputing from scratch for the input sizes that we consider (due to asymptotic difference, the gap can be arbitrarily large).

Kinetic Data Structures

In Chapter 16, we describe a library for transforming ordinary programs into kinetic programs. The transformation involves two steps. In the first step, the programmer transforms the ordinary program into a self-adjusting program. In the second step, the comparisons employed by the program are replaced by kinetic comparisons test. The second step is trivial and can easily be performed automatically. As an example, we implement kinetic geometry tests for computing convex hulls and apply the technique to our self-adjusting convex hull algorithm. We show some snapshots of our kinetic simulations. Our work addresses two key limitations of previous work. Our kinetic programs directly support insertions/deletions, changes to the motion parameters (flight-plan updates), and batch changes. Furthermore, our programs are directly composable.

Self-Adjusting Computation in Dynamic Trees

In Chapter 10, we solve the dynamic-trees problem of Sleator and Tarjan [91] by applying self-adjusting computation to the tree-contraction algorithm of Miller and Reif [62]. In Chapter 17, we present an implementation of our solution in the C++ language and perform an extensive experimental evaluation. When possible, we compare our implementation to an implementation of the fastest dynamic-trees data structure, known as Link-Cut Tress [91, 92]. Our experiments show that for certain external changes self-adjusting tree contraction is up to five time slower than Link-Cut Trees, whereas for certain other changes and queries, it is up to three times faster. It is also important to note that self-adjusting tree contraction supports a broader range of applications and a broader range of input changes than LC-Trees. For example, self-adjusting tree contraction supports subtree queries directly, whereas LC-Trees require a sophisticated extension for subtree queries [23, 84]. Similarly, self-adjusting tree contraction supports batch, *i.e.* multiple simultaneous, changes directly, whereas LC-Trees only support one-by-one processing of changes. These results demonstrate that self-adjusting computation performs well even compared to special-purpose algorithms.

Chapter 2

Related Work

The problem of adjusting a computation to external changes has been studied extensively in both the algorithms/theory community and the programming-languages community. This chapter reviews the previous work, discusses its limitations, and how it relates to this thesis. Section 2.1 reviews the work of the algorithms and the theory community, Section 2.2 reviews the work of the programming-languages community.

2.1 Algorithms Community

The related work in the algorithms community has focused on devising dynamic algorithms (or data structures) and kinetic algorithm (or data structures). This section discusses previously proposed design and analysis techniques (Section 2.1.1), and discusses the limitations of these approaches (Section 2.1.2). Section 2.1.3 discusses the relationship between the work in this thesis and the previous work and how our work addresses the limitations of the previous work.

A *dynamic algorithm (or data structure)* computes a property of its input while allowing the user to change the input. For example, a typical dynamic convex hull algorithm computes the convex hull of a set of points while also allowing the user to insert and delete points to the input. An enormous amount work has been done on dynamic algorithms. The problems that are considered in this thesis have been studied extensively since the early eighties [17, 74, 75, 91, 92, 67, 69, 68, 89, 21, 37, 38, 10, 32, 46, 49, 95]. Dynamic algorithms have also been studied extensively within particular fields. For examples of dynamic algorithms in graphs, and in computational geometry, we refer the interested reader to the papers by Eppstein, Galil, and Italiano [32], and Chiang and Tamassia [21].

Kinetic data structures[13] compute properties of continuously moving objects. For example, a kinetic convex hull algorithm computes (or maintains consistent) the convex hull of a set of points as they move continuously in the plane. By using the continuity of motion, kinetic data structures compute a discrete event space, where an event corresponds to a change in the result of a predicate. Every time an event takes place, the data structure is adjusted to that event by using techniques specific to the problem. For other examples of kinetic data structure, we refer the interested reader to other papers [14, 43, 7, 55, 25, 54].

2.1.1 Design and Analysis Techniques

Most of the fastest dynamic algorithms rely on techniques that exploit the particular structure of the problem considered [91, 92, 37, 38, 10, 46, 49, 95]. Other dynamic algorithms are constructed from static (non-dynamic) algorithms by applying somewhat more general design principles; this is known as dynamization. These techniques include rebuilding and decomposition strategies, and explicit maintenance of the history of computations.

Rebuilding strategies [74, 75, 21] rely on maintaining a static data structure and allowing dynamic updates to violate the invariants of the static data structure for shorts periods of time. Periodically, the data structure is rebuilt to reinforce the invariants. By controlling the number of operations in between periodic rebuilds, and the cost of rebuilding, these techniques can handle dynamic changes reasonably efficiently for certain applications [75, 69, 89, 21].

Decomposition strategies rely on decomposing a problem into smaller problems that are solved using some static algorithm. When a dynamic update occurs, the parts of the problem affected by the update are reconstructed using the static algorithm. The idea is that a small change to the input will affect a small number of the subproblems. This idea dates back at least to the 70s. Bentley and Saxe [17] and Overmars [74, 75] described how saving partial results can be used to efficiently dynamize any algorithm that belong to certain classes of computations. Overmars used the approach, for example, to design a dynamic algorithm for convex hulls that supports insertions and deletions of points in $O(\log^2 n)$ time.

Another approach to dynamization relies on maintaining the history of a computation and updating the history when the input changes. Intuitively, the history is thought of as the trace of the execution of some static, often incremental, algorithm. The update is performed by going back in time and propagating the change forward to reconstruct the history. Mulmuley [67, 69, 68, 70], and Schwarzkopf [89] apply this technique to a number of computational geometry problems In this approach, history and propagation are devised, analyzed, and implemented on a per-problem basis.

Kinetic data structures [43, 13] rely on an approach similar to the history maintenance. In a kinetic data structure, the history of a computation consists of some representation of the predicates determined by the execution of a static (non-kinetic) program. Kinetic data structures simulate motion by determining the time for the earliest predicate failure, and rebuilding the history according to the new value of the failed predicate(s). As with the work of Mulmuley and Schwarzkopf, the structure of the history and the algorithm for rebuilding the history are devised and implemented on a per problem basis.

2.1.2 Limitations

The focus of the algorithms community has been to devise fast dynamic/kinetic algorithms. Indeed, many of the devised algorithms achieve optimal or near optimal update times under external changes. The resulting algorithms, however, have important limitations.

Inherent Complexity

Dynamic algorithms can be very difficult to devise, even for problems that are easy in the static case (when the input does not change). Furthermore, dynamic algorithms are often very sensitive to the input changes considered and to the definition of the problem. A problem can be easy for certain kinds of changes and very difficult for others, or a slight extension to the problem can require substantial, sophisticated revisions to the algorithm.

As an example consider the problem of computing the minimum-spanning tree (MST) of a graph. In the static case, *i.e.*, when the graph does not change, giving algorithm for computing MSTs is easy. Similarly, when the graph is restricted to be changed only by insertions, it is easy to give algorithm for MSTs. When the graph changes by insertions and deletions, however, the problem becomes very difficult. Indeed, it took nearly two decades of research [36, 33, 45, 46, 49] to devise an efficient solution to this problem.

As another example consider the dynamic trees problem [91]. Since Sleator and Tarjan proposed and solved the problem by proposing Link-Cut trees data structure, much research has been done on solving various variations of the problem [91, 92, 23, 84, 46, 96, 10, 38, 11, 95, 6]. A look through these papers should convince the reader that these data structures are very complex and differ significantly from each other, even though they all address variations of the original problem.

Real-world applications often require extending/modifying an existing algorithm to support the needs of an application. Because of the inherent complexity of dynamic algorithms, it is not reasonable to expect a practitioner or a non-expert to extend an existing dynamic/kinetic algorithm. The dynamic trees example mentioned above is an example to this. A look through previously proposed dynamic-trees data structures should convince the reader that a practitioner (even a non-expert) in this area cannot possibly adapt an existing solution for the problem easily, let alone implement it correctly.

Lack of Support for Broad Set of Changes

A key limitation of dynamic algorithms is that they are designed to support a small predefined set of changes to the input. For example, almost all algorithms in the literature support a single insertion or deletion to their input. To be practical, however, a dynamic algorithm or kinetic data structure must support a broader set of changes. This is because 1) in practice applications require support for a broad set of changes, 2) arbitrary changes can arise due to composition of dynamic/kinetic algorithms.

In many application domains, it is important to support multiple simultaneous changes as a batch. For example, the user can make multiple changes at a time, or multiple changes can arise within a dynamic algorithm itself. For example, the dynamic connectivity and the Minimum-Spanning Tree algorithms of Holm *et. al.* [49], maintain a hierarchy of dynamic trees. When a single edge is deleted from the input graph, a number of edges in one level of the hierarchy can be inserted into the next level. This data structure can benefit significantly from a dynamic-tree data structure that supports batch changes. As Guibas points out [43], in kinetic data structures, multiple events can take place in a small time interval. In such cases, it is desirable to process all events together as a batch [43]. Why composition requires support for arbitrary changes is discussed in the next paragraph.

Lack of support for composition

When building practical systems, it is often necessary to compose algorithms, *i.e.*, to run an algorithm on the output of another algorithm. Composition is arguably very important to computation. Algorithms use other algorithms as sub-steps. Unfortunately it is not possible to combine dynamic/kinetic algorithms, because these algorithms

1. cannot detect an external change—rather, they assume that an external change takes place only through a set of operations that they provide, and

2. support only a narrow, predefined set of changes.

Consider for example, a program that counts the number of points in the convex hull of set of points. One way to write such a program is to compute the convex hull first, and call a `length` function on the hull. Suppose inserting a single point to the input of this program. Since the convex hull algorithm is dynamic it will update the convex hull accordingly. One problem is that the `length` function cannot detect the change in the convex hull (its input). It must be given the change, but the dynamic convex hull algorithm does not provide such information. To solve this problem, the programmer must either eliminate composition by changing the internals of the convex hull program to have it output the length of the hull. Or the programmer must devise some way of computing the change in the hull. The second problem is that the `length` function will in general see an arbitrary number of changes to its input. This is because inserting one point to the input can change the convex hull dramatically. Since dynamic algorithms are not designed to process arbitrary changes, the programmer needs to implement a mechanism for splitting up large changes into smaller changes. To the best of our knowledge, no previous work that addresses the issues of composition.

Similarly, when composing kinetic data structures, a kinetic change can propagate to an insertion or deletion, or a change to the motion parameters of objects. To be composable kinetic data structures must be combined with dynamic algorithms [43]. Guibas points out that composition is critical and suggests that the "interaction between kinetic and dynamic data structures needs to be better developed" [43].

2.1.3 This Thesis

In terms of design techniques, the approaches based on maintaining a history of the computation as proposed by Mulmuley [67, 69, 68, 70], and Schwarzkopf [89], and as used in kinetic data structures [43, 13] are closest to the approach presented in this thesis. As with these proposal, we maintain a history of a computation and update the computation by propagating a change through the history. The key difference between our approach and the previously proposed approaches is that, in our approach, the notion of history is defined for general computations. We construct the history automatically as a static (non-dynamic, non-kinetic) program executes, and keep it up to date by using a general-purpose change propagation algorithm.

Our approach does not have the limitations discussed the previous section. Self-adjusting programs can be obtained from ordinary programs by applying a methodical transformation. This addresses the inherent-complexity problem by dramatically simplifying the implementation of self-adjusting programs. For example, it generally takes the author one hour to transform an ordinary planar convex hull algorithm into a self-adjusting convex hull algorithm. Self-adjusting computations adjusts to any external change, including batch changes, or any combination of kinetic and dynamic changes automatically, and correctly. Furthermore, self-adjusting programs are composable.

2.2 Programming Languages Community

In the programming-languages community, the related work has been done in the area of *incremental computation*. For a broad set of references on incremental computation, we refer the interested reader to Ramalingam and Reps's bibliography [85]. Also previous theses on incremental computation all give a good account of related work [87, 50, 81, 34, 56]. Much of the previous work on incremental computation can be broadly divided into three classes based on the approach. The rest this section reviews each approach separately and discusses their limitations. Section 2.2.4 describes how our work addresses the limitations of the previously proposed techniques.

2.2.1 Static Dependence Graphs

Static-dependence-graph techniques rely on representing a computation in the form of a circuit. When the input to the computation changes, the change is propagated through the circuit by recomputing the values affected by the change. Demers, Reps and Teitelbaum [29], and Reps [87] introduced the idea of using dependence graphs for incremental computation in the context of attribute grammars. Reps then showed an algorithm to propagate a change optimally through the circuit [88], and Hoover generalized the techniques outside the domain of attribute grammars [50]. Yellin and Strom applied the static-dependence-graph ideas within the INC language [99], and extended it by having incremental computations within the array primitives. The main limitation of the INC language is that it is not general-purpose—it does not support recursion or looping.

The chief limitation of static-dependence-graph techniques is that the dependence graphs are *static*: the dependence structure remains the same during change propagation. As Pugh points out [81], this severely limits the effectiveness of this approach, because it is undecidable to determine how the dependences of an arbitrary program changes without executing the program. For certain classes of computations, such as *syntax-directed* computations, however, it is possible to update the dependences by exploiting the structure of the computation.

2.2.2 Memoization

Memoization, or function caching, [15, 61, 60] is based on the idea of caching the results of each function call indexed by the arguments to that call. If a function is called with the same arguments a second time, the result from the cache is re-used and the call is skipped. Since in general the result of a function call may not depend on all its arguments, caching results based on precise input-output dependences can dramatically improve result re-use. Abadi, Lampson, Levy [1] and Heydon, Levin, Yu [47] introduced techniques for memoizing function call based on precise input-output dependences.

Pugh [81], and Pugh and Teitelbaum [83] were the first to apply memoization to incremental computation. They were motivated by the lack of a general-purpose approach to incremental computation—previously proposed techniques based on static-dependence-graphs apply only to a restricted class of applications. They developed techniques for implementing memoization efficiently and studied incremental algorithms using memoization. They showed complexity bounds on incremental computation for certain class of divide-and-conquer algorithms based on so called *stable decompositions*.

Unlike static dependence graphs, memoization is general. Any purely functional program can be made incremental using memoization. The effectiveness of memoization, however, crucially depends on the management of memo tables, or function caches. For efficiency it is critical to evict elements from the cache. It is difficult, however, to decide what elements should be kept and what elements should be evicted from the cache. Liu and Teitelbaum made some progress on this problem by using program-analysis techniques [59, 56]. Their techniques automatically determine what results should be memoized and use transformations to make programs incremental.

The main limitation of memoization is its effectiveness. For memoization to be effective, making a small change to the input must affect only a small number of functions calls. Although this may sound to be true at an intuitive level, it is not. In fact, for many applications, most changes will affect a large number of function calls. Consider for example, a program that computes some property of a list by recursively walking down the list. Imagine now changing the input list by inserting a new element in the middle. Since the new element is in the input to half of all the recursive calls, many recursive calls will not find their result in the memo—the update will take linear time on average. Similar scenario's arise in many applications. For example, the best asymptotic bound for sorting with memoization is linear [56].

2.2.3 Partial Evaluation

Other approaches to incremental computation are based on partial evaluation [35, 94]. Sundaresh and Hudak's approach [94] requires the user to fix the partition of the input that the program will be specialized on. The program is then partially evaluated with respect to this partition and the input outside the partition can be changed incrementally. The main limitation of this approach is that it allows input changes only within a predetermined partition [56, 34]. Field [34], and Field and Teitelbaum [35] 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 use partially evaluated results optimally.

2.2.4 This Thesis

Like previous work on incremental computation, our goal is to develop techniques for transforming ordinary programs to programs that can adjust to external changes efficiently. Compared to previous work, our techniques take important steps towards achieving this goal. The key difference between our work and previously proposed techniques are that

- our techniques are general purpose,
- the computational complexity of self-adjusting programs can be determined using analytical techniques, and
- our techniques are efficient both in theory and in practice.

One of the main contributions of this thesis is a data structure that overcomes the limitations of static dependence graphs. The data structure, called *dynamic dependence graphs (DDGs)*, allows the dependence structure to change during change propagation. This enables representing arbitrary computations using dynamic dependence graphs. Although dynamic dependence graphs are general purpose, they alone do not suffice for efficient incremental computation. To achieve efficiency, we introduce *memoized dynamic* *dependence graphs (MDDGs)* and a change-propagation algorithm for them. A key property of memoized DDGs is that they enable memoization under side-effects (mutations) to the memory.

We present language techniques for writing programs so that memoized DDGs of graphs can be constructed by a run-time system. Using these techniques the programmer can transform an ordinary program into a self-adjusting program by making small methodical changes to the code.

A key property of our techniques is that their computational complexity can be determined using analytical techniques. We present complexity bounds for change propagation and present an analysis technique, called trace stability, for determining the computational complexity of self-adjusting programs. Informally we show that a program that has "stable traces" adjust to changes efficiently.

We show that our techniques are effective by considering a number of applications. From a theoretical perspective, we show bounds on the complexity of our applications under certain classes of input changes. These bounds closely match (within an expected constant factor), complexity bounds obtained in the algorithms field. None of the previously proposed incremental computation techniques can achieve these theoretical bounds. From a practical perspective, we implement our techniques and present an experimental evaluation. Our experiments show that our techniques have low practical overhead and perform well even when compared to solutions developed in the algorithms community.

Part I

Algorithms and Data Structures

Introduction

This part of the thesis describes the key algorithms and the data structures for supporting self-adjusting computation. Chapter 4 describes *dynamic dependence graphs or DDGs* for representing computations and a *change-propagation* algorithm for propagating changes through this graph. Although DDGs and change propagation are general-purpose techniques, they rarely yield optimal update times under external changes. Chapter 5 introduces *memoized dynamic dependence graphs (MDDGs)* and a *memoized change propagation* algorithm that addresses this problem. Memoized DDGs and the memoized change propagation algorithm constitute the key data structure and the algorithm behind our techniques.

Chapter 6 presents data structures for memoized DDGs, and memoized change propagation on the standard RAM model. Based on these data structures, we present complexity bounds for the overhead of constructing memoized DDGs and for the memoized change-propagation algorithm. The complexity bound for memoized change propagation applies only to programs written in the so-called *normal form*. This causes no loss of generality, because any program can be transformed to the normal form. Part II provides a higher-level analysis techniques called trace stability for determining the complexity of change propagation.

To facilitate a precise description of (memoized) dynamic dependence graphs and (memoized) change propagation, we define a machine model called the closure machine. The *closure machine* extends the standard RAM model with direct functionality for function calls. This enables remembering the state of function calls in the form of *closures* (*i.e.*, closed functions represented by code plus data). Chapter 3 describes the machine model, defines the *normal form*, and describes how to transform an arbitrary program into the normal form.

Chapter 3

A Machine Model for Self-Adjusting Computation

This chapter describes a Turing-complete machine model, called the *closure machine (CM)* model, and defines the notion of *normal form*. Programs expressed in the model are called *self-adjusting*, because they can be made to self-adjust to changes by running a *change-propagation algorithm*. Normal-form programs are written in a specific form to enable efficient dependence tracking and change propagation.

Self-adjusting computation relies on tracking the data and control dependences of programs as they execute. Once a self-adjusting program completes its execution, the user can perform a *revision* by changing the values stored the memory, and update the execution by running a *change-propagation algorithm*. This revise-and-update step can be repeated as desired. The change-propagation algorithm takes a dependence structure representing an execution and adjusts the execution, the dependence structure, and the memory according to the changes performed during the revision.

The key difference between the standard RAM model and the CM model is that the CM model provides direct support for function calls. This is critical, because the change-propagation algorithm relies on the ability to identify and re-execute function calls affected by an external change. Functions delimit the piece of code that needs to be re-executed when a change takes place.

The CM model provides two different mechanisms for allocating memory. One mechanism is identical to the standard memory allocation, where memory is allocated in blocks by a memory allocator. From the point of the view of the program, the memory allocator is *non-deterministic*, because there is no control over where an allocated block may be placed in memory. The other mechanism provides for *labeled* memory allocation. Each memory allocation specifies a unique label to identify the block being allocated. The labels are then used to ensure that blocks with the same label are allocated at the same address in memory. Since labels are unique throughout an execution, no two allocations return the same block. Labels prove important when analyzing the performance of CM programs under change propagation. It is also possible to omit labels. We take this approach from the language perspective (Chapter 13). From a practical perspective omitting labels is preferable because it is difficult to check statically that labels are unique. From an algorithmic perspective, however, labeled memory allocation enables cleaner analysis and is therefore the preferred approach in this part of the thesis.

A program for the closure model must write to each memory location at most once. This restriction, also known as the *write-once* or *single-assignment* restriction, ensures that the programs written in the model are persistent [31]. Persistence enables tracing the history of an execution and revising it upon an input change. This chapter does not address the issues of statically checking that a program writes to each memory locations at most once. The third part of thesis presents language techniques that address this issue.

The write-once restriction can be eliminated by maintaining a version list for each memory location. Driscoll *et. al.* uses this idea to make data structures persistent automatically. It is possible to extend the techniques presented here for multiple writes. Chapter 14 presents a language and its semantics based on the versioning idea. Multiple writes, however, is not the focus of this thesis.

3.1 The Closure Machine

As the RAM model, the closure machine consists of a random-access memory, and a finite set of registers. Memory consists of a finite but expandable set of *locations*. Every register or location can contain a *primitive value* which is either a reference (pointer) or a primitive type, such as an integer, a floating point number, *etc.* The special value *null* represents an empty reference and the special value $\perp$ represents an *undefined* value. When the machine attempts to use the contents of a register or memory location containing $\perp$, it halts.

A *program* for a closure machine consists of a set of function definitions containing a `main` function where the execution starts and ends. Each function is defined as a sequence of instructions and the instructions of a program are consecutively numbered from one. Figure 3.1 shows the kinds of instructions available in the model. The first kind provides for operations on register data. The second kind provides for conditional branches based on a predicate on the registers. The `read` and `write` instructions read from and write to memory locations specified by the contents of two registers, one specifying the beginning of the memory block and the other specifying the particular location inside that block. Memory is allocated in *blocks* by two kinds of `alloc` instructions. A block of size n contains n memory locations which can be randomly accessed for reading or writing.

The machine provides two mechanisms for allocating memory. The `alloc` r instruction performs *non-deterministic* memory allocation by allocating a memory of blocks whose size is specified by the contents of r. This allocation mechanism is called non-deterministic because the program has no control over where the allocated block may reside in memory. The `alloc`$[r_3, \ldots, r_m]$ r performs labeled memory allocation by allocating a memory block whose size is specified by the contents of r and labeling it with the contents of $r_3, \ldots, r_m$. This allocation mechanism is called *labeled allocation* and provides for deterministic allocation.

The model does not provide an instruction for freeing memory. Unused memory can be recovered via garbage collection without affecting asymptotic performance [44].

To support function calls, the closure machine maintains a *call-stack*. The `call` instructions performs a function call by pushing the address of the next instruction onto the stack, setting all registers that are not passed as arguments to undefined ($\perp$), and passing control to the first instruction of the function. The `ccall` instruction performs a *continuation-call* or a *tail-call*. A continuation call does not push the return address to the stack. Otherwise, a continuation call is like an ordinary function call. Informally speaking, a continuation call is a function call that does not return to its call site. The `return` instruction passes the control to the address at the top of the call stack after setting all but returned registers to undefined.

Setting unused registers to undefined ($\perp$) before a function call ensures that the function does not use

Instruction	Description
$r_1 \leftarrow r_2 \oplus r_3$	Combine the contents of the registers r_2 and r_3 by applying the operation $\oplus$ and place the result in register r_1.
`if` $p(r_1, r_2)$ `then jump` i	If the contents of r_1 and r_2 satisfies the predicate p, then jump to instruction i.
$r_1 \leftarrow$ `read` $r_2[, r_3]$	Place the contents of the memory location specified by the contents of r_3 in memory block specified by the contents of r_2 in r_1. If r_3 is not specified, the first location in the block specified by r_2 is taken.
$r_1, r_2 \leftarrow$ `write` r_3	Place the contents of the register r_3 in the memory location specified by the contents of r_2 in the memory block specified by the contents of r_1. If r_2 is not specified, the first location in the block specified by r_1 is written.
$r_1 \leftarrow$ `alloc` r_2	Allocate a memory block and place a reference to the start of the block to r_1. The number of location in the block is specified by the contents of r_2. All locations in the allocated block have the initial value $\perp$.
$r_1 \leftarrow$ `alloc`$[r_3, \ldots, r_m]$ r_2	Allocate a memory block and place a reference to the start of the block to r_1. The number of location in the block is specified by the contents of r_2 and the allocated block is labeled with the contents of $r_3, \ldots, r_m$. All locations in the allocated block have the value $\perp$.
`call` $f, r_1, \ldots, r_n$	Call function f with arguments $r_1, \ldots, r_n$. Push the return address to the call-stack and set all registers except for the arguments $(r_1, \ldots, r_n)$ to $\perp$ before passing the control to f.
`ccall` $f, r_1, \ldots, r_n$	Continuation-call (tail-call) function f with arguments $r_1, \ldots, r_n$. Set all registers except for the arguments $(r_1, \ldots, r_n)$ to $\perp$ and pass the the control to f. Do not push the return address to the call-stack (continuation-calls do not return to their call site).
`return` $r_1, \ldots, r_n$	Return $r_1, \ldots, r_n$ from function call by passing control to the return-address at the top of the call-stack and setting all registers except for $r_1, \ldots, r_n$ to $\perp$.

Figure 3.1: The instruction set.

any values other than those accessible by its own arguments. Similarly, setting unused registers to undefined after function calls (before the return) ensures that the caller does not use values that it is not returned. These properties enable tracking dependences between data and function calls by observing function arguments and values returned from functions.

The closure machine enables remembering the "state" of a function call by remembering the function and its arguments. This state consisting of the function and the arguments is called a *closure*. The ability to create closures of function calls is critical to change propagation because the change-propagation algorithm operates by identifying and re-executing closures that are affected by input changes.

All valid closure-machine program must satisfy the following restrictions.

1. Write-once restriction: Each memory location is written exactly once.

2. Unique labeling restriction: All memory blocks have a unique label—no two or more memory blocks are given the same label.

The write-once restriction can be checked statically by using type systems. Chapter 11 presents language facilities for writing self-adjusting programs that are guaranteed to be write once. The uniqueness of labels is more difficult to enforce without significantly restricting the set of acceptable programs. Requiring that all labels be unique, however, is stronger than required. Chapter 13 presents language facilities that do not rely on labeling. We chose to support labeled memory allocation because it dramatically simplifies complexity analysis of self-adjusting programs under change propagation.

3.2 The Normal Form

This section defines normal-form programs and describes how an arbitrary program can be transformed into the normal-form. The motivation for introducing normal-form programs is to enable efficient dependence tracking and change propagation. By restricting the kinds of values that can be returned from functions, and by restricting where memory reads and jumps take place, the normal-form ensures that all dependences in a computation can be determined by tracking just the memory operations, and the change-propagation algorithm can update a computation in a single pass.We define normal-form programs as follows.

Definition 1 (Normal Form)
A program is in normal-form if

1. *each function returns either no value, or it returns a number of locations that are allocated at the very beginning of the function;*

2. *all reads take place at the beginning of function calls, more precisely no instruction other than a* `read` *or* `alloc` *precedes a* `read` *inside a function; and*

3. *all branches perform local jumps, i.e., the branch instruction and the instruction that the branch may jump are within the same function call.*

The condition that functions return only memory locations forces all data dependences to arise due to reading and writing of memory. This enables tracking of dependences by observing only memory operations. The condition that all reads take place at the beginning of function calls helps establish a total order on the dependences based on the ordering of function calls. This enables a change-propagation algorithm to update a computation in a single pass over the computation. The condition that all branches be local ensures that the dependences between code and data can be tracked by tracking function calls only.

The transformation of an ordinary program for the closure machine into the normal-form involves

1. converting the functions into *destination-passing style*, and

2. splitting functions at memory reads and branches with continuation-calls.

This transformation can be performed in linear time (in the size of the code) by a compiler. The rest of this section describes the transformation via an example.

1 `fun f() =`	1 `fun f() =`
2 `A`	2 `A`
3 `call g(`r_2, r_3`)`	3 `call g(`r_2, r_3`)`
4	4 $r_0 \leftarrow$ `read` d_0
5	5 $r_1 \leftarrow$ `read` d_1
6	6
7 `if` $r_0 < 0$ `then jump 10`	7 `if` $r_0 < 0$ `then jump 10`
8 `add` $r_0, -1$	8 `add` $r_0, -1$
9 `jump 7`	9 `jump 7`
10 `B`	10 `B`
11 `fun g(`r_2, r_3`) =`	11 `fun g(`r_2, r_3`) =`
12	12 $d_0 \leftarrow$ `alloc` $word\_size$
13	13 $d_1 \leftarrow$ `alloc` $word\_size$
14 `A`	14 `A`
15	15 $d_0 \leftarrow$ `write` r_0
16	16 $d_1 \leftarrow$ `write` r_1
17 `return` r_0, r_1	17 `return` d_0, d_1

Figure 3.2: Transforming function to destination-passing style.

The first step of the transformation forces function calls to return their result through memory instead of registers by using a form of *destination passing*. Consider a function that returns some values. The function is rewritten so that it first allocates a number of *destinations* (locations), and then writes its return values to these destinations. All callers of the function are then rewritten to read the returned destinations.

Figure 3.2 shows an example for this transformation. For the example, assume that `A` and `B` denote basic blocks of straightline code with no function calls or branches. The transformation rewrites function `g` so that it creates two destinations d_0 and d_1 and writes its results to these destinations and returns them. The caller `f`, is rewritten so that the destinations are read into the registers that the `f` expects `g` to return. To denote destinations, we use special registers written as $d_0, d_2, \ldots$. Since each function can return no more than n values where n is the number of registers available to the original program, the transformed program requires no more than $2n$ registers.

The second step of the transformation divides functions so that (1) all reads take place at the beginning of function calls, and (2) all branches are local. The transformation is similar to the well-known transformation of arbitrary programs into the *continuation-passing style*, or *c.p.s.* The main difference is that the continuations are not passed as arguments but instead are called explicitly.

Figure 3.3 shows an example for the second step of the transformation. The function `f` is split into three functions `f_0`, `f_1`, `f_2` and `f_3`. Functions `f_1`, `f_2` and `f_3` take all registers and destinations, denoted $r_1, \ldots, r_n$, as arguments (for brevity, we do not differentiate between registers and destinations in argument positions). The function `f_1` is created to ensure that the reads take place at the start of a function call. The functions `f_2` and `f_3` are created to ensure that branches are local. These two functions start at instructions that are targets of branches. The functions are connected via continuation-calls (`ccall` instructions) and non-local jumps are replaced with continuation calls. Figure 3.4 shows the example program and its normal form yielded by the transformation.

```
1  fun f() =
2     A
3     call g(r2,r3)
4
5
6
7     r0 ← read d0
8     r1 ← read d1
9
10
11
12    if r0 < 0 then jump 17
13       add r0,−1
14       jump 12
15
16
17    B

18 fun g(r2,r3) =
19    d0 ← alloc word_size
20    d1 ← alloc word_size
21    A
22    d0 ← write r0
23    d1 ← write r1
24    return d0,d1
```

```
1  fun f_0() =
2     A
3     call g(r2,r3)
4     ccall f_1(r1,...,rn)
5
6  fun f_1(r1,...,rn)
7     r0 ← read d0
8     r1 ← read d1
9     ccall f_2(r1,...,rn)
10
11 fun f_2(r1,...,rn)
12    if r0 < 0 then ccall f_3(r1,...,rn)
13       add r0,−1
14       jump 12
15
16 fun f_3(r1,...,rn)
17    B

18 fun g_0(r2,r3) =
19    d0 ← alloc word_size
20    d1 ← alloc word_size
21    A
22    d0 ← write r0
23    d1 ← write r1
24    return d0,d1
```

Figure 3.3: Splitting function calls at reads and jumps.

3.2.1 The Correspondence between a Program and its Normal Form

The dependence tracking techniques and the change propagation algorithms described in this thesis assume that programs are written in the normal form. For the purposes of complexity analysis of self-adjusting programs under change propagation, we consider not just normal but also ordinary programs. Our analysis techniques, therefore, must relate the normal form of a program to the ordinary version of that program. This section describes the correspondence between a program and its normal form and introduces some terminology to be used in later chapters (in particular in Section 7.3).

Consider some program P its normal form P_n obtained by the normal-form transformation. We say that P is a *native program* and P_n is a *primitive program*. Similarly, we call functions of P and P_n *native* and *primitive* functions respectively. The normal-form transformation splits each native function into one *primary* primitive function and a number of *secondary* primitive functions. The primitive functions derived from a native function `f` are called the *primitives* of `f`. The primary primitive function for a native function `f` is consistently given the name `f_0`.

For example, in Figure 3.4, the function `f` is split into the primitive functions `f_0`, and `f_1`, `f_2`, and `f_3`. The primitives of the native function `f` consists of the functions `f_0`, `f_1`, `f_2`, and `f_3`. The function `f_0` is a primary primitive function, whereas `f_1`, `f_2`, and `f_3` are secondary functions.

The dependence tracking techniques rely on representing the function calls in an execution as a *function-*

1 `fun f() =`	1 `fun f_0() =`
2 `A`	2 `A`
3	3 `call g(`r_2,r_3`)`
4	4 `ccall f_1(`$r_1,\ldots,r_n$`)`
5 `call g(`r_2,r_3`)`	5
6	6 `fun f_1(`$r_1,\ldots,r_n$`)`
7	7 $r_0 \leftarrow$ `read` d_0
8	8 $r_1 \leftarrow$ `read` d_1
9	9 `ccall f_2(`$r_1,\ldots,r_n$`)`
10	10
11	11 `fun f_2(`$r_1,\ldots,r_n$`)`
12 `if` $r_0 < 0$ `then jump 17`	12 `if` $r_0 < 0$ `then ccall f_3(`$r_1,\ldots,r_n$`)`
13 `add` $r_0, -1$	13 `add` $r_0, -1$
14 `jump 12`	14 `jump 12`
15	15
16	16 `fun f_3(`$r_1,\ldots,r_n$`)`
17 `B`	17 `B`
18 `fun g(`r_2,r_3`) =`	18 `fun g_0(`r_2,r_3`) =`
19	19 $d_0 \leftarrow$ `alloc` $word\_size$
20	20 $d_1 \leftarrow$ `alloc` $word\_size$
21 `A`	21 `A`
22	22 $d_0 \leftarrow$ `write` r_0
23	23 $d_1 \leftarrow$ `write` r_1
24 `return` r_0, r_1	24 `return` d_0, d_1

Figure 3.4: An ordinary program and its normal-form.

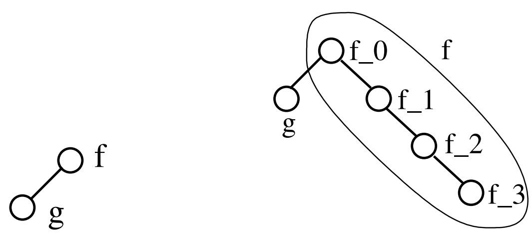


Figure 3.5: Example call tree for a native program and its primitive version.

call tree. The *function-call tree* or the *call tree* of an execution consists of vertices that represent the function calls and edges that represent the caller-callee relationships—an edge (u, v) represents the function call u that performs the call v. The function-call tree of an execution is constructed by tracking the `call` and `ccall` instructions.

When analyzing the performance of change propagation, we argue about the correspondence between function-call trees of native and primitive programs. Consider executing a native program P on some inputs I, and executing the normal-form P_n of P on the same input. Let T be the function call tree for the execution with P and let T_n be the function call tree for the execution with P_n. For example, Figure 3.5 shows the function-call trees of the programs shown in Figure 3.4 for a hypothetical run with the same inputs. Each function call in the native call-tree (the call tree for the native program P), is represented by a sequence of calls in the primitive call-tree (the call tree for the normal program P_n). For example in Figure 3.4, the function call to `f` is represented by the sequence of calls `f_0, f_1, f_2`, and `f_3`. Of the sequence of calls representing `f`, `f_0` is special because it is created by a `call` instruction; the calls to `f_1, f_2, f_3` are created by `ccall` instructions. We refer to the call to `f_0` as the *primary (primitive) call* of `f`. Similarly, we refer to the calls to `f_1, f_2, f_3` as the *secondary (primitive) calls* of `f`. It is a property of the transformation that

1. all primary calls are executed via the `call` instruction,
2. all secondary calls are executed via the `ccall` instruction, and
3. each native function call corresponds to one primary call, and a number of secondary calls.

Chapter 4

Dynamic Dependence Graphs

This chapter introduces the *Dynamic-Dependence-Graph (DDG)* data structure for representing computations and a change-propagation algorithm for adjusting computations to external change. The change-propagation algorithm takes as input the DDG of a computation and a set of external changes to the memory, and adjusts the computation and the DDG to that change. It is a property of the change-propagation algorithm that the DDG and the memory after an execution of the algorithm are identical to those that would be obtained by running the program from scratch on the changed memory.

4.1 Dynamic Dependence Graphs

The dynamic dependence graph of an execution represents the control and data dependences in an execution. The nodes of a dynamic dependence graph consists of *vertices* and *locations*. The vertices represent the function calls and locations represent memory locations allocated during the execution. The edges are partitioned into call edges and dependence edges. *Call edges* represent control dependences between functions, and *dependence (or read) edges* represent data dependences between memory locations and function calls. Formally, a *dynamic dependence graph* $DDG = ((V, L), (E, D))$ consists of nodes partitioned into *vertices* V and *locations* L, and a set of edges partitioned into *call edges* $E \subseteq V \times V$, and *dependences:* $D \subseteq L \times V$. The tree (V, E) is the *(function-)call tree* for the execution.

To enable change propagation, DDGs are augmented with a *tag function*, denoted α, and a memory, denoted σ.

Tags(α): The function α maps each vertex in V to a *tag* consisting of a triple of (1) the function being called, (2) the values of the arguments, and (3) a time interval consisting of a pair of time stamps. The *time-interval* of a vertex corresponds to the execution-time interval that the corresponding function-call is executed. The next section motivates and describes time-intervals.

Memory(σ): The function σ, called the *memory*, maps each location to a value.

Figure 4.1 shows the dynamic dependence graph of an execution of a hypothetical program on some input. The memory is drawn as an array of locations. The vertices are drawn as circles and are labeled with their time interval consisting of two time stamps. Each time stamp is represented by an integer. The curved

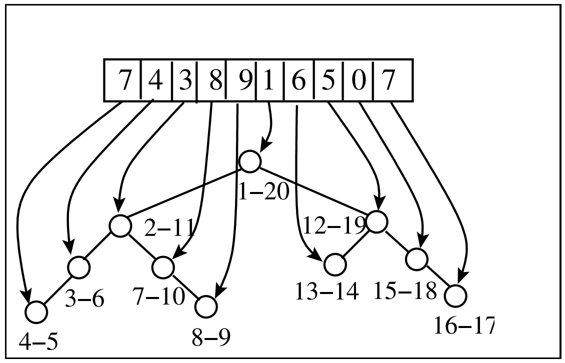


Figure 4.1: An example dynamic dependence graph.

edges between locations and the vertices are the read edges. The straight edges between vertices are the call edges. The tags other than the time stamps are not shown in this example. Throughout this thesis, we omit time stamps drawing DDGs and assume that a preorder traversal of the call tree corresponds to the execution order (and thus the time order) of function calls.

4.2 Virtual Clock and the Order Maintenance Data Structure

To enable change propagation and to construct the DDG of an execution efficiently, the run-time system relies on a *virtual clock* data structure. The data structure maintains the *current time*, dispenses *time-stamps*, and provides a number of operations on time stamps. Figure 4.2 shows the operations supported by the virtual clock. The data structure maintains the current time, `currentTime` explicitly. The `init` operation initializes the virtual clock by creating the initial time stamp and setting the current time to this time stamp. Every time stamp created will be inserted after the initial time stamp. The `advanceTime` operation advances the current time and the `next` operation returns the first time stamp following the current time. The `compare` operation compares two time stamps, and the `delete` operation deletes a given time stamp. All but the `next` operation are directly supported by the order-maintenance data structure of Dietz and Sleator [30] in constant time. It is straightforward to extend this order-maintenance data structure to support `next` operations in constant time.

The run-time system uses the virtual clock to time stamps function calls. For each function call, the system creates two time stamps, one in the beginning, and one at the end of the call. The time stamps of a call are called the *start time stamp*, and the *stop time stamp*. The start and stop time stamps of a call together define the *time interval* for that call. For example, in Figure 4.1, the root call has the start time 1 and stop time 20; the left child of the root has the start time 2 and stop time 11. The system ensures that the ordering of start and stop time stamps accurately represent the time intervals in which functions execute. The system uses time-stamps for determining (1) the sequential execution order of function calls, and (2) the *containment relation* between function calls.

Sequential execution order: The sequential execution order of function calls is defined by the time at which they start executing. For example, if the call $f(a)$ starts executing before the call $g(b)$, we say that $f(a)$ comes before $g(b)$ in sequential execution order. The sequential execution order of function

`currentTime`	:	the current time (stamp).
`init()`	:	initialize the data structure.
`advanceTime()`	:	insert a new time stamp t immediately after the current time (before any other time stamp that comes after the current time); return t.
`delete(t)`	:	delete time stamp `t`.
`compare(t_1, t_2)`	:	compare the time stamps `t_1` and `t_2`.
`next (t)`	:	return the time stamp that comes immediately after the time stamp `t`.

Figure 4.2: The operations on time stamps.

calls can be determined by comparing their start time stamps.

Containment relation: We say that a function call $g(b)$ is *contained* in some other function call $f(a)$ if $g(b)$ is a descendant of $f(a)$ in the call tree. The containment relation between two calls can be determined by comparing their time intervals. In particular if the time-interval of `g(b)` lies within the time interval of another function `f(a)`, then `g(b)` is contained in `f(a)`. For example, in Figure 4.1, the left child of the root is contained in the root. Indeed, the interval of the left child, 2-11, is contained in the interval of the root, 1-20.

4.3 Constructing Dynamic Dependence Graphs.

As a program executes, a run-time system constructs the DDG of the execution by tracking the `read`, `write`, `call`, and `ccall instructions`. To enable the construction and change propagation, the system globally maintains the following:

1. a virtual clock,
2. the DDG being built, denoted $DDG = ((V, L), (E, D))$,
3. the tags α for function calls,
4. the memory σ, and
5. an *active-call queue* Q, containing function calls.

The run-time system initializes V, E, D, Q to the empty set and initializes the virtual clock via the `init` operation. The input of the execution determines the initial values for the set of locations L, and the memory σ. The construction is performed by the `alloc`, `write`, `call`, `ccall`, and `read` instructions; other instructions execute as usual. Executing an `alloc` instruction extends the set of locations, with the allocated location. Figure 4.3 shows the pseudo-code for the functions specifying the executions of `write`, `call`, `ccall`, and `read` instructions. The instructions `call` and `ccall` execute identically.

Memory = σ
$DDG = ((V, L), (E, D))$
Tags = α
Queue = Q

write l, a
    if $\sigma[l] \neq a$ then
        $\sigma[l] \leftarrow a$
        $Q \leftarrow Q \cup \{v \mid (l, v) \in D\}$

$u :$ call(ccall) $f, a_1, \ldots, a_n$
    $v \leftarrow$ newVertex()
    $V \leftarrow V \cup \{v\}$
    $E \leftarrow E \cup \{(u, v)\}$
    $t_s \leftarrow$ advanceTime()
    execute $f(a_1, \ldots a_n)$
    $t_e \leftarrow$ advanceTime()
    $\alpha(v) \leftarrow (f, (a_1, \ldots, a_n), (t_s, t_e))$

$u :$ read l
    $D \leftarrow D \cup \{(l, u)\}$
    return $\sigma[l]$

Figure 4.3: The pseudo-code for `write`, `call`, and `read` instructions.

For the pseudo code, we assume that the parameters to instructions are the contents of the registers. For `read` and `write` instructions, we assume that the location being written and read is provided explicitly. Throughout, we denote locations by the letter l and its variants, denote primitive values by the letter a and its variants. The notation $u : \langle$`instruction`$\rangle$ means that function call corresponding to vertex u is executing the `instruction`.

The write instruction:

The `write` instruction extends the memory σ and updates the active-call queue. Execution of "`write` l, a" first checks if the value a being written to l is the same as $\sigma[l]$, the value currently in memory, by comparing their bit patterns. If the two values have the same bit pattern, they are considered equal, and no action is taken. Otherwise, $\sigma[l]$ is set to a and all functions that read l are inserted into the active-call queue.

Since all locations can be written at most once during an execution, the active-call queue is always empty during an initial execution. During change propagation, however, the active-call queue becomes non-empty due to input changes or writes to memory locations.

The `call` and `ccall` instructions:

The executed `call` and `ccall` instructions build the call tree and advance the time. Figure 4.3 shows the pseudo code for these instructions. The vertex u denotes the function-call executing the instruction. Execution of "`call(ccall)` $f, a_1, \ldots, a_n$" creates a new vertex v and extends V with v

($V = V \cup \{v\}$), inserts a call-tree edge from the caller u to v ($E = E \cup \{(u, v)\}$), advances the time by creating two time stamps before and after the execution of the function call, and extends α by mapping v to the triple consisting of the function being called f, the arguments, and the time-interval (t_s, t_e), *i.e.*, $\alpha(v) = (f, (a_1, \ldots, a_n), t)$.

The `read` function:

The executed `read` instructions build the dependence edges. Execution of "`read` l" inserts a dependence edge from the location l to the vertex u of the function that performs the read, *i.e.*, $D = D \cup \{(l, u)\}$.

4.4 Change Propagation

Self-adjusting computation hinges on a change propagation algorithm for adjusting a program to external changes. The pseudo-code for change propagation is shown in Figure 4.4. The algorithm takes a set of changes χ that maps locations to new values. First, the algorithm changes the memory. For each pair $(l, v) \in \chi$, the algorithm set the value stored in $\sigma[l]$ to the v (line 19). Second, the algorithm inserts the vertices reading the changed locations into the priority queue(line 20). A vertex is called *active* during change propagation if the values of one or more of the locations that it reads has changed since the last execution.

After initializations, the algorithm proceeds to re-execute the active vertices via the `propagateLoop` function. The `propagateLoop` function re-executes the active vertices in order of their start time stamps. Since the start time stamps correspond to the sequential execution order of the vertices, the vertices are re-executed in the sequential execution order. Each iteration of the while loop processes an active vertex v. To process a vertex, the algorithm

1. rolls the virtual clock back to the start time t_s of the function,
2. identifies the vertices, V^-, whose start time stamps are within the interval of the time interval of v, and deletes V^- and all the edges incident to V^- from the DDG by using the `delete` function,
3. determines the caller u of v and re-executes the function-call for v. Re-executing v inserts the call tree and the dependences obtained by the re-execution into the dynamic dependence graph.

The set of vertices V^- deleted in the second step is the descendants of the re-executed vertex v including v itself. As described in Section 4.2, this is because the time-interval of a vertex u is contained in another vertex v if and only if u is a descendant of v in the call tree (or u is contained in v). Since the time-intervals are properly nested—two time intervals are either disjoint, or one falls within the other—checking that the start time stamps are within the interval of the vertex being re-executed suffices to determine containment. The second and third steps replace the dependences created by the previous execution of v with those obtained by the re-execution.

The `delete` function takes a set of vertices and deletes all information pertaining to them. To this end `delete` (1) deletes the vertices and their tags, (2) deletes the call edges adjacent to the vertices, (3) deletes the dependence edges leading into the vertices, (4) deletes the time stamps of the vertices, and (5) deletes the vertices from the queue. The deleted vertices are called *deleted*.

```
Memory = $\sigma$
DDG = $((V, L), (E, D))$
Tags = $\alpha$
Queue = $Q$

delete ($V^-$)
    $E^- \leftarrow \{(u, v) \in E \mid u \in V^- \vee v \in V^-\}$
    $V \leftarrow V \setminus V^-$
    $E \leftarrow E \setminus E^-$
    $D \leftarrow D \setminus \{(l, u) \mid u \in V^- \wedge (l, u) \in D\}$
    $\alpha \leftarrow \{(v, tag) \mid (v, tag) \in \alpha \wedge v \notin V^-\}$
    $\forall u \in V^-$.  delete timeStamps($u$)
    $Q = Q \setminus V^-$

propagateLoop ()
    while ($Q \neq \emptyset$)
        $v \leftarrow$ findMin($Q$)
        $(f, a, [t_s, t_e]) = \alpha(v)$
        currentTime $\leftarrow t_s$
        $V^- \leftarrow \{y \mid t_s \leq$ startTime($y$) $< t_e\}$
        delete ($V^-$)
        $u \leftarrow$ parent($v$)
        $u$ : call $f, a$

propagate ($\chi$)
    $\sigma \leftarrow (\sigma \setminus \{(l, \sigma(l)) \mid (l, \_) \in \chi\}) \cup \chi$
    $Q \leftarrow \{v \mid (l, \_) \in \chi, (l, v) \in D\}$
    propagateLoop ()
```

Figure 4.4: The change-propagation algorithm.

Re-executing active function calls in sequential-execution order, or in the order of their start time stamps, ensures that

1. values read from memory are up-to-date. For example, consider function-calls f and g such that f writes to a memory location that g reads. If f were not re-executed before g, then g may read a location that has not been updated.

2. function calls that do not belong to the computation are not re-executed. When a vertex is re-executed all vertices that are descendants of that vertex are removed from the queue by the `delete` function. This is critical because the function calls corresponding to these vertices may not belong to the revised computation (due to conditionals). Since descendants of a vertex have larger start time stamps than the vertex itself, re-executing vertices in order of their start time stamps ensures that vertices that do not belong to the revised computation are deleted from the queue before they are re-executed.

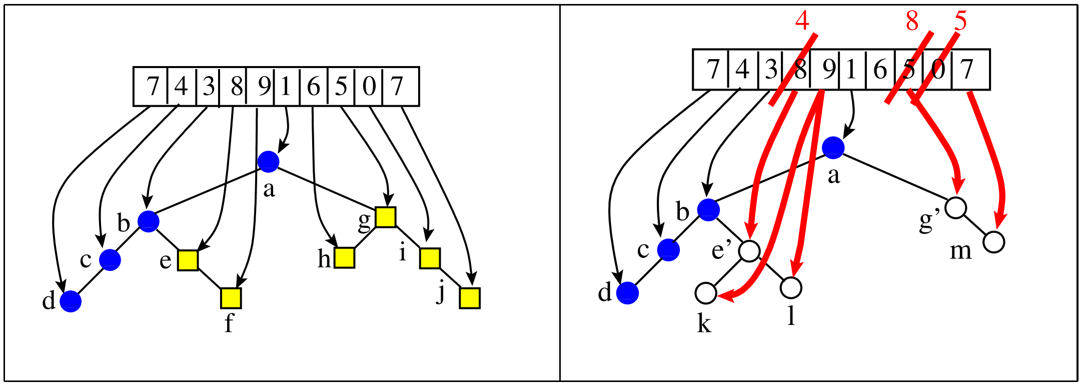


Figure 4.5: Dynamic dependence graphs before and after change propagation.

4.4.1 Example Change Propagation

Figure 4.5 shows an example for change propagation. The dynamic dependence graph for an execution of a hypothetical program with some input is shown on the left. The dynamic dependence graph after changing the input value 8 to 4 is shown on the right. The change propagation takes as input the DDG on the left and the external change and yields the DDG on the right. Since the function call e reads the changed location, the change-propagation algorithm first re-executes e. Assume that this changes the value 5 in memory to 8 and the value 0 to 5. Since these locations are read by g and i, these vertices become active. Since g has the smaller start time (comes earlier than i in a pre-order traversal), the change algorithm re-executes g next. Since i is a descendant of g, re-execution of g deletes i and removes it from the active call queue. Since there are no more active vertices, change propagation terminates.

In this example, some of the vertices are not affected by the change. These vertices are drawn as dark circles. All deleted vertices are drawn as squares, and all fresh vertices, which are created during re-execution are shown as empty circles. All data dependences created during change propagation are drawn as thick lines.

A property of the change-propagation algorithm is that the DDG after change propagation is identical to the DDG that would have been given by a from-scratch re-execution of the program on the changed input. We prove this claim in Chapter 11 by giving a semantics for change propagation and proving its correctness.

Chapter 5

Memoized Dynamic Dependence Graphs

Dynamic dependence graphs (DDGs) enable a change-propagation algorithm to adjust a computation to an external change by re-executing the function calls that are affected by the change. Since change-propagation algorithm re-executes affected function calls from scratch, its running time depends critically on how expensive the re-executed function calls are. For certain classes of applications and external changes, the change-propagation algorithm on DDGs suffices to yield efficient updates. For most applications, however, it performs poorly, because a change often affects an expensive function call. It can be shown for example, that for quick sort and merge sort algorithms, an insertion/deletion of a key can take $\Theta(n \log n)$ time.

To address this problem, we combine memoization and DDGs by introducing *memoized dynamic dependence graphs (MDDGs)* and the *memoized change-propagation algorithm*. Memoized DDGs are annotated DDGs whose vertices can be re-used via memoization. The memoized change-propagation algorithm is an extension of the change-propagation algorithm that performs memo look ups before executing function calls. Self-adjusting computation relies on MDDGs and memoized change propagation for adjusting computations to external changes. The rest of this thesis concerns these techniques unless otherwise stated.

Memoized DDGs and memoized change propagation address the fundamental challenge of performing memoization under side effects. Since memoization relies on remembering and re-using results of function calls, the underlying computation must be *pure*, *i.e.*, the values stored in memory locations cannot be changed via side effects. This is critical because otherwise, there is no effective way of knowing if a result can be re-used.[1] Self-adjusting computation, however, critically relies on side effects. Once a program is executed, any memory location can be changed, and furthermore, memory locations can be written (side-effected) during change propagation.

To provide result re-use under side effects, we rely on DDGs and change propagation. We allow a function call to be re-used only if its DDG is also available for re-use. When a function call is re-used, we perform a change propagation on the DDG of that call. This change propagation adjusts the function call to the side effects that took place since the call was first computed.

Memoized DDGs and memoized change-propagation have the following properties.

- **Side effects:** Memoized DDGs enable re-use of function calls under side effects to memory.

[1] In principle, all memory locations that the function call may read can be checked, but this is prohibitively expensive both in theory and in practice. For example, checking if a function that computes the length of a list can be re-used takes linear time.

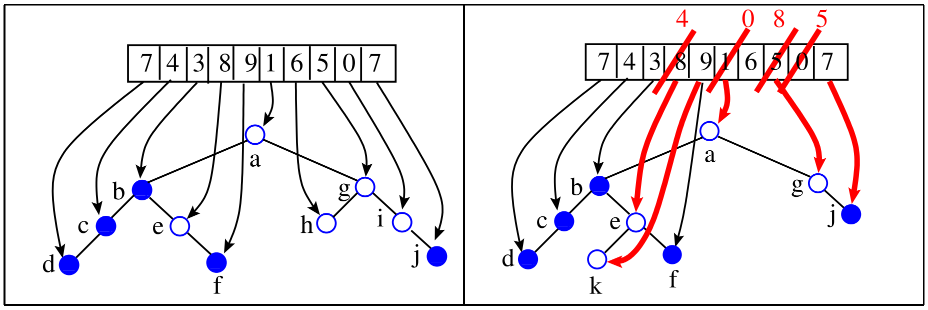


Figure 5.1: Dynamic dependence graphs before and after change propagation.

- **Equality checks:** To check if a function call can be re-used, it suffices to use shallow equality.
- **Efficient Re-use:** Memoized change-propagation supports efficient result re-use.
- **Efficiency:** Memoized DDGs requires $O(1)$ overhead and the same space (asymptotically) as DDGs.

Since memoized DDGs enable adjusting function calls to side effects, shallow equality can be used to determine when function calls can be re-used. *Shallow equality* deems two locations equal if they have the same address, regardless of the values stored in the locations. Note that, due to side effects, a location can contain different values at different times. The key property of shallow equality is that it requires constant time. For example, two lists are equal as long as they reside at the same location, regardless of their contents.

A key property of memoized DDGs is that they enable efficient re-use of computations and they yield to asymptotic analysis techniques. In Chapter 6 and Part II, we present techniques for analyzing the performance of change propagation. In Section 17.3, we apply trace stability to a number of applications and show complexity bounds that closely match best known bounds for these applications.

Memoized DDGs can be implemented efficiently both in terms of time and space. In terms of both time and space, the technique introduces $O(1)$ overhead over dynamic-dependence graphs. Chapter 6 proves these efficiency claims.

5.1 Limitations of Dynamic Dependence Graphs

As an example of how an external change can require an expensive change propagation, consider the DDGs of an hypothetical program shown in Figure 5.1.

Assume for this example that the labels of vertices denote the names of the function being called, and each function takes the location that it reads as the argument. If we are given the DDG on the left, then we can obtain the DDG on the right by changing the appropriate memory locations and running a change propagation algorithm. Since the root labeled with `a` reads one of the changed locations, change-propagation algorithm will re-execute `a` after deleting the tree rooted at `a` (the whole tree). Change propagation will therefore recompute the whole computation from scratch.

This example demonstrates a typical problem that arises when using DDGs. Especially when functions are combined, a simple change to the input can propagate to changes that affect expensive function calls. It

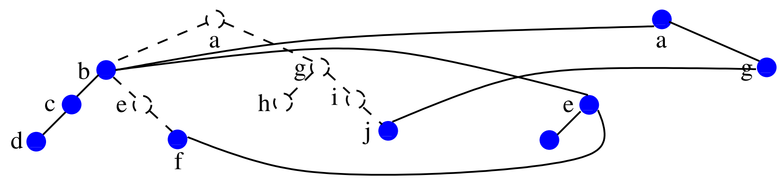


Figure 5.2: Call trees in change propagation.

can be shown for example that the merge sort and the quick sort algorithms both perform no better than a from-scratch executions for nearly all insertions and deletions.

5.2 Memoized Dynamic Dependence Graphs

Consider the example in Figure 5.1, and note that change-propagation can be performed more effectively if the function calls `b,c,d,f,j`, which remain unaffected by the change, can be re-used. These calls are not affected because the memory locations that they read have not been changed. Memoization DDGs enable such re-use.

Change-propagation with (ordinary) DDGs maintains a queue of affected function calls and re-executes these calls. Before re-executing a call, the algorithm deletes the subtree rooted at that call. In the example shown in Figure 5.1, the algorithm deletes the whole tree before re-executing `a`. The deletion of the subtree is critical for correctness because the function `a` can now take an entirely different execution path on the new value read from memory. Often however, re-executing a function call on a slightly different input will perform a computation similar to the previous computation. In the example (Figure 5.1), the subtrees rooted at `a` are quite similar. The idea behind memoized DDGs is to enable vertices of the deleted subtree to be re-used.

The memoized change-propagation algorithm maintains a queue of affected functions and re-executes them. When re-executing a function call, the algorithm does not delete the subtree. Instead, every time a function call is executed, it checks if the call is already in the subtree. If so, the call is re-used and change propagation is performed on the subtree of the call recursively. This change propagation ensures that the re-used call is adjusted to the side-effects appropriately.

As an example consider the memoized dynamic dependence graphs shown in Figure 5.1. Suppose we are given the memoized DDG on the left and suppose that the memory location containing the value `1` is changed to `0`. Since this change only affects `a`, we re-execute `a`. Figure 5.2 shows the call-tree being build during change-propagation. During re-execution, suppose that `a` changes the value `8` to `4` and makes a call to function `b`. Changing `8` to `4` inserts `e` into the queue of calls to be re-executed. Since the call to `b` is already in the subtree being re-executed, it is re-used, and a change-propagation is performed on the subtree rooted at `b` recursively. During this change-propagation, the function `e` is re-executed. Suppose that this function changes the value `5` in memory to `8` and inserts `g` to the call queue, and performs a call to `k` and `f`. Since `f` is in the subtree being re-executed (the subtree rooted at `e`), it is re-used. This completes the recursive change-propagation on the subtree rooted at `b`. Note that the function call `b` is re-used even though it needs to be adjusted to changes. The control now passes to function call `a` that was being re-executed. Now `a` calls `g` which is found in the subtree rooted at `a`, and a change-propagation on `g` takes place. This change propagation re-executes `g`. Re-executing `g` calls `j` and that call is also re-used. Since there are no more function calls in the call queue, change-propagation terminates.

Memoized change propagation creates the complete memoized DDG for a computation by a mixture of "new" and "old" function calls. The new function calls are those that take place during change propagation and the old function calls are those that are re-used from the memoized DDG of the computation. In Figure 5.2, the new calls are drawn on the right, and the old calls are drawn on the left to emphasize the distinction. The calls of the (old) memoized DDG that are not used are deleted. Figure 5.2 shows these calls with dashed lines.

5.3 Constructing Memoized Dynamic Dependence Graphs.

The memoized DDG of an execution is built by a run-time system and is adjusted to external changes by a change propagation algorithm. To enable the construction and change propagation, the system globally maintains the following:

1. a virtual clock,
2. the DDG being built, denoted $DDG = ((V, L), (E, D))$,
3. the tags α for function calls,
4. the memory σ, and
5. an *active-call queue* Q, consisting of function calls.
6. a *memo finger* Φ, which is the last time stamp of the current memo window.

The only difference between this global information and the information maintained in the original construction (Chapter 4) is the memo finger Φ. The memo finger and the current-time stamp together define the *current memo window*, which is defined as the interval $[currentTime, \Phi]$. The current memo window is used to determine if a function call can be re-used. A function-call is re-used in place of another call, if the two calls are equal, and if the interval of the function call is within the current memo window. We say that an interval $[t_1, t_2]$ is *within* another interval $[t_3, t_4]$, written $[t_1, t_2] \sqsubset [t_3, t_4]$ if t_1 is greater than t_3 and t_2 is less than t_4.

The run-time system initializes V, E, D, Q to the empty set and initializes the virtual clock and sets the memo finger to the initial time stamp created. The input to the computation determines the set of locations L and the memory σ. As in the original construction, the dependence graph and the active-call queue is built by `alloc`, `write`, `ccall`, `call`, and `read` instructions. Executing an `alloc` instruction extends the set of locations with the allocated location. Figure 5.3 shows the pseudo-code for `write`, `ccall`, `call`, and `read` instructions. The algorithm for `write`, `read`, and `ccall` instructions remain the same as in the original construction; the `call` instruction is different. When writing pseudo-code, we use the v : $\langle$`instruction`$\rangle$ notation, where the vertex v is the function call executing the `instruction`.

```
Memory = $\sigma$
$DDG = ((V, L), (E, D))$
Tags = $\alpha$
Queue = $Q$
Memo Finger = $\Phi$

write $l, a$
    if $\sigma[l] \neq a$ then
        $\sigma[l] \leftarrow a$
        $Q \leftarrow Q \cup \{v \mid (l, v) \in D\}$

$v$ : read $l$
    $D \leftarrow D \cup \{(l, v)\}$
    return $\sigma[l]$

$u$ : ccall $f, a_1, \ldots, a_n$
    $v \leftarrow$ newVertex()
    $V \leftarrow V \cup \{v\}$
    $E \leftarrow E \cup \{(u, v)\}$
    $t_s \leftarrow$ advanceTime()
    execute $f(a_1, \ldots a_n)$
    $t_e \leftarrow$ advanceTime()
    $\alpha(v) \leftarrow (f, (a_1, \ldots, a_n), (r_1, \ldots, r_n), (t_s, t_e))$

$u$ : call $f, a_1, \ldots, a_n$
    if $(\exists v \in V.\ \mathit{fun}(v) = f \wedge (a_1, \ldots, a_n) = \mathit{args}(v)\ \wedge$
        timeInterval$(v) \sqsubset$ [currentTime, $\Phi$])
    then (* Memo match *)
        $(f, \_, (a'_1, \ldots, a'_n), [t_s, t_e]) = \alpha(v)$
        $E \leftarrow E \cup \{(u, v)\})$
        $V^- \leftarrow \{y \in V \mid$ currentTime $<$ startTime$(y) < t_s\}$
        delete ($V^-$)
        propagateLoop ($t_e$)
        currentTime $\leftarrow t_e$
        $r_1 \leftarrow a'_1 \ldots r_n \leftarrow a'_n$
    else (* No memo match *)
        $v \leftarrow$ newVertex()
        $V \leftarrow V \cup \{v\}$
        $E \leftarrow E \cup \{(u, v)\}$
        $t_s \leftarrow$ advanceTime()
        execute $f(r_1, \ldots r_n)$
        $t_e \leftarrow$ advanceTime()
        $\alpha(v) \leftarrow (f, (a_1, \ldots, a_n), (r_1, \ldots, r_n), [t_s, t_e])$
```

Figure 5.3: The pseudo-code for `write`, `read`, `ccall`, and `call` instructions.

The executed `ccall` and `call` instructions build the call tree and advance the time. Figure 5.3 show the pseudo code for these instruction. The algorithm for `ccall` remains the same as in the original construction but the `call` is extended to perform memo look ups. Execution of `call` $f, r_1, \ldots, r_n$ first checks if there is a matching call to f with the current arguments that lie within the current memo window. The algorithm then proceeds based on the outcome of the memo lookup.

No memo match: If no matching result is found within the current memo window, then the algorithm proceeds as usual. It extends V with a new vertex v, inserts a call-tree edge from the caller u to v ($E = E \cup \{(u, v)\}$), advances the time, and performs the function call. When the call completes, v is tagged with the tuple consisting of the function being called, f, the arguments, $(r_1, \ldots r_n)$, the result currently in the registers, and the time interval for the call.

Memo match: There is matching result if there exists some $v \in V$ such that *fun*$(v) = f$, and $(a_1, \ldots, a_n) =$ *args*(v), and the time interval of v lies within the current memo window. In this case, the algorithm inserts an edge from the caller u to v ($E = E \cup \{(u, v)\}$); this connects the caller to the call tree of the re-used call. The algorithm then determines the set V^- of vertices whose start times are within the interval defined by the current time and t_s (the first time-stamp of v), and removes V^- using the `delete` subroutine. As in the original construction, the `delete` subroutine deletes the subgraph of DDG induced by V^-. The pseudo-code for `delete` is shown in Figure 4.4

Since equality of arguments is checked using shallow equality and since shallow equality defines two locations equal if their addresses are the same (even if their contents may not be), the algorithm can decide to re-use a function call that is not identical to the call being performed. The algorithm therefore performs change propagation until t_e, the last time-stamp of the call. This change propagation ensures correctness by adjusting the function call to mutations that has been performed since the last execution of the call.

The algorithm completes by recovering the contents of the registers to those at the end of the re-used function call.

An important point of the memo look up mechanism is that it takes place with respect to the *current memo window* defined by the interval $[currentTime, \Phi]$. A function-call with interval $[t_s, t_e]$ can be re-used only if its interval lies within the current memo window, *i.e.* $[t_s, t_e] \sqsubset [currentTime, \Phi]$ (line 17 in Figure 5.3). The memo finger is initialized to the base time stamp by the run time-system and can be changed only by the change-propagation algorithm.

A key property of the techniques it that function calls are only re-used during change propagation. This is ensured by setting the memo finger to the initial time stamp. As we will see in the next section, the change-propagation algorithm sets the memo finger to the last time stamp of the call currently being re-executed. This makes all calls that are in the subtree of the call currently being re-executed available for re-use.

When a function call is found in the memo, the `call` subroutine removes all function calls whose start times are between the current time and the beginning time of the function call being re-used (line 22 in Figure 5.3). This effectively *slides* the windows past the function call being re-used. Sliding the window ensures that no function call prior to this call is re-used. Note also that the function call itself can never be re-used.

5.4 Memoized Change Propagation

The pseudo-code for memoized change propagation is shown in Figure 5.4. The algorithm takes a set of changes χ that maps locations to values. First, the algorithm changes the memory. For each pair $(l, v) \in \chi$,

```
Memory = $\sigma$
$DDG = ((V, L), (E, D))$
Tags = $\alpha$
Queue = $Q$
Finger = $\phi$

propagateLoop ($t$)
1       while ($Q \neq \emptyset$)
2           $v \leftarrow$ findMin($Q$)
3           $(f, a, r, (t_s, t_e)) = \alpha(v)$
4           if ($t_s \leq t$) then
5               $\phi \leftarrow t_e$
6               currentTime $\leftarrow t_s$
7               $u \leftarrow$ parent($v$)
8               $u$ : call $f, r$
9               $V^- \leftarrow \{y \in V \mid$ currentTime $\leq$ startTime($y$) $< t_e\}$
10              delete ($V^-$)

propagate ($\chi$)
11      $\sigma \leftarrow (\sigma \setminus \{(l, \sigma(l)) \mid l \in \chi\}) \cup \chi$
12      $Q \leftarrow \{v \mid (l, \_) \in \chi, (l, v) \in D\}$
13      propagateLoop ($t_\infty$)
```

Figure 5.4: The change-propagation algorithm.

the algorithm sets $\sigma[l]$ to v (line 11). Second, the algorithm inserts the vertices reading the changed locations into the priority queue(line 12). A vertex is called *active* during change propagation if it reads a reads a location that has changed since the last execution.

After the initialization, the algorithm proceeds to re-execute all active vertices by calling `propagateLoop`. The `propagateLoop` subroutine takes a time-stamp t and performs change propagation until t by re-executing the active vertices before t. The change propagation algorithm calls `propagateLoop` with the largest time stamp t_∞; the subroutine is called with other time-stamps when a memo match occurs (line 24 in Figure 5.3). The subroutine re-executes the active vertices in order of their start time stamps. Since the start time stamps correspond to the sequential execution order of the vertices, this mimics the sequential execution order. Each iteration of the while loop processes an active vertex v. If the start time of v is greater than t, then the algorithm returns. Otherwise, the algorithm

1. sets the memo finger to the end-time t_e of v,
2. rolls the virtual clock back to the start time t of the function,
3. determines the caller u of v and re-executes the function-call for v,
4. determines the function calls in the current memo window, and deletes them by calling the subroutine `delete`.

Note that re-execution of v adds the call tree for the re-execution and dependences into the dynamic dependence graph.

The `delete` subroutine takes a set of vertices V^- and removes all call edges induced by the vertices in V^-, removes all dependence edges leading into the vertices in V^-, deletes the time stamps of the vertices in V^-, and deletes each vertex in V^- from the queue. The code for `delete` (Figure 4.4) remains the same as with the original change-propagation algorithm.

The key difference between the memoized change-propagation algorithm and the original (non-memoized) algorithm from Section 4.4 is the use of `delete`. Before re-executing some function call v, the non-memoized algorithm deletes all vertices contained in the interval of v from the graph by calling the `delete` subroutine. This effectively deletes all descendants of v from the DDG. The memoized change-propagation algorithm makes the descendants of the re-executed function available for re-use by setting the memo finger to the end time stamp of v. When the re-execution of a vertex completes, the algorithm deletes only those vertices that have not be re-used. Since the `call` subroutine slides the window past the re-used vertices by deleting all vertices that start between the current time and the re-used vertex (line 22 in Figure 5.3), the vertices that have not been re-used during re-execution have start times between the current time and t_e.

A relatively small difference between the memoized and non-memoized change-propagating algorithms is that, in memoized propagation, the `propagateLoop` subroutine now takes a time stamp and performs change propagation only up to that time stamp. This is important to enable the `call` instruction to adjust a re-used function call by performing change propagation on the re-used call (until the re-used call is updated and no further). This ensures that the memoized change-propagation algorithm simulates a complete sequential execution by sweeping the execution time-line from beginning to the end, skipping over unaffected parts, and rebuilding the affected parts of the execution.

5.5 Discussions

Memoized DDGs and the change-propagation algorithm described has two key limitations. First, re-use of function calls take place only during change propagation. Second, when a call is re-used, all calls that are executed between the current time and a re-used call are deleted (and cannot be re-used).

The first limitation ensures that the subgraph of a memoized DDG induced by the call edges is a tree. In other words, there is no sharing of function calls within a computation. It is conceivable that memoized DDGs can be extended to support sharing. With this extension, the call graph would take the form of a directed acyclic graph (DAG) instead of a tree.

The second limitation ensures that the function calls are re-used in the same order as they are originally executed. This is critical to ensure that time stamps of re-used function calls respect their existing ordering as defined by virtual clock. It is possible to eliminate this limitation by a using virtual clock data structure that allows time stamps to be exchanged (swapped). Such a data structure can be implemented in logarithmic time based on balanced trees but it is likely difficult, if not impossible, to achieve this in constant time.

Chapter 6

Data Structures and Analysis

This chapter presents efficient data structures for memoized dynamic dependence graphs and memoized change propagation. The data structures assume the standard RAM model of computation. Based on these data structures, we present complexity bounds on the overhead of constructing memoized DDGs and the time for memoized change propagation.

This chapter concerns only memoized DDGs and memoized change propagation unless otherwise states. The terms "memoized" is often omitted for the purposes of brevity.

6.1 Data Structures

This section describes the data structures for representing virtual clocks, memoized dynamic dependence graphs, and priority queues. We represent memoized dynamic dependence graphs by a combination of dependence graphs and memo tables. The memo tables map the function calls of dynamic dependence graphs to their results.

6.1.1 The Virtual Clock

Section 4.2 describes the virtual-clock data structure required for (memoized) dynamic dependence graphs and (memoized) change propagation. In addition to operations for creating, deleting, and comparing time stamps, the data structure supports a `next` operation for retrieving the time stamp that comes immediately after a given time stamp. These operations can be supported efficiently by using the order maintenance data structure of Dietz and Sleator [30]. Order-maintenance data structures support creating, deleting, and comparing time stamps in constant time. The Dietz-Sleator order-maintenance data structures can be easily extended to support the `next` operation in constant time.

6.1.2 Dynamic Dependence Graphs

To represent dynamic dependence graphs, we maintain a set of locations and set of vertices for the function calls. Data-dependence (read) edges between locations and vertices are represented by maintaining a list of pointers from each location to the vertices that read that location. Since the change-propagation algorithm relies on the time stamps to determine the containment relation between vertices, we do not represent the call edges explicitly.

The `call` instruction and the change propagation algorithm both rely on the ability to determine the set of vertices whose start times are within a particular interval. To support this operation, we maintain a pointer at each time-stamp t that points to the vertex for which t is the start time. For a specified interval, we traverse interval from start to end using the `next` operation of virtual clocks and collect the vertices that are pointed by the time stamps.

6.1.3 Memo Tables

We use hash tables for representing memo tables. Each function is associated with its own memo table. Each memo table entry corresponds to a function call and maps the arguments to the result and the time interval for that call. Since a function can be called with the same arguments more than once, there can be multiple entries for the same call—these entries have different time intervals.

A memo lookup first performs a usual hash-table look up to determine the set of time-intervals for that call, and selects the earliest interval that is within the current memo window. Checking that an interval is contained within the current memo window requires comparing the start and the end time stamp of the interval to those of the current memo window, and therefore takes constant time.

When a vertex is deleted during change propagation, the memo table entry for that call is deleted. This ensures that the number of memo table entries are never more than the number of vertices in the dynamic dependence graph. Memo tables therefore add a constant factor space overhead to dynamic dependence graphs.

If each function is called with the same arguments no more than a constant number of times, then all memo table operations can be implemented with expected constant-time overhead using standard hashing techniques [66, 24]. This assumption holds for the classes of programs that we consider. In this thesis, we assume that memo table operations take constant time.

6.1.4 Priority Queues

The change-propagation algorithm requires a priority queue data structure that supports `insert`, `remove`, and `find-minimum` operations. For the analysis, we assume a priority-queue data structure that supports the `find-minimum` operation in constant, and all other operations in $p(\cdot)$ time as a function of the size of the priority queue.

We assume that it is ensured that the same vertex is not inserted into the priority queue multiple times. This is easily achieved by maintaining a flag for each vertex that indicates whether the vertex is in the priority queue or not.

6.2 Analysis

We show that the (memoized) DDG of a program can be constructed with constant time overhead and present time bounds for change propagation.

We first show a theorem that bounds the overhead of constructing (memoized) DDGs during a from-scratch execution of a program. We refer to such an execution as *initial execution*.

Theorem 2 (Overhead for Constructing (Memoized) DDGs)
A self-adjusting program can be executed on the standard RAM model while constructing its (memoized) DDG with constant-time overhead.

Proof:

We show that each closure-machine instruction can be simulated on the standard RAM model with constant-time overhead.

The register-to-register instructions of closure model can be performed directly on the standard RAM model with no overhead.

The non-labeled memory allocation instruction `alloc` can be performed in constant time using standard allocation techniques. The labeled memory allocation `alloc` can be performed in constant time by memoizing the standard allocation instruction.

The `read` instruction creates a pointer from the location being read to the vertex that reads that location. This requires constant time.

The `write` instruction changes the value at a given location. Since each location is written at most once during an initial execution, and the read of a location can take place only after that location is written, a location has no reads when it is written. Executing a `write` instruction therefore takes constant time.

The `ccall` instruction creates a vertex and two time stamps and executes the function call. Since creating a vertex and a time stamp takes constant time, the overhead of the `ccall` instruction is constant.

Since the memo-window is empty during an initial execution, no memo-matches take place. The overhead for an `call` instruction is therefore constant. Note that in an initial execution, the `call` instructions with memoized and non-memoized DDGs execute identically.

We conclude that the (memoized) DDG of a program can be constructed with constant time overhead.

■

We now present a bound on the time for memoized change propagation. For brevity, we refer to memoized change propagation simply as change propagation. For the analysis, we distinguish between a few types of vertices.

Definition 3 (Active, Deleted, and Fresh Vertices)
Consider executing change propagation on some program and let (V, E) and (V', E') denote the function-call trees before and after the change propagation algorithm. We call certain vertices of V and V' active, deleted, or fresh as follows.

Active: *A vertex $v \in V$ is called active if it is inserted into the priority queue during change propagation (line 4 in Figure 5.3, or line 12 in Figure 5.4).*

Deleted: *A vertex $v \in V$ is said to be deleted if it is deleted from the call tree by the* `delete` *subroutine (Figure 4.4).*

Fresh: *A vertex $v \in V'$ is called fresh if it is created by a function call during change propagation (lines 28 and 9 in Figure 5.3).*

Lemma 4 (Active Vertices)
All active vertices are deleted.

Proof: By definition, an active vertex is inserted into the priority queue. Since change propagation continues until the priority queue becomes empty and since only the `delete` subroutine removes vertices from the queue, all active vertices are deleted. ∎

For the analysis, we assume a priority queue data structure that supports `find-minimum` operations in constant, and all other operations in $p(\cdot)$ time in the size of the priority queue. We analyze the time for priority-queue operations separately from the time spent for change propagation. We charge the cost of the `find-minimum` operations to the change-propagation algorithm. To account for all other operation, we define the *priority queue overhead* as the total time spent by the `remove`, `insert`, and `find-minimum` operations.

Lemma 5 (Priority-Queue Overhead)
Let N_A be the number of active vertices during change propagation. The priority-queue overhead is $O(N_A \cdot p(m))$, where m is maximum size of the base priority queue during change propagation.

Proof: Consider the first time an active vertex v is inserted into the priority queue. Since the change-propagation algorithm ensures that the priority queue consists of distinct vertices, v, will never be inserted into the priority queue until it is removed. When removed from the priority queue, v is either discarded or it is re-executed. If discarded v will never be inserted into the priority queue again. Suppose now v is re-executed. After v starts re-executing, no memory location that is read by v can be written to, because otherwise v would be reading a location that has not been written, or a location must be written more than once. Therefore, v is inserted into the priority queue once, and removed once. We therefore conclude that the priority queue overhead is bounded by $O(N_A \cdot p(m))$, where m is maximum size of the priority queue during change propagation. ∎

The analysis bounds the time for change propagation in terms of the number of active vertices and the total execution time of all fresh and deleted vertices. In our bounds, we disregard the cost of initializing the active-call queue. Since initializations are performed only at the beginning of change-propagation, it is straightforward to generalize the theorems to take the initialization cost into account.

The analysis assumes that the number of read edges going into a vertex is proportional to the execution time of that vertex. This is a conservative assumption, because the number of read edges can be less than the execution time. It is therefore possible to tighten the analysis by considering the read edges separately. For all applications we consider, however, this bound is tight.

We define the weight of a vertex as follows.

Definition 6 (Weight of a Vertex)
The weight of a vertex v, denoted $\mathrm{w}(v)$ is the execution time for the corresponding function call excluding the time spent by the callees of the function.

Theorem 7 (Change Propagation)
Consider executing change propagation on some program and let (V, E) and (V', E') denote the function-calls trees before and after the change propagation algorithm. Let

1. *N_A be the number of active vertices,*

2. *$V_D \subseteq V$ be the set of deleted vertices and W_D be their total weight $W_D = \sum_{v \in V_D} \mathrm{w}(v)$,*

3. *$V_F \subseteq V'$ be the set of fresh vertices and W_F be their total weight, $W_F = \sum_{v \in V_F} \mathrm{w}(v)$,*

4. *$p(\cdot)$ be the time for a priority queue operation,*

5. *m be the maximum size of the priority queue during change propagation.*

The time for change propagation is

1. *$O\left(W_D + W_F + N_A \cdot p(m)\right)$, and*

2. *amortized $O\left(W_F + N_A \cdot p(m)\right)$.*

Furthermore, $m \leq N_A \leq W_D$.

Proof: Consider the change-propagation algorithm, we separate the total time for change propagation into the following disjoint items:

I. the cost of finding the vertices to be deleted,

II. the cost of `delete`,

III. the cost for the iteration of the main loop of `propagateLoop`,

IV. the cost of calls to `ccall` subroutine,

V. the cost of calls to `call` subroutine,

VI. the cost of finding the active vertices, and

VII. the priority queue overhead.

As described in Section 6.1.2, the vertices to be deleted are found by traversing the time-interval that contain them. This takes time proportional to the number of deleted vertices. Item I is therefore $O(W_D)$. Since a vertex can be deleted in constant time, and the number of dependences leading into a vertex is bounded by its execution time, Item II, the cost of `delete` operations, is bounded by the total the weight of the deleted vertices $O(W_D)$. This cost excludes the priority-queue operations, which will be cumulatively accounted for. Item III, the cost of the iteration of the change-propagation loop, is bounded by the number of active vertices. Since each active vertex is deleted by Lemma 4, the cost for the main loop is $O(W_D)$. The `ccall` subroutine creates a vertex and the time stamps for the function being executed and executes the function call. Since vertices, and time stamps are created in constant time and each `ccall` creates a fresh vertex whose weight is equal to the execution time for the function call being executed, the total time for `ccall` instructions is bounded by $O(W_F)$.

To analyze the cost for `call` instructions, we divide the cost into three disjoint items

1. the cost for executing the function being called,

2. the cost for memo lookups, and
3. the cost for finding and deleting the vertices prior to the re-used vertex.

Item 1 is the cost of performing an ordinary non-memoized call and is incurred only when a memo miss occurs. Since each call that misses the memo creates a fresh vertex, Item 1 is bounded by $O(W_F)$. To bound Item 2, we will consider the memo looks that miss and hit separately. The cost for the memo look ups that miss is bounded by $O(W_F)$ because each miss performs a fresh call, and memo look ups take constant time. Consider now a function call that is found in the memo. If the caller of this function is a fresh call, then the cost of performing the memo lookup, which is constant, is charged to the caller. The cost for this case is therefore bounded by expected $O(W_F)$. If the caller is the change propagation algorithm itself, then this is an active call. Since each active call is deleted, the cost for performing the memo lookup is charged to the cost of deleting that vertex. The cost for this case is therefore $O(W_D)$. Item 3 is bounded by $O(W_D)$ as discusses previously (Items I and II).

To bound Item VI, the time for finding the active vertices (line 4, note that the cost of finding the active vertices is proportional to the total number of dependences leading into them. Since this is no more than the execution time of the active vertices, and all active vertices are deleted, Item VI is bounded by $O(W_D)$.

We bound Item VII, the time for the priority queue operations (line 2 in Figure 5.4 and line 8 in Figure 4.4), over the complete execution of change propagation. By Lemma 5, the overhead of the base priority queue operations is $O(N_A \cdot p(m))$. Since the time for `find-minimum` operations is constant, the cost of these operations can be charged to the cost of executing the main loop. Since the priority queue contains distinct vertices, size of the priority queue m is bounded by N_A, $m \leq N_A$. By Lemma 4, each active vertex is deleted and therefore $N_A \leq W_D$.

The total time for change propagation is therefore bounded by $O(W_D + W_F + N_A \cdot p(m))$. Since a vertex can be deleted exactly once through a sequence of change propagations, the cost of deleting a vertex can be charged to performing the execution that created the vertex. This yields the amortized bound $O(W_F + N_A \cdot p(m))$.

■

Part II

Trace Stability

Introduction

This chapter presents an analysis technique, called *trace stability*, for bounding the time for memoized change propagation under a class of input changes. Since the memoized change-propagation algorithm is more powerful than the non-memoized change-propagation algorithm, the rest of this thesis considers only the memoized change-propagation algorithm. For brevity, the term "change propagation" refers to "memoized change propagation" unless otherwise stated.

Trace stability applies to a large class of computations, called *monotone computations*, and does not require that programs be written in the normal-form. Most programs are either naturally monotone, or are easily made monotone by small changes to the program. For example, all applications that we consider in this thesis are monotone. The techniques can be generalized to arbitrary, non-monotone programs by incurring an additional logarithmic factor slowdown.

Trace stability relies on representing a computation with a trace consisting of function calls performed in that computation. The *distance* between two computations is measured as the symmetric set difference between the sets of function calls. The chapter shows that for two computations, the change-propagation algorithm will transform one computation to the other in time proportional to the distance between the traces of the computations. More precisely, if the distance between the traces T and T' of the computations is d, then the change-propagation algorithm requires $O(d \cdot p(m))$ time for transforming one computation to the other, where $p(m)$ is the priority-queue overhead during change propagation with $m \leq d$. Using a standard priority queue, this yields a $O(d \log d)$ bound on change propagation. Chapter 7 makes precise the notion of traces and proves this bound. Chapter 8 generalizes this bound for a class of input changes.

To present tight asymptotic bounds for change propagation, it is often necessary to bound the priority-queue overhead carefully. Chapter 8 presents two techniques for this. We introduce the notion *dependence width* between two computations, and show that the dependence width gives a bound on the size of the priority queue when transforming one computation to the other. By analyzing the dependence width, it is often possible to show that the size of the priority queue, and therefore the priority-queue overhead, is constant. As a second technique, we define a class of computations, called *read-write* regular, for which a constant-time FIFO queue extended with priorities, called FIFOP queue, can be used during change propagation. Since the data structure requires constant time per operation, the priority queue overhead for read-write regular computations is bounded by a constant.

Chapter 7

Traces and Trace Distance

This chapter defines the notions of traces and trace distance, and presents time bounds for memoized change propagation in terms of the distance between traces. We define the trace of a computation as the function call tree of the computation annotated with certain information. For two traces that satisfy certain requirements, we define the distance based on the symmetric set difference of the set of function calls of the traces.

7.1 Traces, Cognates, and Trace Distance

We define the trace of an execution (or a computation) as the ordered function-call tree of the execution, where each function-call is tagged with certain information. Each vertex of a trace represents a function call and each edge represents the caller-callee relationship between the parent and the child.

Definition 8 (Trace)
A trace is an ordered tree representing the function-call tree of an execution. The children of a vertex are ordered in their execution order. Every vertex is annotated with a sequence of values that together constitute a tag. *The tag of a vertex v consists of*

1. *the function being called, denoted* $\text{fun}(v)$,
2. *the arguments* $(a_1, \ldots, a_k)$ *to the function call, denoted* $\text{args}(v)$,
3. *the tuple* $(a_1, \ldots, a_m)$ *of values read in the body of the function (the value extracted not the location being read), denoted* $\text{reads}(v)$,
4. *the tuple* $(a_1, \ldots, a_n)$ *of values returned to the function from its callees, denoted* $\text{returns}(v)$, *and*
5. *the weight of the function, which is equal to its execution time (not including the running time of its callees), denoted* $\text{w}(v)$.

Figure 7.1 shows two traces of two hypothetical executions. Each vertex is labeled with the function call and the arguments, the reads, and the returns. For example, the root represents the call to function `a` with argument `1`, reads `2`, and returns `3` and `4`.

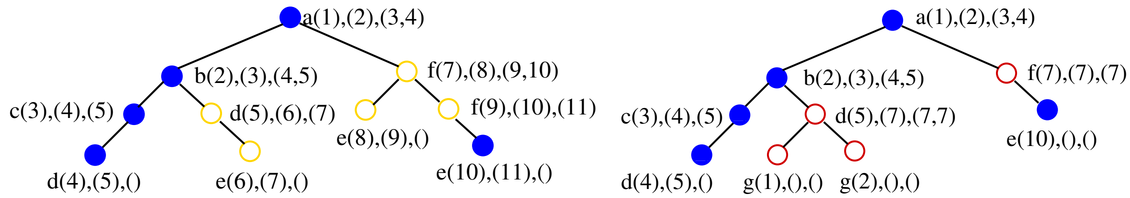


Figure 7.1: Example traces.

Our analysis techniques rely on a notion of similarity and equality between function calls. We say that two function calls are similar if they call the same function and the arguments are equal. Given two arguments $(a_1, \ldots, a_m)$ and $(b_1, \ldots, b_n)$, we say that the arguments are equal if $m = n$ and $\forall i. 1 \leq i \leq m. a_i = b_i$. We determine equality of values, *i.e.*, a_i's and b_i's, using *shallow equality*. Two primitive values, such as integers, booleans, *etc.*, are equal if they are the same value, *e.g.*, $1 = 1$, $true = true$. Two locations are equal if they have the same labels. Note that, since locations are considered equal based on their labels or addresses, two functions can be similar even if the values stored at their arguments are different. Since the closure machine allocates locations that have the same labels at the same memory address, at the machine level, shallow equality is the same as bit-pattern equality.

Definition 9 (Call Similarity)
The calls v and v' are similar, denoted $v \sim v'$, if

1. $\text{fun}(v) = \text{fun}(v')$, *and*

2. $\text{args}(v) = \text{args}(v')$.

We say that two function-calls equal if they are similar, they read the same values from the memory, and they receive the same values from the functions that they call. As with arguments, we define equality of returns and reads based on shallow equality.

Definition 10 (Call Equality)
Two function calls v and v' are equal, denoted $v \equiv v'$, if

1. $v \sim v'$,

2. $\text{reads}(v) = \text{reads}(v')$, *and*

3. $\text{returns}(v) = \text{returns}(v')$.

Since the arguments, the values read, and the values returned completely specify the values that the execution of the function depends on, two equal function calls execute identically. Note that this is true only for the part of the execution that takes place inside the body of the function—not for its subcalls.

Given two traces T and T', we will pair up equal vertices from T and T' and define the distance between the traces as the total weight of the vertices that do not have a pair. The notion of *cognates* formalizes this idea.

Definition 11 (Cognates)
Consider the traces T and T' of a program on two inputs and let V and V' be the set of vertices of T and T' respectively. A cognates, C, of T and T' is a relation such that

1. $C \subset V \times V'$,
2. *each pair of cognates consists of equal vertices: if $(v, v') \in C$ then $v \equiv v'$, and*
3. *no vertex of V or V' is paired up with more than one vertex: if $(v, v'_1) \in C$ and $(v, v'_2) \in C$, then $v'_1 = v'_2$. Similarly if $(v_1, v') \in C$ and $(v_2, v') \in C$, then $v_1 = v_2$.*

For example, in Figure 7.1, the set of vertices with labels
{`(a(1),a(1)),(b(2),b(2)),(c(3),c(3)),(d(4),d(4)),(e(10),e(10))`} is a well-defined set of cognates for the two traces.

To see how cognates formalize the notion of sharing between traces, consider two traces T and T' and a cognate relation C between the vertices V and V' of T and T' respectively. Color the vertices in V and V' with blue, yellow, or red as follows.

Blue Vertices: The set of blue vertices B consists of vertices of T that have a cognate, $B = \{v \mid v \in V \wedge \exists v'. (v, v') \in C\}$.

Yellow Vertices: The set of yellow vertices Y consists of vertices of T that have no cognates, $Y = V \setminus B$.

Red Vertices: The set of red vertices R consists of vertices of T' that have no cognates, $R = V' \setminus \{v' \mid v' \in V' \wedge \exists v. (v, v') \in C\}$.

If we are given the trace T, then we can construct T' by deleting the yellow vertices from T, re-using the blue vertices (of T), and creating the red vertices by executing the corresponding function calls. For example, in Figure 7.1, the trace on the right can be obtained from the trace of the left by re-using the vertices labeled with {`a(1),b(2),c(3),d(4),e(10)`}, deleting the vertices labeled with
{`d(5),e(6),f(7),e(8),f(9)`}, and executing the vertices labeled with {`d(5),g(1),g(2),f(7)`}.

Based on this re-use technique, we can obtain T' from T by performing work proportional to the execution time of red vertices plus the time for deleting the yellow vertices. Deleting a yellow vertex requires time proportional to the number of edges incident to it. Since the number of edges incident on a vertex is bounded by the weight (execution time) of the vertex, the time for deletion is upper bounded by the execution time.[1] The following distance metric defines the distance between two traces based on this idea.

Definition 12 (Trace Distance)
Let T and T' be the traces of a program on two inputs and let V and V' denote the set of vertices of T and T' respectively. Let $C \subseteq V \times V'$ be a cognate relation for V and V'. Let Y (yellow) and R (red) be the subsets of V and V' consisting of vertices that do not have cognates, i.e., $Y = V \setminus \{v \mid v \in V \wedge \exists v'. (v, v') \in C\}$, and $R = V' \setminus \{v' \mid v' \in V' \wedge \exists v. (v, v') \in C\}$. The distance from T to T' with respect to C, denoted $\delta_C(T, T')$ is defined as

$$\delta_C(T, T') = \sum_{v \in Y} \mathrm{w}(v) + \sum_{v' \in R} \mathrm{w}(v').$$

[1] This upper bound is tight for all the applications that we consider in this paper, but may not be in general.

For example, in Figure 7.1, the distance between the two traces is 9 if we assume that the each function call has weight 1.

Since all yellow vertices are deleted exactly once, and the time for their deletion can be bounded by their execution time, the cost of deleting yellow vertices can be amortized to the cost of creating them. Based on this amortization, another notion of trace distance can be defined as the total weight of red vertices. We prefer the above-defined measure, because it is symmetric and it helps present real-time, instead of amortized, bounds.

7.2 Intrinsic (Minimum) Trace Distance

We define the intrinsic distance between two traces as the minimum distance over all possible cognates. The intrinsic distance gives a lower bound for the technique of updating traces by re-using blue, deleting yellow, and re-executing red vertices under the assumption that deleting a vertex takes time proportional to its execution time. In the next section, we show that the distance between two traces is equal to the intrinsic distance between them, if the traces are *monotone*. For monotone traces, we show that the time for change propagation can be bounded as a function of the intrinsic distance between them.

Definition 13 (Intrinsic Distance and Maximal Cognates)
Let T and T' be the traces of a deterministic program on two inputs and let V and V' denote the set of vertices of T and T' respectively. The intrinsic distance from T to T', denoted $\delta^{\mathrm{min}}(T,T')$, is equal to the minimum distance from T to T' over all possible cognates between T and T'. More precisely, $\delta^{\mathrm{min}}(T,T') = \min_{C \subseteq V \times V} \delta_C(T,T')$. We say that a cognate relation C^{max} is maximal, if $\delta_{C^{max}}(T,T') = \delta^{\mathrm{min}}(T,T')$.

Given two traces T and T', a maximal cognate relation can be constructed by greedily pairing equal vertices from T and T'. We start with an empty cognate set $C = \emptyset$, and add pairs to C by considering the vertices of T in an arbitrary order. For each vertex v of T, we check there is any vertex v' of T' that is equal to v, $v \equiv v'$, and that has not yet been paired with another vertex, *i.e.* $\forall\, u.\, (u,v) \notin C$. If such a vertex v' exists, we extend C with the pairing (v,v'), $C \leftarrow C \cup (v,v')$. If there are multiple choices for v, we pick one arbitrarily.

This greedy-pairing algorithm yields a maximal relation. To see this, note first that the algorithm pairs equal vertices, and it pairs a vertex with at most one other vertex. The algorithm therefore constructs a well-defined cognate relation. Since any vertex can have at most one cognate, the relation is maximal in size (the number of pairs). Since all equal vertices have the same weight, the algorithms yields a maximal cognate relation.

7.3 Monotone Traces and Change Propagation

This section presents a bound on the time for change propagation in terms of the intrinsic distance between the traces of a computation. The bound holds only for certain computations that we call monotone. We start by defining *concise* programs and monotone traces. We then prove the main theorem of this section.

Definition 14 (Concise Programs)
Let P be a program and let $\mathcal{I}$ be the set of all inputs that are valid for P. We say that P is concise if it satisfies the following properties:

1. *all memory allocation instructions are uniquely labeled, and*

2. *any function call that is performed in an execution of P with any input $I \in \mathcal{I}$ is unique, i.e., for any two function calls $f(a)$ and $f(b)$, of the same function f, the arguments a and b are different.*

We say that the traces of a concise program are monotone, if they satisfy certain properties. Let P be a concise program with input domain $\mathcal{I}$ and let T and T' be traces of P with two inputs from $\mathcal{I}$. We say that T and T' are (relatively) monotone, if the execution order, and the caller callee relationships between similar functions are the same in both T and T'. Formally, we define monotone traces as follows.

Definition 15 (Monotone Traces)
Let P be a concise program with input domain $\mathcal{I}$ and let T and T' be the traces of P with two inputs from $\mathcal{I}$. We say that T and T' are monotone if for any u and v, for which there exists two vertices u' and v' from T', such that $u \sim u'$ and $v \sim v'$, the following are satisfied

1. *if u comes before v in a preorder traversal of T, then u' comes before v' in a preorder traversal of T', and*

2. *if u is an ancestor of v, then u' is an ancestor of v'.*

Throughout this thesis, we only compare traces belonging to the same program. For brevity, we often omit the reference to the program.

Definition 16 (Monotone Cognates and Monotone Distance)
Consider two monotone traces T and T' and let V and V' be the set of vertices of T and T' respectively. The monotone cognates of T and T', denoted $\mathcal{C}_{Monotone}(T,T')$, is $\mathcal{C}_{Monotone}(T,T') = \{(v,v') \mid v \in V \wedge v' \in V \wedge v \equiv v'\}$. The monotone distance between T and T' is the distance with respect to monotone cognates, i.e., $\delta_{\mathcal{C}_{Monotone}(T,T')}(T,T')$.

We first show that monotone cognates satisfy Definition 11 and therefore are well defined. It is easy to see that Definition 16 satisfies the first and the second conditions of Definition 11. The third condition that no vertex is paired with more than one vertex follows from the fact that the monotone traces consists of unique function calls.

We show that monotone cognates are maximal and therefore the distance measure defined based on monotone cognates is equal to the intrinsic (minimum) distance.

Theorem 17 (Monotone Cognates are Maximal)
Consider two monotone traces T and T'. The set C of monotone-cognates of T and T', $C = \mathcal{C}_{Monotone}(T,T')$ is maximal, and the monotone distance between T and T' is equal to the intrinsic distance between them, i.e., $\delta_C(T,T') = \delta^{\min}(T,T')$.

Proof: Since T and T' both consist of unique function calls, each vertex v of T can have at most one cognate in any possible cognate relation between T and T'. The maximal cognate C^{max} relation between

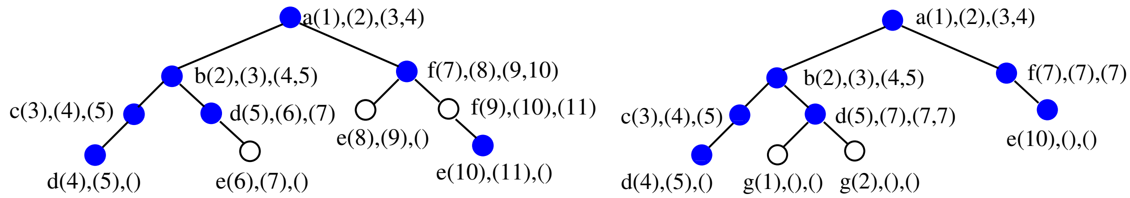


Figure 7.2: Example monotone traces.

T and T' will therefore consists of all equal vertices from T and T'. Therefore $C^{max} = \mathcal{C}_{Monotone}(T, T')$ and the intrinsic distance is equal to the distance with respect to C^{max}. ∎

Figure 7.2 shows a pair of traces that are monotone (this is the same example as shown in Figure 7.1). To determine if two traces are monotone or not, it suffices to consider just the function names and arguments—the reads and returns can be ignored. The function calls that are similar in both traces are drawn as full circles, other function calls are drawn as empty circles. To see that the traces are monotone, it suffices to consider the dark circles and show that they have the same ancestor-descendant, and execution order relationships. An easy way to check for monotonicity is to check that the traces are equal when the empty circles are contracted away.

To compute the distance between the traces shown in Figure 7.2, we ignore the structure of the traces and compare the sets of function calls. Assuming that the weights of each function call is 1, the distance between these traces is 9—the function call contributing to the distance are the set of calls $\{$`d(5),e(6),f(7),e(8),f(9)`$\}$ from the first trace, and $\{$`d(5),g(1),g(2),f(7)`$\}$ from the second trace, because these vertices do not have a cognate.

7.3.1 Change Propagation for Monotone Traces

This section presents a bound on the time for change propagation in terms of the distance between monotone traces. For the analysis, we consider programs written for the closure model, called *native programs*, and transform native program programs into *primitive programs* by applying the normal-form transformation described in Section 3.2. This is necessary because dynamic dependence graphs and change propagation is defined only for normal-form programs.

We use different notation for denoting trace vertices, which represent native function calls, and MDDG vertices, which represent primitive function calls. We say that a vertex or the corresponding function call is *primitive* if the vertex is created during the execution of a primitive program. Similarly, we say that a vertex or the corresponding function call is *native* if the vertex is created during the execution of a native program. We use lower case letters $u, v, \ldots, x$'s for primitive function calls and use lower case letters with a hat, $\hat{u}, \hat{v}, \ldots, \hat{x}$'s for native function calls. When not ambiguous, we use the same letter for a primitive call and the native call (distinguished by the presence of the hat) that the primitive call belongs to.

The analysis often requires checking the similarity and equality of primitive vertices. We define similarity and equality of primitive vertices the same way as native (trace) vertices.

As described in Section 3.2.1, the normal-form transformation splits a function into one *primary* and a number of *secondary* functions. The primary function is always called via the `call` instruction and the

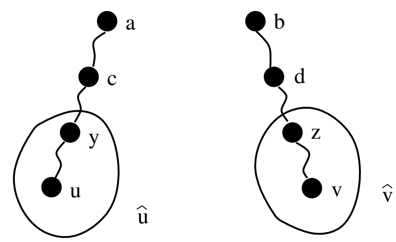


Figure 7.3: The primitives of $\hat{u}$ and $\hat{v}$ and their ancestors.

secondary functions are always called via the `ccall` instruction. It is a property of the transformation that each native function call corresponds to one primary function call (vertex), and a number of secondary function calls. We say that the primitive function calls *belong* to the native function call that they correspond to. Section 3.2.1 describes the correspondence between native and primitive function calls and the properties of primitive programs in more detail.

The main idea behind the proof is to show that each vertex deleted during change propagation and each fresh vertex created during change propagation corresponds to a native function call in the trace that does not have a cognate. We start by proving a key lemma that shows that vertices related by re-execution are similar.

Lemma 18 (Re-Execution Yields Similar Vertices)
Let P be a concise program and G be the dynamic dependence graph of P with some input I. Consider changing the input to I' and performing change propagation on G. Let G' be the dynamic dependence graph obtained after change propagation. Let T and T' be the traces of P with inputs I and I'. Let u be a primitive vertex that is re-executed during change propagation and let v be the vertex created by the re-execution of u. Let $\hat{u}$ and $\hat{v}$ denote the native vertices corresponding to u and v respectively. The vertices $\hat{u}$ and $\hat{v}$ are similar.

Proof: For the proof, we will consider the primitive vertices of G and G' in relation to the native vertices of T and T'. Figure 7.3 illustrates the call trees of G and G', $\hat{u}, \hat{v}$, and other vertices considered in the proof.

Let y and z be the primary vertices for $\hat{u}$ and $\hat{v}$ respectively. By the normal-form transformation, we know that y and $\hat{u}$ are tagged with the same function call, and that z and $\hat{v}$ are tagged with the same function call. That is, we know that $y \sim \hat{u}$ and $z \sim \hat{v}$. To show that $\hat{u} \sim \hat{v}$, we will show that $y \sim z$.

Since primary vertices are never re-executed, y and u are distinct vertices, and so are v and z, *i.e.*, $u \neq y$ and $v \neq z$. Suppose now that y or some other ancestor of u that is a descendant of y is re-executed. Since all secondary vertices are performed via `ccall` instructions, and these instructions do not perform memo lookups, the vertex u will be deleted. But then u cannot be re-executed. We therefore conclude that no proper ancestor of u that is a descendant of y is re-executed.

Consider all the ancestors of u and v in G and G' respectively. If none of the ancestors of u is re-executed, then we know that all proper ancestors of u and v read the same values. Since the primary vertices y and z are ancestors of u and v, we know that $y \sim z$.

Suppose now that an ancestor of u is re-executed, and let a be the least re-executed ancestor of u and let b be the vertex created by the re-execution of a. Since u is re-executed, some ancestor c of u is found in the memo during the execution of b. Let d be the descendant of b that finds c in the memo. Since memo

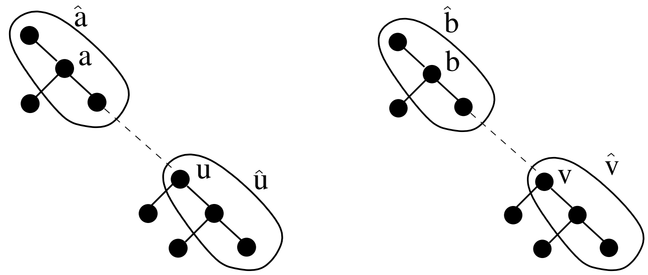


Figure 7.4: The call trees of a computation before and after a change.

re-use take place between similar function calls, we know that $c \sim d$. Since only primary function calls can perform memo lookups, we know that c is an ancestor of y. Since a is the least re-executed vertex, we know that no other vertex on the path between c and u is re-executed. Since $c \sim d$, we know that the vertices on the path from c to u are similar to the vertices from d to v. We therefore conclude that the primary vertices $y \sim z$.

■

We prove a lemma that shows that the weight of deleted and fresh vertices during change propagating can be bounded by the total weight of the vertices of T and T' that do not have a cognate. Since the trace distance is the sum of all vertices of T and T' that have no cognates, the main theorem follows directly from this lemma and the change-propagation theorem (Theorem 7).

Lemma 19 (Cognates versus Fresh and Deleted Vertices)
Let P be a program and let G be the dynamic-dependence graph of P with some input I. Consider changing the input to I' and performing change propagation on G. Let T and T' be the traces of P with inputs I and I' and let $\hat{V}$ and $\hat{V}'$ be the set of vertices of T and T' respectively. If T and T' are respectively monotone, and C is monotone-cognates of T and T', then

1. *the total weight of the vertices deleted during change propagation, denoted W_D, is no more than the total weight of the vertices from $\hat{V}$ that does not have cognate, i.e.,*
 $W_D \leq \sum_{\hat{v} \in (\hat{V} \setminus \{\hat{u} \mid \exists \hat{u}'.\, (\hat{u},\hat{u}') \in C\})} \mathrm{w}(\hat{v})$.
2. *the total of the fresh vertices created during change propagation, denoted W_F, is no more than then the total weight of the vertices from $\hat{V}'$ that does not have cognate, i.e.,*
 $W_F \leq \sum_{\hat{v} \in (\hat{V}' \setminus \{\hat{u}' \mid \exists \hat{u}.\, (\hat{u},\hat{u}') \in C\})} \mathrm{w}(\hat{v}')$.

Proof:

Consider two native vertices $\hat{u}$ and $\hat{v}$ of T and T' respectively. We show that if $\hat{u}$ and $\hat{v}$ are cognates, then none of the primitive vertices that belong to $\hat{u}$ and $\hat{v}$ are fresh or deleted.

The proof relies on the correspondence between the call trees of the native program P and the its normal form P_n as described in Section 3.2.1. Figure 7.4 illustrates the call trees of G and G' respectively and the primitive, and native vertices defined in this proof.

Let u and v denote the primary vertices of $\hat{u}$ and $\hat{v}$ respectively. There are two cases depending on whether u has an ancestor that is re-executed.

In the first case, none of the ancestors of u is re-executed. Since $\hat{u} \equiv \hat{v}$, none of the primitives that belong to $\hat{u}$ are re-executed either, and thus none of the primitives of $\hat{u}$ are deleted and none of the primitives of $\hat{v}$ are fresh.

In the second case, some ancestor of u is re-executed. Let a be the greatest ancestor of u that is re-executed during change propagation and let b denote the vertex created by the re-execution of a.

Let $\hat{a}$ and $\hat{b}$ denote the native vertices for a and b respectively. Since a is re-executed, the native of a, $\hat{a}$, and the native of b, $\hat{b}$, are similar, by Lemma 18. Since the traces T and T' are monotone, and $\hat{a} \sim \hat{b}$ and $\hat{u} \sim \hat{v}$, and $\hat{a}$ is an ancestor of $\hat{u}$, it follows that $\hat{b}$ is ancestor of $\hat{v}$.

Since $\hat{u} \equiv \hat{v}$, and since the primaries u and v perform the same function call as $\hat{u}$ and $\hat{v}$ respectively, we know that $u \sim v$. Consider the execution of b during change propagation. Since the traces are monotone, no other primary call in the subtree rooted at b that comes before v can have a similar primary call that comes after u in the preorder traversal. Therefore when v is executed u will be found in the memo. Since $\hat{u} \equiv \hat{v}$ all values read by the primitives of $\hat{u}$ are unchanged in the memory, and therefore no other secondary of $\hat{u}$ is executed. We therefore conclude that no primitives of $\hat{u}$ are deleted, and no primitives of $\hat{v}$ are fresh.

Therefore the total weight of deleted vertices is bounded by the weight of vertices of T that have no cognates, and similarly, the total weight of fresh vertices is bounded by the weight of vertices of T' that have no cognates. ∎

We are ready to prove the main result of this section.

Theorem 20 (Change Propagation and Trace Distance)
Let P be a program for the closure machine and let G be the dynamic dependence graph of P with some input I. Consider changing the input to I' and performing change propagation. Let

1. *T and T' be the traces of P with I and I' respectively,*
2. *m be the maximum size of the active-call queue during change propagation,*
3. *$p(\cdot)$ be the time for priority queue operations,*
4. *N_A be the number of active vertices,*
5. *$d = \delta^{\min}(T, T')$.*

If T and T' are respectively monotone, then the time for change-propagation is bounded by the following,

1. *$O(d + N_A \cdot p(m))$,*
2. *$O(d + d \cdot p(m))$,*
3. *$O(d \cdot p(d))$.*

Proof: Let W_F and W_X be the total weight of fresh vertices that are created during change propagation and the total weight of vertices that are deleted during change propagation. By Lemma 19 and Definition 12, we know that $W_F + W_O \leq \delta(T, T')$. Since the traces are monotone, we know, by Theorem 17, that $W_F + W_O \leq \delta^{\min}(T, T')$. The theorem follows from this inequality and from Theorem 7. ∎

Chapter 8

Trace Stability

This chapter formalizes the notion of trace stability. Informally, a program is said to be $O(f(n))$-stable for some class of input changes Δ, if the distance between the traces for that program before and after a change consistent with Δ is bounded by $O(f(n))$. We show that if a program is $O(f(n))$ stable, then it self-adjusts in $O(f(n) \cdot p(m))$ time, where $p(m)$ is the priority-queue overhead. We show that $p(m) \in O(\log f(n))$, and therefore time for change propagation is $O(f(n) \log f(n))$.

To enable tighter bounds on the priority-queue overhead, we identify two classes of computations, called *bounded-width* computations, and *read-write regular* computations. We show that for these computations, the priority-queue overhead can be reduced to $O(1)$. This yields the improved bound of $O(f(n))$ for change propagation.

8.1 Trace Models, Input Changes, and Trace Stability

We define a trace model as a triple consisting of a machine model, a set of traces, and a trace generator that maps programs and inputs to their traces. For the definition, let $\mathcal{M}$ denote a machine model, $\mathcal{P}$ be the set of valid programs, and $\mathcal{I}$ the set of possible start states (inputs) for the model.

Definition 21 (Trace Model)
A trace model *for a machine $\mathcal{M}$ consists of a set of possible* traces $\mathcal{T}$, *a* trace generator, $T : \mathcal{P} \times \mathcal{I} \to \mathcal{T}$, *that maps the set of all valid programs $\mathcal{P}$ and the set of all possible start states (inputs) $\mathcal{I}$ to traces, and a* trace distance $\delta(\cdot, \cdot) : \mathcal{T} \times \mathcal{T} \to \mathbb{N}$.

Given some trace model, we are interested in measuring the distance between the traces of a program under some class of input changes. We define a class of input changes as follows.

Definition 22 (Class of Input Changes)
A class of input changes *is a relation Δ, whose domain and co-domain are both the input space, i.e., $\Delta \subseteq \mathcal{I} \times \mathcal{I}$. If $(I, I') \in \Delta$, then we say that I and I' are* related *by Δ, and that the modification of I to I' is a* consistent *with Δ.*

For the purposes of analysis, we often partition Δ into disjoint subclasses of the same size for some notion of size. We say that Δ is *indexed*, if Δ can be partitioned into $\Delta_0, \Delta_1, \ldots$ such that $\forall i, j \in \mathbb{N}.\, i \neq$

$j. \Delta_i \cap \Delta_j = \emptyset$ and $\bigcup_{i=0}^{\infty} \Delta_i = \Delta$. We write an indexed Δ as $\Delta = \Delta_0, \Delta_1, \ldots$, where Δ_i denotes the i^{th} partition. In all our uses, Δ is partitioned and indexed according to the size of one of the two inputs. In general, Δ can be indexed in any way. For example, for output-sensitive programs, Δ can be indexed according to the size of change in the output.

As an example, consider a sorting program. For this program, we may want to consider the class of input changes Δ consisting of single insertions. To analyze the stability of such a program under Δ, we will partition Δ into $\Delta_0, \Delta_1, \ldots$ indexed by the length of the first list. Assuming that the lists contain natural numbers, the partition Δ_0 is the infinite set $\Delta_0 = \{([\,], [0]), ([\,], [1]), ([\,], [2]), \ldots\}$.

We define trace stability as a measure of the similarity of traces of a program on inputs related by some class of inputs changes.

Definition 23 (Worst-Case Stability)
For a particular trace model, let P be an program, and $\Delta = \Delta_0, \Delta_1, \ldots$ be an indexed class of input changes. For all $n \in \mathbb{N}$, define

$$d(n) = \max_{(I,I') \in \Delta_n} \delta(T(A, I), T(A, I')).$$

We say that P is S stable for Δ if $d(\cdot) \in S$.

Note that $O(f(\cdot))$ stable, $\Omega(f(\cdot))$ stable, and $\{f(\cdot)\}$ stable are all valid uses of the stability notation.

For randomized programs, it is important to use the same random bits on inputs related by some class of input changes—because otherwise, traces will certainly not be stable. We use $T_r(P, I)$ to denote the trace generator for a randomized program that uses the random bits r on input I. When analyzing stability of randomized programs, we assume that the same random bits are used on inputs related by a class of input changes, and take expectations over all possible choices of the random bits based on some probability distribution on the initial random bits.

We define the *expected distance* between the traces of some program P on inputs I and I' with respect to some probability-density function ϕ on the initial random bits, written $E_\phi[\delta(T(P, I), T(P, I'))]$, as

$$E_\phi[\delta(T(P, I), T(P, I'))] = \sum_{r \in \{0,1\}^*} \delta(T_r(P, I), T_r(P, I')) \cdot \phi(r).$$

Based on this notion of expected distance, we define the expected-case stability as follows.

Definition 24 (Expected-Case Stability)
For a particular trace model, let P be a randomized program, and $\Delta = \Delta_0, \Delta_1, \ldots$ be an indexed class of input changes, and let $\phi(\cdot)$ be a probability-density function on random bit strings $\{0, 1\}^*$. For all $n \in \mathbb{N}$, define

$$d(n) = \max_{(I,I') \in \Delta_n} E_\phi[\delta(T(P, I), T(P, I'))].$$

We say that P is expected S stable for Δ and ϕ if $d(\cdot) \in S$.

Note that expected $O(f(\cdot))$ stable, $\Omega(f(\cdot))$ stable, and $\{f(\cdot)\}$-stable are all valid uses of the stability notation.

All of our trace-stability results apply to monotone programs that are defined as follows.

Definition 25 (Monotone programs)
We say that P is monotone with respect to some class of input changes Δ, if P is concise and if for any $(I, I') \in \Delta$, the traces $T = T(P, I)$ and $T' = T(P, I')$ are respectively monotone.

8.2 Bounding the priority queue overhead

This section presents two techniques for bounding the priority-queue overhead of change propagation.

The first technique introduces the notion of *dependence width* between two computations. We show that, for monotone programs, the dependence width gives a bound on the size of the priority queue during a change propagation. This result implies that if the dependence width of a computation is constant, then the priority-queue overhead of that execution is constant even when using a general purpose priority-queue data structure. Many of the applications that we consider in this thesis have constant dependence width under a constant-size change to the input.

The second technique introduces a class of computations called *read-write regular*. The memory reads and writes in a read-write-regular computation exhibit a particular structure. By exploiting this structure, we show that it is possible to use a constant-time first-in-first-out (FIFO)-based queue for change propagation instead of a general purpose queue. Since all queue operations take constant time, the priority queue overhead for read-write regular computations is constant.

8.2.1 Dependence width

Let P be a program and I be an input for P. We define the *ordered vertex set* of the execution of P with I, denoted $V(P, I)$ as the set of vertices in the trace of the execution $T = T(P, I)$ ordered with respect to their execution times. The ordering on the vertices is also given by a preorder traversal of T. For the rest of this section, we treat $V(P, I)$ as an ordinary set and use the terms "come after/before" when referring to the ordering relation between the vertices.

For an execution of P with input I, we define the memory, denoted $\sigma(P, I)$, as the function that maps all locations created during the execution to their values. Given a vertex $v \in V(P, I)$, we define the *prefix (memory)* of a memory σ at v, denoted σ_v as a subset of σ, $\sigma_v \subseteq \sigma$, that consists of location value pairs $(l, a) \in \sigma$, such that l is written before v starts to execute; more precisely l is written during the execution of some vertex that comes before v.

Consider some program P that is monotone under some class of input changes Δ. Let $(I, I') \in \Delta$, be a pair of inputs, and define $V = V(P, I)$, $V' = V(P, I')$. We define the ordered set of *shared vertices* of V and V', denoted $S(V, V')$ as the set $S(V, V') = \{(v, v') \mid v \in V \wedge v' \in V' \wedge v \sim v'\}$, where the pairs are ordered with respect to one of their elements. That is for any pair of vertices $(u, u') \in S(V, V')$, and $(v, v') \in S(V, V')$, (u, u') comes before (v, v') if u comes before v or u' comes before v'. We call a pair $(v, v') \in S(V, V')$ a *shared pair*, and say that each of v and v' is *shared*.

Note that shared vertices are defined only for monotone programs. To see that the total order on the shared vertices is well defined, recall that executions of monotone programs do not contain two similar vertices, and that the relative order of similar vertices in two executions remains the same. More precisely, since P is monotone for Δ, the set V contains no distinct vertices $u, v \in V$ such that $u \sim v$; the same holds for V'. Therefore, the shared pairs each vertex from V and V' with at most one other vertex. Also for any

$u, v \in V$, $u', v' \in V'$ such that $u \sim v$ and $v \sim v'$, the ordering of u and v is the same as the ordering of u' and v'; if u comes before v, then u' comes before v'. Therefore given any $(u, u') \in S(V, V')$ and $(v, v') \in S(V, V')$, we know that either u comes before v (and u' comes before v'), or u comes after v (and u' comes after v').

For two executions, we define the set of affected locations at a shared vertex as follows.

Definition 26 (Affected Locations)
Let P be a program that is monotone for some class of input changes Δ. Consider the executions of P with inputs I and I' and let $V = V(P, I)$, $V' = V(P, I')$, $S = S(V, V')$, $\sigma = \sigma(P, I)$, and $\sigma' = \sigma(P, I')$. The set of affected locations *at $(v, v') \in S(V, V')$, denoted $A(\sigma_v, \sigma'_{v'})$, consists of the locations that have different values in σ_v and $\sigma'_{v'}$, i.e., $A(\sigma_v, \sigma'_{v'}) = \{l \mid l \in \mathit{dom}(\sigma_v) \wedge l' \in \mathit{dom}(\sigma'_{v'}) \wedge \mathit{label}(l) = \mathit{label}(l')\}$.*

As an example, consider the prefix memories $\sigma_v = \{(l_1, 1), (l_2, 2), (l_3, 3))\}$ and $\sigma'_{v'} = \{(l'_1, 0), (l'_2, 2), (l'_3, 4))\}$. Suppose that the label of l_i is equal to the label of l'_i for $i = 1, 2, 3$, then the set of affected locations at v is $\{l_1, l_3\}$.

Definition 27 (Active Vertices)
Let P be a monotone program for Δ and consider the executions of P with inputs I and I', $(I, I') \in \Delta$. Let $V = V(P, I)$ and $V' = V(P, I')$ and $S = S(V, V')$. Let $(u, u') \in S$ be a shared pair and let $(v, v') \in S$ be the first shared pair that comes after (u, u') in S. We say that a vertex w, $w \in V$, is active at (u, u'), if w reads a location that is affected at (v, v') and w comes after u.

Note that the definition is not symmetric, it is defined only with respect to the first set of vertices (V).

For two executions, we define the dependence width of a as the maximum number of active vertices over all shared vertices.

Definition 28 (Dependence Width)
Let P be a program that is monotone for some class of input changes Δ and consider the executions of P with inputs I and I', $(I, I') \in \Delta$. Let $V = V(P, I)$ and $V' = V(P, I')$. The dependence width of the executions is the maximum number of affected vertices over all shared vertices $S(V, V')$.

Consider the execution of P with I and suppose that we change the input to I' and perform a change propagation. We show that the size of the priority queue during change propagation can be asymptotically bounded by the dependence width of the executions with I and I'.

Theorem 29 (Dependence Width and Queue Size)
Let P be a program monotone for some class of input changes Δ and consider executing P with input I and changing the input to I' and performing a change propagation. The size of the priority queue is asymptotically bounded by the dependence with of the executions of P with I and I'.

Proof:

Since change propagation is only defined for programs written in the normal form, we need to consider the primitive versions of native programs. Note first that the executed `alloc`, `write`, and `read` operations of a native program and its primitive are identical except for the operations that allocate, read, and write destinations. Recall that destinations are created by the primitive programs to force all functions return their values through memory.

We first show that the size of the priority-queue is increased by at most a constant due to operations performed on destinations. The normal-form transformation ensures that destinations 1) are read immediately after they are written, 2) are read by one primitive function, and 3) are read exactly once. These properties ensure that there is at most one function call that reads an affected destination, and that there is at most one affected destination whose readers are not re-executed. The memory operations on destinations can therefore increase the size of the priority queue at most by one. For the rest of the proof, we therefore consider operations performed by both the native and the primitive programs.

Since only insertions into the queue increase the size of the priority queue, we will only consider insertions. For the rest of the proof, let $\hat{V} = V(P, I)$, $\hat{V}' = V(P, I')$, and $\hat{S} = S(V, V')$.

Consider the execution of a `write` instruction that inserts a vertex into the priority queue. Let v' be the vertex currently being executed and let $\hat{v}'$ be the native of v'. Let $(\hat{u}, \hat{u}') \in S$ and $(\hat{w}, \hat{w}') \in S$ be shared vertices such that $\hat{u}'$ is the latest shared vertex that comes before $\hat{v}'$ and $\hat{w}'$ is the first shared vertex that comes after $\hat{v}'$ such that $\hat{w}' \neq \hat{v}'$. Let u, u' be the primaries of $\hat{u}$ and $\hat{u}'$, and w, w' be the primaries of $\hat{w}$ and $\hat{w}'$.

Consider the set of all affected locations A during the execution of the `write` instruction. Since v comes before w, and since the set of affected locations grows monotonically during change propagation, we know that $A \subseteq A(\sigma_w, \sigma'_w)$. Since v comes after u, and since the priority queue only contains vertices that come after v, the vertices in the priority queue are the primitives of $\hat{u}$ or the primitives of the vertices that come after $\hat{u}$. Therefore, the vertices in the priority queue belong to the active vertices at $(\hat{u}, \hat{u}')$.

Since each native vertex has a constant number of primitive vertices, and since the dependence with of the executions is defined as the maximum number of active vertices over all shared pairs in S, the lemma follows.

■

Definition 30 (Constant-Width Programs)
Let P be a program that is monotone with respect to a class of inputs changes Δ. The program has constant width for Δ, if for any $(I, I') \in \Delta$, the dependence width of the executions of P with I and I' is asymptotically bounded by a constant.

By Theorem 29, we know that the priority-queue overhead for constant-width programs is constant.

Theorem 31 (Priority-Queue Overhead for Constant-Width Programs)
Let P be a program that is monotone with respect to a class of inputs changes Δ and suppose that P is constant-width for Δ. The priority queue overhead of change propagation for P under Δ is constant.

8.2.2 Read-Write Regular Computations

We show that for read-write regular computations, a $O(1)$-time, FIFO-like queue can be used for change propagation instead of a general-purpose logarithmic-time priority queue.

Definition 32 (Read-Write Regular Computation)
A program P with input domain $\mathcal{I}$ is read-write regular for a class of input changes $\Delta \subseteq \mathcal{I} \times \mathcal{I}$, if the following conditions are satisfied:

1. *in an execution of P with any input $I \in \mathcal{I}$, all locations are read by a constant number of times,*
2. *for an execution of P with any input $I \in \mathcal{I}$, if a location l_1 is written before another location l_2, then all function calls that read l_1 start to execute before any function call that read l_2.*
3. *for any $(I, I') \in \Delta$, if l_1 and l_2 are locations allocated in both the executions of P with I and I', then l_1 and l_2 are written in the same order in both executions.*

For read-write regular computations, we will use a specialized version of the change-propagation algorithm that inserts active function calls into the priority queue in sorted order. In particular, when a location is changed, the algorithm will first sort all function calls that read that location and insert them into the priority queue in that order. Since each location is read by a constant number of times, sorting the function calls does not change the asymptotic running time.

We show that if a program is read-write regular for some class of input changes, then the specialized change-propagation algorithm can use a constant-time priority queue. The priority queue, called *FIFOP* (FIFO with Priorities), extends a standard first-in-first-out (FIFO) queue with priorities. Deletions from a FIFOP queue only occur from the front of the queue as with FIFO queues. Insertions are slightly different. To insert an item i, we compare the priority of i and the item f at the front of the queue. If the priority of i is less than that of f, then i is inserted at the front. Otherwise, i is inserted at the tail of the queue.

Theorem 33 (Priority Queue Overhead for Read-Write-Regular Computations)
If a program is read-write regular for some class of input changes Δ, then the priority queue overhead for change-propagation under Δ is $O(1)$.

Proof: Consider the normal form P_n of P obtained from the normal form transformation described in Section 3.2. We will show that a FIFOP queue with priorities as time stamps suffices for change propagation with P_n. We first show that the following invariant holds: during change propagation, the priorities of the items in the FIFOP queue increase from the front to the tail of the queue.

The invariant holds at the beginning of change propagation, because the priority queue is initialized by inserting the function calls reading the changed locations in sorted order. Assume now that the invariant holds at some time during change propagation. We will show that it holds after the next remove or insert operation. Since the remove operation deletes the item at the front of the queue, the invariant holds trivially.

For insertions, we will distinguish between two kinds of locations, *destination* that are allocated only by P_n for the purposes of supporting function-call returns, and all other locations. Consider an insertion. In change propagation, an insertion occurs only when some location, say l, is written. Let f be the front of the queue and let t be the item at the tail of the priority queue, and let i be the item being inserted. If t and i both read l, then l is not a destination, because destinations are read by exactly one function call. Since function calls reading the same location are all inserted in order of their time stamps, the time stamp of i is greater than the time stamp of t. Item i will therefore be inserted immediately after t. In this case the invariant is satisfied. Suppose now that t and i read different location. Let l' be the location read by t. We will consider two cases, depending on whether l is a destination or not. If l is not destination, then we know that l' is written before l. Since the program is read-write regular, all read of l' start before the reads of l. Therefore, the time stamp of i is greater than that of t. In this case, i will be inserted at the tail of the queue, and the invariant is satisfied. Suppose now l is a destination. Since a destination is written only at the end of some function call, and read immediately after that function call returns, none of the reads that are currently in the queue can have a time stamp that is less than that of i. Therefore, i will be inserted at the front of the queue and the invariant is satisfied.

As a result of this invariant, we know that using this FIFOP queue instead of a general priority queue is correct. Since all locations are read by a constant number of function calls, the change propagation algorithm will be slowed down by a constant factor when using a FIFOP queue. Since inserts and removes with FIFOP queues take constant time, the theorem follows. ■

8.3 Trace Stability Theorems

This section presents two trace-stability theorems that bound the time for change propagation in terms of the trace stability of monotone programs.

Theorem 34 (Worst-Case Trace Stability)
Consider the trace model for closure machine consisting of all set of traces and the minimum trace-distance measure $\delta^{\mathrm{min}}(\cdot,\cdot)$. If a program P in the closure model is monotone for a class of input changes Δ and P is $O(f(n))$-stable for Δ, then P self-adjusts for Δ in $O(f(n)\log f(n))$ time. In general, if the priority-queue overhead is bounded by $O(p(m))$, then P self-adjusts in $O(f(n)\,p(m))$ time. If P is read-write regular, or if P has constant-width under Δ, then P self-adjusts for Δ in $O(f(n))$ time.

Proof: Consider a program P that is $O(f(n))$ stable for some class of input changes Δ. Let T and T' be the traces of P on inputs I and I' where $(I, I') \in \Delta$. Since P is $O(f(n))$ stable, $\delta^{\mathrm{min}}(T, T') \in O(f(n))$. The general case bound follow directly from Theorem 20 and from the fact that the priority queue overhead is bounded by the logarithm of the size of the queue. By Theorems 31 and 33, the priority-queue overhead for constant-width and read-write regular computations is bounded by constant. The bounds for these computations is therefore equal to the stability bound itself. ■

Theorem 35 (Expected-Case Trace Stability)
Consider the trace model for closure machine consisting of all set of traces and the minimum trace-distance measure $\delta^{\mathrm{min}}(\cdot,\cdot)$. Let P be a monotone program with respect to a class of input changes Δ. If P is expected $O(f(n))$ stable for Δ and the priority-queue overhead is bounded by $O(1)$, then P self-adjusts under Δ in expected $O(f(n))$ time. In particular, if P is read-write regular for Δ, or if P has constant-width under Δ, then P self-adjusts under Δ in expected $O(f(n))$ time.

Proof: Consider a program P that is expected $O(f(n))$ stable for some class of input changes Δ. Let T and T' be the traces of P on inputs I and I' with the same initial random bits where $(I, I') \in \Delta$. If the priority-queue overhead is bounded by $O(1)$, then the time for change propagation is bounded by $O(\delta^{\mathrm{min}}(T, T'))$ by Theorem 20. Since P is expected $O(f(n))$ stable, $\delta((,T), T')$ is expected $O(f(n))$, where the expectation is taken over all initial random bit strings r. Change propagation therefore takes expected $O(f(n))$ time. The special case for constant-width and read-write regular programs follows by Theorems 31 and 33. ■

Part III

Applications

Introduction

This chapter proves trace-stability results for a range of algorithms under insertions and deletions to their inputs. Based on these bounds and the trace-stability theorems (Theorems 34 and 35), we obtain time bounds for change-propagation under the considered classes of input changes. Our bounds are randomized and match the best known bounds within an expected constant factor.

We consider a number of algorithms on lists including combining of data in a list with an arbitrary associative operator, the merge sort and quick sort algorithms, and the Graham's Scan algorithm for planar convex hulls. In Chapter 9, we study the list algorithms. As an algorithm on trees, we consider, in Chapter 10, tree-contraction algorithm of Miller and Reif [62, 63]. To show the effectiveness of the technique, we choose algorithms that span a number of design paradigms including random sampling (list combination), two forms of divide and conquer (merge sort and quick sort), incremental result construction (the Graham's Scan algorithm), and bottom-up, data-parallel computing (tree-contraction).

Chapter 9

Algorithms on Lists: Combining, Sorting, and Convex Hulls

This chapter considers a number of algorithms on lists including an algorithm for combining the values in a list with an arbitrary associative operator (Section 9.1), merge sort (Section 9.2), quick sort (Section 9.3), and the Graham's Scan algorithm for convex hulls (Section 9.4). For each considered algorithm, we show a bound that is within an expected constant factor of the best known upper bound for the corresponding problem.

9.1 Combining with an Arbitrary Associative Operator

We consider combining the elements of a list using any associative operator. We do not require commutativity or an inverse. In addition to the obvious operators such as addition and maximum, combining can be used for many other applications including finding the longest increasing sequence or evaluating a linear recurrence.

Scanning the list from the head is not stable because inserting an element anywhere near the front of the list can propagate through all the rest of the partial sums. Building a tree will alleviate this problem, but it is important that the tree structure is balanced and stable with respect to insertions and deletions to arbitrary positions in the list. Any deterministic method for building the tree based on just list position is not going to be stable—a single insert can change everyone's position. Instead we use a randomized approach.

The code for the approach is shown in Figure 9.1. It is based on a function `HalfList` which takes a list of pairs consisting of a value and a key, and returns a list of key-value pairs which is expected to be half as long. We assume that the keys are unique. For each key in the input, `HalfList` flips an unbiased coin an keeps the key if the coin comes up heads. The value of a kept key is the partial sum of all input keys from the kept element up to, but not including, the next kept key. To account for keys up to the first kept key, `HalfList` also returns the partial sums of those keys. `HalfList` is applied repeatedly in `Combine` until the list is empty.

To generate random coin flips, the code uses a family H of random-hash functions. The family H is a collection of 2-wise independent hash functions mapping the identifiers to $\{0,1\}$. We randomly select one

```
function Combine (l, round)
  case *l of
    nil =>
      return Identity
  | cons(_, _) =>
      (v, l') ← HalfList(l, round)
      v' ← Combine(l', round + 1)
      return (v + v')

function HalfList(l, round) =
  case *l of
    nil =>
      return (Identity, l)
  | cons((key, v), t) =>
      (v', t') ← HalfList(t, round)
      if binaryHash(key, round) = 1
        l' ← allocate{key, round}(3)
        *l' ← cons((key, v + v'), t')
        return l'
      else
        return (v + v', t')
```

Figure 9.1: Combining with the associative operator + and with identity element `Identity`.

member of H per round. Since there are families H with $|H|$ polynomial in n [97], we need $O(\log^2 n)$ random bits. We analyze trace stability assuming fixed selection of random bits and take expectations over all selections of bits. For fixed random bits, the algorithm will generate the same coin flip on round r for any key k in any two executions. This ensures that the trace remains stable in different executions.

9.1.1 Analysis

Consider the class of single insertions or deletions, denoted Δ, to be the pair of inputs that differ by a single insertion or deletion. We show that the algorithm is expected $O(\log n)$-stable for Δ where the expectation is over the initial randomness.

To simplify the analysis and to establish an analogy to *skip lists* [82], we consider a slightly different version of the `Combine` function. We assume that the input list contains a head item that initially contains the identity element. We refer to this cell and its copies as the *sentinel*. The modified `Combine` functions calls `HalfList` with the tail of the first cell and adds the value returned by `HalfList` to the value stored in the sentinel. No modifications are required to `HalfList`. The modified `Combine` code is shown below.

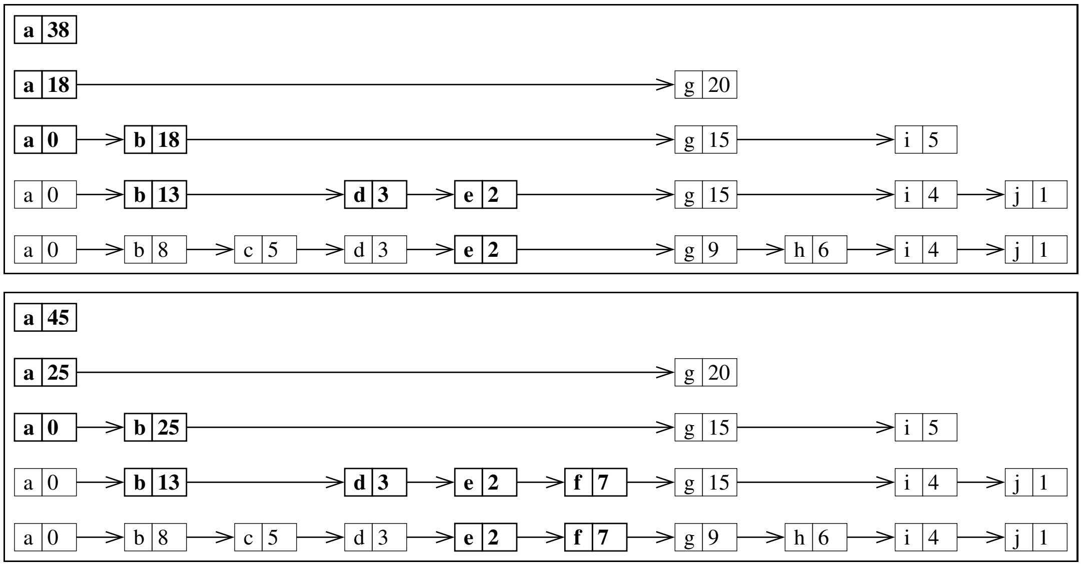


Figure 9.2: The lists at each round before and after inserting the key f with value 7.

```
function Combine(cons((key,v),t),round)
  case (*t) of
    nil => return
  | cons(_,_) =>
      (v',t') ← HalfList(t,round)
      Combine(cons((key,v+v'),t'),round+1)
      return
```

As an example, Figure 9.2 shows an execution of `Combine` with the lists $[(a,0),(b,8),(c,5),(d,3),(e,2),(g,9),(h,6),(i,4),(j,1)]$, and $[(a,0),(b,8),(c,5),(d,3),(e,2),(\mathbf{f},\mathbf{7}),(g,9),(h,6),(i,4),(j,1)]$, which differ by the key f, with value 7. The keys are $a,\ldots,k$. The head of both lists is the sentinel cell with key a.

The trace for the algorithm consists of calls to `Combine` and `HalfList`. Recall that the calls are tagged with the arguments, the values read, and value returned from subcalls. The calls to `Combine` are tagged with the round number, the value of the first key, the second key (if any), and the value and the list returned from the call to `HalfList`. The second key is needed because the tail is labeled with that key. The calls to `HalfList` are tagged with the round number, first cell in the input, and the value and the list returned from the recursive call.

Theorem 36
The combine application is monotone for single insertions and deletions.

Proof: First note that all memory allocations are labeled. Since all calls to `HalfList` take the round number as an argument and the list, and since the list cells are labeled with the key of the first cell, calls to `HalfList` are unique. Calls to `Combine` are also unique, because `Combine` is called exactly once in each round, and each call to `Combine` takes the round number as an argument. The program is therefore concise.

For the monotonicity of execution order, and calling relationships, we consider the calls to `Combine` and `HalfList` separately. Since `Combine` calls `HalfList` deterministically exactly once in each round, the interaction between the two obeys the monotonicity requirements.

Both the calling order and the caller-callee relationships between `HalfList` calls in each round are determined by the ordering of keys in the input list. Since a single insertion/deletion does not alter the relative order of existing keys, the calls to `HalfList` satisfy monotonicity properties. Note that calls to `HalfList` that occur in two different rounds are independent. The calling-order and the caller-callee relationship between calls to `Combine` are determined by the round numbers and therefore satisfy monotonicity properties. ∎

For our analysis, we consider insertions only. Since the application is monotone, the bound for insertions and deletions are the same. To analyze the stability of `Combine`, we consider executing `Combine` with an input I with n keys and I' that is obtained from I by inserting a new key α^*, and count the number of calls to `Combine` and `HalfList` that do not have a cognate.

Throughout the analysis, we denote the executions with I and I' as X and X' respectively. We denote the number of rounds in X and X' as R and R' respectively. Given some round r, we denote the input list to that round in X and X' as I_r and I'_r respectively. We first show a property of the lists in each round.

Lemma 37 (Lists differ by at most α^*)
The execution X and X' satisfy the following properties.

1. $R \leq R'$,

2. *for any round $r \leq R$, I'_r consists of the same keys as I_r plus perhaps α^*, and*

3. *for any round $R < r \leq R'$, the input I'_r consists of the sentinel and perhaps α^*.*

Proof: Whether a key is kept at one round is determined entirely by the key and by the random hash function being used. Since the executions X and X' are performed with the same initial randomness, any key α will see the same random bits in both executions. Therefore, the lists I_r and I'_r will only differ by the newly inserted key α^*, for any $r \leq R$. Therefore $R \leq R'$. Since all keys from I will be deleted at the end of round R, the lists I'_r will consists of the sentinel key and perhaps α^*, if α^* has not yet been deleted. ∎
For example in Figure 9.2, the lists in each round contain the same keys except for the key f.

We show that the distance between the traces of X and X' can be determined by counting the number of affected keys.

Definition 38 (Affected Keys)
We say that a key α is affected at round r, if α is in at least one of I_r or I'_r in that round, and any one of the following hold

1. *α is affected at some previous round,*

2. *$\alpha = \alpha^*$ or α comes immediately before α^* in I'_r,*

3. *the key β that comes immediately after α in I_r and I'_r is affected, and β is deleted in round r.*

4. *α is the sentinel key and $r > R$.*

Note that the third possibility is well defined because if $\beta \neq \alpha^*$, then β is the same in both I_r and I'_r and since the decision to delete β is made solely on the round number, it will either be deleted (or not deleted) in both executions. In the example shown in Figure 9.2, the cells containing the affected keys are highlighted.

We show that if a key has two different values in two executions, then it is affected.

Lemma 39 (Affected Keys and Values)
If α is a key that is in both I_r and I'_r, and the values of α in the two lists are different, then α is affected at round r.

Proof: The proof is by induction on the number of rounds. The lemma holds trivially in the first round, because all keys that are present in both input lists have the same values. Suppose now the lemma holds at some round $r - 1 > 1$ and consider round r. Let α be some key that is present in both lists I_r and I'_r with different values. If this is not the first round in which the values of α differ, then by the induction hypothesis, α is affected at some previous, and therefore, at this round. Suppose that this is the first round in which the values of α differ. Consider the set of key $\beta_1 \ldots \beta_m$ in I_r that come after α and that are deleted in X, and $\beta'_1 \ldots \beta'_n$ in I'_r that come after α and deleted in X'. Note that, by Lemma 37, $\{\beta'_1, \ldots, \beta'_n\} \setminus \{\beta_1, \ldots, \beta_m\} \subseteq \{\alpha^*\}$. Since the values assigned to α are determined by the sum of these values, the two sums must differ. Therefore, $\beta'_n = \alpha$, the value associated with β_i for some i is different in I_r and I'_r. In both cases α is affected at round r by Definition 38. ∎

Lemma 40 (Affected Keys and Trace Distance)
Consider the traces of `Combine`, T and T', with the input lists I and I', where I' is obtained from I by inserting some key α^*. The monotone distance between T and T', $\delta^{\min}(T, T')$, is bounded by the total number of affected vertices summed over all rounds.

Proof: Since the traces are monotone, $\delta^{\min}(T, T')$ is equal to the number of calls from T and T' that do not have a cognate in the other trace. To bound this quantity, we will consider the executions X and X' of `Combine` with I and I' respectively, and show that each call without a cognate is uniquely associated with an affected key. We will consider the calls to `Combine` and `HalfList` separately. For the proof, let R and R' denote the number of rounds in X and X' respectively.

Consider the calls to `Combine`. Recall that the calls to `Combine` are tagged with the round number, the value of the first key, the second key (if any), and the value and the list returned from the call to `HalfList`. Consider a call to `Combine` that does not have a cognate, and let r be the round of this call. We will show that the sentinel (first) key is affected in this round. If $r > R$ then, this holds trivially by Definition 38. Suppose that $r \leq R$ and assume that the sentinel is not affected. By Lemma 39, the values of the sentinels are the same in both I_r and I'_r, and, by Definition 38, the keys β and β' that come after the sentinel in I_r and I'_r respectively are the same. Furthermore, β is either not deleted, or it is not affected at round r. If β is not deleted, then the call will be returned the identity value and the same lists in both X and X'. The calls to `Combine` will therefore be cognates. If β is deleted, then β is not affected. We therefore know that all deleted keys that contribute to the value returned by β are not affected. In this case too, the call will be returned the same value and the same list in X and X' and the call are cognates. In both cases, we reach a contradiction. Therefore we conclude that the sentinel is affected in round r.

Consider the calls to `HalfList`. Recall that the calls are tagged with the round number, first cell in the input, and the value and the list returned from the recursive call. Consider a call to `HalfList` that does not have a cognate and let r be the round of this call and α be the first key. We will show that α is affected in round r. Assume, for the purposes of contradiction, that α is not affected. Since α is not affected, $\alpha \neq \alpha^*$, and the values of α in I_r and I'_r are the same, and the keys next to α in both lists are the same key β. Furthermore, β is either not deleted in both X and X', or it is not affected. If β is not deleted, then α will be returned the same value and the same result cell, and thus the call to `HalfList` in two executions will be cognates. Suppose that β is deleted. Since we know that β is not affected, we know that all deleted keys that contribute to the value returned to β are the not affected. Since unaffected keys have the same values, α will be returned the same value. Since a key that immediately precedes α^* is affected, α will also be returned the same cell. We conclude that the calls to `HalfList` with key α are cognates. In both cases, we reach a contradiction. We therefore conclude that α is affected. ■

Based on the definition of affected keys, it can be shown that the number of affected keys after a single insertion is bounded by $O(\log n)$. In fact, we set up the algorithms so that there is an isomorphism between the lists from each round and the skip-lists data structure [82]. In particular, if the key α^* is inserted into list l, then the set of affected keys is the same as the keys that are compared to α^* when performing an insertion into the skip lists data structure that contains the keys in l. Since, in skip lists, insertion time is bounded by expected $O(\log n)$, we have the following theorem.

Theorem 41 (Stability of Combine)
The `Combine` *code is $O(\log n)$ stable with respect to insertions or deletions in the input list.*

To obtain a complexity bound on change-propagation all that remains is to bound the priority-queue overhead. This is straightforward, because the algorithm is read-write regular—each location is read at most once and if a location is written before another, then all its reads are performed before the other. By Theorem 35, it follows that the function `Combine` self-adjusts to any insertion/deletion in expected $O(\log n)$ time.

Theorem 42 (Self-Adjusting List Combine)
The `Combine` *code self-adjust to insertions/deletions in expected $O(\log n)$ time.*

9.2 Merge sort

We consider a randomized version of the merge sort algorithm, and analyze its stability. The deterministic merge sort algorithm splits its input list evenly in the middle into two sublists, recursively sorts the sublists, and merges them. This is not stable, however, because inserting/deleting a key from the input can change the input to both recursive calls due to deterministic splitting. We therefore consider a randomized version that splits the input by randomly deciding which list each key should be placed.

The code for merge sort with randomized splitting is shown in Figure 9.3. The function `Split` splits a list into two lists by randomly deciding the lists that each key should be placed. For randomization, the

```
function Split(c) =
  case c of
    nil => return (nil, nil)
  | cons(h, t) =>
      (left, right) ← Split(*t)
      new ← alloc{c}(2)
      if binaryHash(h) then
        *new ← left
        return (cons(h, new), right)
      else
        *new ← right
        return (left, cons(h, new))

function Merge(ca, cb) =
  case (ca, cb) of
    (nil, _) => return cb
  | (_, nil) => return ca
  | (cons(ha, ta), cons(hb, tb)) =>
      t ← alloc{ha, hb}(2)
      if ha < hb then
        *t ← Merge(*ta, cons(hb, tb))
        return cons(ha, t)
      else
        *t ← Merge(cons(ha, ta), *tb)
        return (cons(hb, t))

function MSort(c) =
  if length(c) < 2 then
    return c
  else
    (ca, cb) = Split(c)
    return (Merge(MSort(ca), MSort(cb)))
```

Figure 9.3: Merge Sort

function relies on a family H of pairwise independent binary hash functions. We select one member of the family for each round (recursion depth of `MSort`). Since there are families H with $|H|$ polynomial in n [97], we need $O(\log^2 n)$ random bits in expectation. The `merge` function takes two lists and merges them in the standard manner.

9.2.1 Analysis

Consider the class of single insertions or deletions, denoted Δ, to be the pair of inputs that differ by a single insertion or deletion. We will show that the merge sort algorithm is expected $O(\log n)$-stable for Δ where the expectation is over all initial random bits used to select the random hash functions for randomized splitting.

The trace for merge sort consists of calls to `MSort`, `Merge` and `Split`. Each call to `MSort` is tagged with the first cell of the input and the two cells returned by the subcall to `Split`. Each call to `Merge` is tagged with the first cells of the two inputs, and first cell of the sorted list returned by the recursive call. Note that the reads of the tails do not contribute to the tag, because the read values are only passed to a subcall. The calls to `Split` are tagged with the first cell of the input and the two cells returned by the recursive call. Note that reading the tail does not contribute to the tag because the read value is only passed to the recursive call.

Since none of the functions are called on the same list cell(s) more than once, the algorithm is concise. It is also straightforward to show that merge sort is monotone for Δ based on the following properties
1) the execution order of calls to `Split` are determined by the order of keys in the input list,
2) the execution order of calls to `Merge` are determined by the sorted order of keys in the input list, and
3) the execution order of calls to `MSort` is determined by the recursion level and the order of keys in the input.

Since both `Merge` and `Split` are singly-recursive function calls, the caller-callee relationships are determined by their execution order. Since one insertion/deletion into/from the input does not swap the locations of two keys in the input, the calls to `Merge` and `Split` satisfy the monotonicity requirements. Inserting/deleting one key may introduce one new call to `MSort` at each level without swapping the order of calls along any recursive calls path. Since one insertion/deletion does not change the relative execution order of other keys, the calls to `MSort` satisfy the monotonicity requirements. The merge sort application is therefore monotone for one insertion/deletion.

Since merge sort is monotone, stability bounds with respect to insertions and deletions are symmetric. We will therefore consider insertions only. For the rest of this section, consider two lists I and I', where I' is obtained from I by inserting α^*. Let X and X' denote the executions, and T and T' denote the traces of merge sort with I and I' respectively.

For the analysis, we will divide the execution into rounds based on the recursion level of the `MSort` calls. The first round corresponds to the top call to `MSort` and contains the top-level call to `Split` and `Merge`. The second level corresponds to the two recursive calls of the top-level `MSort` and the corresponding calls to `Split` and `Merge`. For each round, we show that the number of function calls without cognates is constant in expectation.

Consider calls to `Split` in X and X'. The key α^* is the first cell of one call to `Split` in each round. Furthermore α^* is returned to an expected two calls in each round. All the other call have the same tags. Therefore calls to `Split` contribute an expected three to the trace distance in each round. Consider calls to `MSort`. Since `MSort` is called at once with a list that contains α^* in each round, call to `MSort` add at most one per round to the distance between T and T'. Since the expected number of rounds in both X and X' is $O(\log n)$ where n is the number of keys in I, the calls to `Split` and `MSort` contribute $O(\log n)$ to the trace distance.

We show that the contributions of calls to `Merge` at each round is also expected constant. Consider some call to `Merge` that does not have a cognate. There are several possibilities to consider. One is that the arguments to `Merge` are different. In this case, the cells are either created by a `Merge` function that performs a comparison that takes place only in one execution, or the cells themselves contain keys that have never been compared. In this case, we charge this cognate to the comparison. In the second case, the call is returned a list that is different than before. This means that either the recursive call performs a comparison that take place only in the one execution or it callee does. In this case too, we charge the cognate to the comparison. Since each `Merge` performs one comparison and allocates one cell, each comparison is charged a constant number of times. To analyze the stability of `Merge`, it suffices to count the number of comparisons that differ in X and X'. We first show the key sampling lemma behind the analysis.

Lemma 43
Consider merging two sorted lists that are sampled randomly from a list by flipping a coin for each key and putting together the keys with the same outcome. The expected number of comparisons that a key is involved in is 2.

Proof: Since input lists to merge are selected randomly, each key is equally likely to be placed in either of the inputs to merge. Fix some input list to `merge` and let k be in the list and let k' be its predecessor. The expected number of keys in the sorted output list between k and k' is 1 (since all keys in between must be selected to the other list). Therefore, the expected number of times that k is compared is 2. ■

Since the inputs I and I' differ by a single key, and since each key is input to exactly one `Merge` call in each round, the number of comparisons that differ between the two executions is expected constant per round. Since the expected number of rounds is $O(\log n)$ in the size of the input, the comparisons differ by expected $O(\log n)$ between X and X'. Therefore, the calls to `Merge` contribute $O(\log n)$ to the trace distance.

Theorem 44 (Merge Sort Stability)
Merge sort with randomized splitting of lists is expected $O(\log n)$ *stable for the class of single insertion/deletion to/from the input list.*

Based on the facts that (1) a single insertion/deletion changes the output of `Split` and the output of `Merge` at a single location by inserting/deleting a key, and (2) each location is read at most two times, we can show that merge sort has constant-width for one insertion/deletion. By Theorem 35, the stability bound yields an equivalent time bound.

Theorem 45 (Self-Adjusting Merge sort)
Merge sort with randomized splitting self-adjusts to insertions/deletions anywhere in the input in expected $O(\log n)$ *time.*

9.3 Quick Sort

We consider one of the many variants of quick sort on a list. It uses the first key of the input as pivot and avoids appends by passing a sorted tail in an accumulator. We note that many other variants would work, including using the last pivot, or a maximal priority element as the pivot. Using the middle element as a pivot would not work since this is not stable—randomly inserting an element has a constant probability of changing it. Figure 9.4 shows the code for the version that we consider. The argument `rest` is the accumulator for the sorted tail, and the location `dest` is where the result will be stored (the destination).

```
function Split(c, p, rest, dest)
  case c of
    nil => (nil, nil)
  | cons(h, t) =>
    (ca, cb) ← Split(*t, p, rest, dest)
    l ← allocate{v, p}(2)
    if h < p then
      *l ← ca
      return (cons(h, l), cb)
    else
      *l ← cb
      return (ca, cons(h, l))

function QSort(c, rest, dest)
  case c of
    nil => *dest ← rest
  | cons(h, t) =>
      (ca, cb) ← Split(c, p, rest, dest)
      mid ← allocate{h}(2)
      QSort(cb, mid, rest)
      QSort(ca, cons(h, mid), dest)
```

Figure 9.4: The code for quick sort

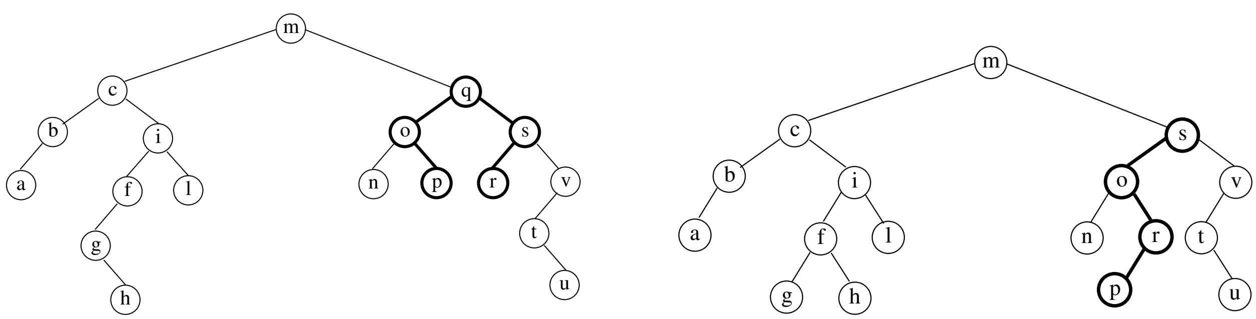


Figure 9.5: Pivot trees for quick sort before and after the deletion of `q`.

9.3.1 Analysis

Consider the class of single insertions or deletions, denoted Δ, to be the pair of inputs that differ by a single insertion or deletion. We will show that quick sort is expected $O(\log n)$-stable for Δ where the expectation is over all permutations of the input.

The analysis relies on an isomorphism between quick sort and random search trees, such as Treaps [90]. Consider the call-tree formed by executed `QSort` calls and label each vertex (call) with the pivot of that call, we call this the *pivot tree* of the execution. It is a property of quick sort that the pivot tree is isomorphic to the binary search tree obtained by starting with an empty tree and inserting the keys in the order specified by the input list. The pivot tree therefore is isomorphic to the treap obtained by assigning each key a priority that is equal to its position in the input. Figure 9.5 shows the pivot trees for the executions of quick sort with inputs $[m, \mathbf{q}, s, c, v, u, f, o, n, b, r, a, p, g, h, t, l, u]$ and $[m, s, c, v, u, f, o, n, b, r, a, p, g, h, t, l, u]$.

For the rest of this section, consider the executions X and X' of `QSort` with inputs I and I', where I' is obtained from I by deleting the some key α^*. Let P and P' denote the pivot trees for quick sort.

We start by showing that the pivot trees P and P' are related by a *zip* operation. For a pivot tree P and a node α in P, we define the *left spine* of α as the rightmost path of the subtree rooted at the left child of α. We define the *right spine* of the α as the leftmost path of the subtree rooted at the right child of α. In the example shown in Figure 9.5, the left spine of q contains the vertices o, p and the right spine of q contains the vertices s, r. To *zip* a tree at the vertex α, we replace the subtree rooted at α, with the tree obtained by merging the subtrees P_l and P_r rooted at the left and the right children of α. We merge P_l and P_r by creating a path consisting of the vertices of the left and the right spines of α in their input order and connecting the remaining subtrees to this path. For example, zipping the tree on the left shown in Figure 9.5 at q yields the tree on the right, assuming that the keys are inserted in the following relative order: $[s, o, r, q]$.

Given the pivot tree P, we say that a key α is a *secondary* key if α is on the left or the right spine of α^*. We say that a key α is *affected*, if α is a primary or a secondary key. We say that a key is *unaffected* if it is not affected. For example, in Figure 9.5, the affected keys are m, q, o, p, s, r, the primary keys consists of m and q, and the secondary keys consist of o, p, s, r.

Given an execution of `QSort` and the pivot tree P for the execution, we associate an interval with each key. Let α be a key in P and let Π be the path from α to the root of P. Let α_1 be the least key in p whose right child is on Π or $-\infty$ if no such α_1 exists, and let α_2 be the least key in p whose left child is on Π or $+\infty$, if no such α_2 exists. We define the *execution interval* of α to be the pair (α_1, α_2). As an example, consider the tree on the left in Figure 9.5, the execution interval of p is (o, q), the execution interval of i is (c, m), and the execution interval of a, is $(-\infty, b)$.

The trace of quick sort consists of calls to `QSort` and `Split`. The arguments to `QSort` contribute to the tag the first cell of the input list, the first cell of the sorted accumulator list (rest), and the destination location. The returns contribute to the tag the first cells of the splits lists. For the calls to `Split`, the arguments contribute the first cell of the input, the pivot, the first cell of the accumulator, and the destination. Returns contribute the two cells returned by the recursive call. Since the value read is only passed to a recursive call, it is redundant, and therefore does not contribute to the tag.

We show a lemma that establishes a connection between affected vertices and function calls.

Lemma 46 (Affected Keys and Function Calls)
Let v be a `QSort` call in T and let α be the pivot of that call. Let v' be a `QSort` call in T' with pivot α. Let $w_1 \ldots w_m$ and $w'_1, \ldots, w_r$ be the recursive calls to `Split` starting at v and v' respectively. If α is not affected, then $v \equiv v'$ and $m = r$ and $w_i = w'_i$ for all $1 \leq i \leq m$.

If α is a primary key and $\alpha \neq \alpha^*$, then $r = m + 1$ and all but a expected five of the calls $w_i, 1 \leq i \leq m$ and $w'_i, 1 \leq i \leq r$ have cognates. Furthermore if $\alpha \neq \alpha^*$ and α is not the parent of α^*, then $v = v'$.

Proof:

Let P and P' be the pivot trees for X and X'. Recall that P' is obtained from P by applying the zip operation. The zip operation ensures that 1) all subtrees of P and P' except for those rooted at affected vertices are identical, and 2) the parents of an unaffected key are the same in both P and P'.

We will consider the two cases separately.

α **is not affected:** By the property 1, we know that the input lists to v and v' consists of the same keys. By property 2, we know that the lists are in fact identical because the tags of input cells are determined by the parent and the keys in the input. We conclude that the input lists to v and v' are identical, *i.e.*, consists of equal cells; and therefore the cells returned to them by subcalls to `Split` are also identical.

To show that the destinations and first cells of the accumulators of v and v' are equal, we will show that α has the same execution interval in P and P'. It is a property of the algorithm that the execution intervals determine these two arguments. We will consider two cases. Consider the case where α is not descendant of α^* in P. In this case, the paths from α to the root of P, and from α to the root of P' are identical. Therefore, the execution intervals of α are the same in both P and P'. Suppose now that α is a descendant of α^* in P and let β be the least affected key that is an ancestor of α. We will consider two symmetric cases. Consider the first case, where β is on the left spine. Let (α_1, α_2) be the execution interval of α in P. We know that α_2 is a descendant of β, and α_1 is either a descendant of α, or an ancestor of α^*, or $-\infty$. For example, in Figure 9.5, if $\alpha = n$, then $\alpha_1 = m$ and $\alpha_2 = o$. Consider P' and let (α'_1, α'_2) be the execution interval of α in P'. Recall that P' is obtained from P by zipping P at α^*, and the zip operation merges the left and the right spines, and leaves the subtrees hanging of the spines intact. Since all the keys in the right spine are bigger than the keys in the left spine, when the two spines are merged, their only child on the merged path would be a left child. Therefore $\alpha_1 = \alpha'_1$ and $\alpha'_2 = \alpha_2$. The case for when β is on the right spine is symmetric.

Since all arguments to v and v' are the same, and since the input lists are identical, the recursive calls to `Split` are equal.

α **is a primary key and** $\alpha \neq \alpha^*$**:** Since α is a proper ancestor of α^*, α has the same execution interval with respect to both P and P'. We therefore conclude that the destination and the first cell of the accumulator are the same for both v and v'. Furthermore, we know that v and v' have the same pivot. If α is not the parent of α^*, then α^* is not the first or the second key in the input list, and therefore, the first cell of the inputs to v and v' are the same and thus $v \equiv v'$. Furthermore, the cells returned by `Split` are also the same.

Since the input to v and v' differ only by α^*, and since α^* is the first key in one call to `Split`, and is expected to be the first key in two calls to `Split`, all but five (2+2+1) of the recursive calls to `Split` have cognates.

■

A key property of the algorithm is that the calls to `QSort` and `Split` in T whose pivots include a secondary key or the key α^*, do not have a similar call in T'. This is because the calls in T have α^* in their execution interval and therefore either their destination or the accumulator is determined by α^*. Since α^* is not present in T', none of these calls have a similar function call in T'. By inspecting the pivot trees P and P' it is easy to see that for all other calls, the execution order and the caller-callee relationships remain the same in both T and T'. Since the quick sort algorithm is concise, we conclude that it is monotone for a single insertion/deletion.

To analyze the stability of `QSort`, we will compare the set of function calls performed before and after an insertion/deletion. We perform the analysis in two steps. First, we show that quick sort is expected $O(n)$ stable for the deletion of the first key in the input. For this bound, expectations are taken over all permutations of the input. Second, we show that quick sort is expected $O(\log n)$ stable for a deletion at a position that is uniformly randomly selected. For this bound the expectation is over all permutations of the input list as well as over all possible deletion positions. Note these bounds only depend on the position of the deleted key but not on the key itself.

Lemma 47
Quick sort is $O(n)$-stable for an insertion/deletion to/from the head of the input list with n keys.

Proof:

Since quick sort is monotone, insertions and deletions are symmetric. Therefore we shall consider only deletions. Let I be an input list and let α^* be the fist key of I, *i.e.* $I = \alpha^* :: I'$. Let I' be the list obtained by deleting α^*. Let P and P' denote the pivot trees with I and I' respectively and let T and T' be the traces with I and I' respectively. Note that P' is obtained from P by zipping P at α^*.

By Lemma 46, we know that all calls to `QSort` and `Split` whose pivots are not affected equal. Since quick sort is monotone, all these calls also have cognates and therefore they do not contribute to the trace distance. We therefore need to consider the calls whose pivots are affected.

Since the deleted key α^* is at the root of the tree, there are no primary keys. Consider now the calls associated with secondary keys—the keys that are on the left and the right spine of α^* (in P). The total number of calls to `QSort` along the left and right spine of α^* are expected to be logarithmic in the size of the input because the depth of the Quick sort pivot tree is expected logarithmic (over all permutations of the input). The total number of calls to `Split` associated with a key is the size of the subtree rooted at that key. To bound the number of calls associated with the secondary keys, we show that the total size of the subtrees along the any path in the tree starting at the root is linear in the size of the input.

Consider the sizes of subtrees for the vertices on a path and define the random variables $X_1 \ldots X_k$ such that X_i is the least number of vertices down the spine for which the subtree size becomes $\left(\frac{3}{4}\right)^i n$ or less after it first becomes $\left(\frac{3}{4}\right)^{(i-1)} n$ or less. We have $k \leq \lceil \log_{4/3} n \rceil$ and the sum of the sizes of the subtree along the path is

$$C(n) \quad \leq \quad \sum_{i=1}^{\lceil \log_{4/3} n \rceil} X_i \left(\frac{3}{4}\right)^{i-1} n.$$

Since the probability that a key on the path has rank between $1/4m$ and $3/4m$ where m is the size of the subtree at that key is $\frac{1}{2}$, $E[X_i] \leq 2$ for $i \geq 1$. We therefore have

$$\begin{aligned} E[C(n)] \quad &\leq \quad \sum_{i=1}^{\lceil \log_{4/3} n \rceil} E[X_i] \left(\frac{3}{4}\right)^{i-1} n \\ &\leq \quad \sum_{i=1}^{\lceil \log_{4/3} n \rceil} 2 \left(\frac{3}{4}\right)^{i-1} n. \end{aligned}$$

Thus, $E[C(n)] = O(n)$. This argument applies to the left and the right spine of the α^*, and also the path obtained by merging them. Since each call to `Split` and `QSort` takes constant time excluding the time spent in other calls, each call has constant weight. We conclude that quick sort is $O(n)$ stable for insertions/deletions into/from the head of the input list. ■

Lemma 48
Quick sort is expected

1. $O(\log n)$ *stable for any deletion from the input list, where expectations is taken over all permutations of the input, and*

2. $O(\log n)$ *stable for an insertion at a uniformly randomly chosen position in the input list, where expectations is taken over all permutations of the input as well and the insertion positions, and*

Proof:

Since Quick sort is monotone, insertions and deletions are symmetric. We will therefore consider deletions only. Let I be a list and let α^* be some key in I. Let I' be the list obtained after deleting α^* from I. Let P be the pivot tree for I. Let m be the size of the subtree of P rooted at α^*. Let P' be the pivot tree obtained by zipping P at α^*. We know that the tree P' corresponds to an execution of quick sort with the list I'.

To show the lemma, we will consider the function calls whose pivots are in the subtree of α^*, and the calls whose pivots are not in the subtree separately.

For keys in the subtree of α^*, we will take expectations over all permutations of I for which the subtree of α^* has size m. We will invoke Lemma 47 and show that the distance between the traces of quick sort before and after the deletion of α^* is bounded by $O(m)$. To invoke the lemma, we must take care that the expectations are taken appropriately. Let A be the set of affected keys and consider all permutations of I, for which the subtree of α^* consists only of the keys in A. By Lemma 47, the trace difference for the calls with affected pivots is $O(m)$. Consider now all permutation of I for which the subtree of α^* has exactly m keys, by applying Lemma 47 to each affected set, we know that the trace distance due to affected calls is $O(m)$. Over all permutations of I, the size of the subtree rooted at α^* is $O(\log n)$ [90]. We therefore conclude that the trace distance due to affected calls is $O(\log n)$.

Consider now the calls whose pivots are not in the subtree rooted at α^*. By Lemma 46, we know that all calls whose pivots are the same unaffected key in T and T' are equal. Since quick sort is monotone these calls have cognates and do not contribute to the trace distance. Consider now the calls whose pivots are primary keys. By Lemma 46, these calls add an expected $O(\log n)$ to the trace distance. We therefore conclude that the trace distance is bounded by $O(\log n)$. ■

By combining Lemma 47 and Lemma 48 we have the following stability for quick sort.

Theorem 49 (Stability of Quick Sort)
Quick sort is

1. *expected* $O(n)$ *stable for insertions/deletions at the beginning of the input,*

2. *expected $O(\log n)$ stable insertions/deletions at a uniformly random position in the input.*

To give a tight bound for change propagation with quick sort, we will make a small change to the code to bound the dependence width by a constant. Instead of having `Split` allocate locations based on comparisons, we will have `Split` allocate locations based on the locations along with the `prev` and `next` arguments. This will ensure that locations allocated by the secondary vertices before and during change propagation are disjoint. Based on this fact that and the fact that a single insertion/deletion changes the output of `Split` at a single location by inserting/deleting a key, it is straightforward to show that the quick sort algorithm has constant-width for insertions/deletions. This modification does not change the stability bound. We therefore have the following complexity bound for self-adjusting quick sort.

Theorem 50 (Self-Adjusting Quick Sort)
Quick sort self-adjusts to an

1. *insertion/deletion the beginning of the input in expected $O(n)$ time,*

2. *insertion/deletion at a uniformly random position in expected $O(\log n)$ time.*

All expectations are taken over all permutation of the input as well as internal randomization (but not only the values inserted).

9.4 Graham's Scan

As an example of an incremental algorithm that builds its output by inserting each input one by one, we consider the Graham's Scan algorithm [42] for computing the convex hulls in the plane. For the sake of simplicity, we only consider finding the upper hull. Since the lower hull can be computed using the same algorithm with after negating the outcome of line-side tests, this assumption causes no loss of generality. Figure 9.6 shows the code for the algorithm. The algorithm first sorts the input points from left to right (in order of their x coordinates, with ties broken in by comparing the y coordinates). The algorithm then constructs the hull by incrementally inserting each point to the current hull `h`, which start out empty. The hull is represented as a list of points from ordered right to left.

We assume that points are in general position (no three points are collinear). Since any two points can be connected by a line, the general-position assumption implies that the points are unique. To simplify the terminology and the different cases that arise during the analysis, we will assume that the first point of the input is a special point p_{min}. We define p_{min} to be the leftmost point; and any point is to the right of a line that contains p_{min}. This assumption causes no loss of generality, because comparison functions can be extended to check for the special point with constant time overhead. The true convex hull can be obtained by removing p_{min} from the output in constant time.

9.4.1 Analysis

Since we analyze sorting algorithms separately in the previous two sections, we shall assume, for the analysis, that the input is sorted, and all changes obey the sorted ordering. Consider the class of single insertions/deletions, denoted Δ, to be the pair of inputs sorted from left to right that differ by a single insertion

```
function Insert(p,c) =
  case c of
    nil =>
      return c
  | cons(a, ta) =>
      case (*ta) of
        nil =>
          t ← allocate{p}(2)
          *t ← c
          return cons(p, t)
      | cons(b, tb) =>
          if leftTurn(p, a, b) then
            return Insert (p, cons(b, tb))
          else
            t ← allocate{p}(2)
            *t ← c
            return cons(p, t)

function Scan(cl, ch)
  case cl of
    nil => return ch
  | cons(p, t) =>
      Scan(*t,Insert(p, ch))

function GrahamScan(l) =
  ls ← Sort(l)
  return Scan(ls, nil)
```

Figure 9.6: Code for Graham's Scan.

or deletion. We will show an input sensitive bound on the stability of the algorithm for Δ. Based on this bound, we will show an $O(1)$ expected stability bound, where expectations are taken over all positions of the insertion or deletion.

Given two points a and b, we denote the (infinite) line passing through a and b as $\overrightarrow{ab}$, and the line segment from a to b as ab. For a line l, we define the upper half plane U_l as the set of points that are above (or to the left of) the line l. We say that a point p *sees* some line segment ab, if p is to the left of the line segment ab, *i.e.*, $p \in U_{\overrightarrow{ab}}$. If p sees ab, then we say that ab is visible from p.

We start by stating a known property of convex hulls. The lemma is stated only for insertions but it holds for deletions by symmetry.

Lemma 51
Consider a list of points I and let H be the convex hull of I. Let I' be a list obtained from I inserting the single point p^*. The convex hull H' of I' can be obtained from H be inserting p^* into H in the correct left-to-right order and deleting all line segments of H that are visible from p^*. Any point in H' except for p^* is in the hull H.

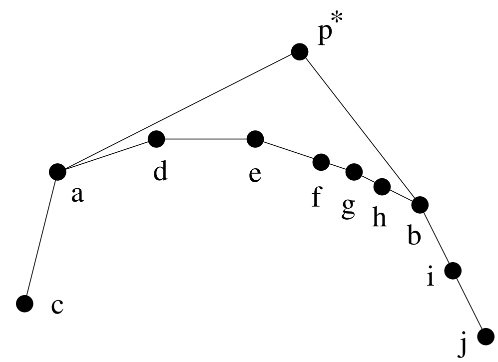


Figure 9.7: Inserting p^* into the hull.

As an example, consider Figure 9.7. The initial hull H is $[c, a, d, e, f, g, h, b, i, j]$ and the hull after inserting p^* is $[c, a, p^*, b, i, j]$. Inserting p^* removes the points d, e, f, g, h and inserts p^*.

All comparisons performed by `GrahamScan` involve a point and a line segment. More specifically, the algorithm compares the point p currently being inserted to all segments on the convex hull from right to left until it finds a segment that is not visible from p. We distinguish between two types of comparisons performed by `GrahamScan`. The first type involves a line segment on the hull and a point that sees that segment. The second type of comparison involves a point p and a line segment ab that is not visible from p. This type of comparison arises when ab is the rightmost line segment in the hull when p is being inserted, but p does not see ab; or when p see the line segment bc to the right of ab but does not see ab. We say that a point p *probes* the line segment ab if it is compared to ab but does not see it. For example in Figure 9.7, the hull consists of the points $[c, a, d, e]$ when p^* is being inserted, and p^* sees the segments ad, de and probes ca.

Consider executions X and X' of Graham's Scan algorithm with some input I and with I' where I' is obtained from I by inserting the point p^*. Let C and C' denote the set of comparisons (line-side tests) performed in X and X'. We bound the symmetric set difference between C and C' based on an input-sensitive cost metric. Given an input and any point p in the input, we define the cost of p, denoted $degree(p)$ as the total number of comparisons (line-side tests) that involves p in a from-scratch execution of `GrahamScan` with I. The value $degree(p)$ is bounded by the total number calls to `Insert` with p plus the number of calls to `Insert` at which p appears as the first point of the hull. We show that `GrahamScan` is $O(degree(p*))$ stable for insertion/deletion of p^*.

Lemma 52
The following properties hold between probing comparisons.

1. *If p probes the line segment de in X' where $d \neq p^*$ and $e \neq p^*$, and $p \neq p^*$, then p probes de in X.*

2. *If p probes the line segment de in X but not in X', then there is some point a to the left of e or $a = e$, such that p probes ap^* or p^*a.*

Proof: Let H_p and H'_p be the convex hulls consisting of the points to the left of p in I and I' respectively. We consider the two properties separately.

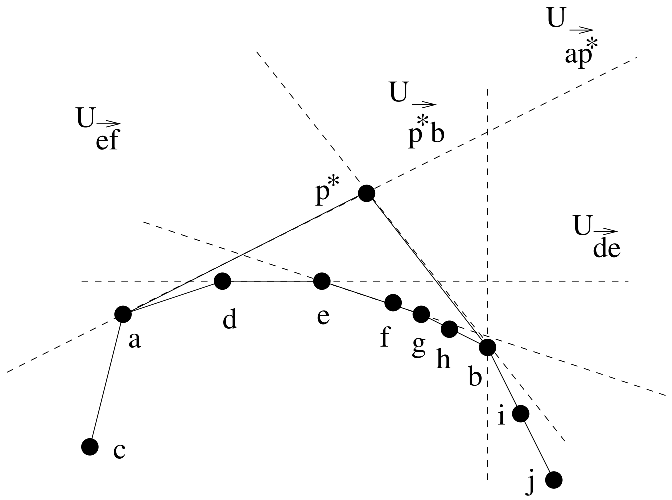


Figure 9.8: The half planes defined by $\overrightarrow{de}$, $\overrightarrow{ap^*}$, $\overrightarrow{ef}$, and $\overrightarrow{p^*b}$.

Consider the first property. If p is to the left of p^*, H_p and H'_p are identical and the property holds trivially. Suppose that p is to the right of p^*. We consider the cases where p may probe de separately. In the first case de is the rightmost segment of the H'_p. By Lemma 51, de is also the rightmost segment of H_p, and therefore p probes de. In the second case, there is some segment ef that is visible to p. We have two cases. In the first case, $f \neq p^*$, and therefore, both de and ef are in H_p and thus p probes de. In the second case, $f = p^*$. Let dk be a line segment in H_p. If dk exists, then we know, by Lemma 51 that p^* sees dk. Since p sees dp^*, we know that p sees dk and therefore p probes cd in X. In all cases, p probes de in X.

Consider the second property. If p is to the left of p^* and p probes de in X', then p will probe de in X because the hulls H_p and H'_p are identical. Suppose now that p is to the right of p^*. If de is the rightmost segment in H_p, then p^* is the rightmost point in H'_p. In this case, p will probe the rightmost segment ap^*, and the property holds. Suppose now that de is not the rightmost segment and let ef be a segment in H_p. In this case, we know, by Lemma 51, that de or ef (or both) are buried by p^*. If ef is not buried, we know that de is buried. In this case, p will probe either dp^* or p^*e, (pick $a = d$ or $a = e$). Suppose now that ef is buried. Let ap^* and p^*b denote the line segments in H'_p with end points p^*. Since p_{min} is the always in the H'_p, a exists. Note that, if p sees ap^*, then p will see de, because the part of the half plane $U_{\overrightarrow{ap^*}}$ to the right of b is contained in the part of the halfplane $U_{\overrightarrow{de}}$ to the right of b (see Figure 9.8). Since p does not see de, p does not see ap^*. We have two cases depending on whether b exists or not.

p^*b **exists.** We know that p sees p^*b, because, as shown in Figure 9.8, the subset of $U_{\overrightarrow{ef}}$ consisting of the points to the right of b is a subset of $U_{\overrightarrow{p^*b}}$. Therefore p sees p^*b and will be compared to ap^*. Since p does not see ap^*, p probes ap^* and therefore the property holds.

p^*b **does not exist.** In this case, only ap^* exists and p^* is the rightmost point to the left of p. Therefore p will be compared to ap^*. Since p does not see ap^*, p probes ap^* and therefore the property holds.

■

Lemma 53
Consider executions X and X' of Graham's Scan algorithm with some input I and with I' where I' is obtained from I by inserting the point p^*. Let C and C' denote the set of comparisons (line-side tests) performed in X and X'. The number of comparisons in the symmetric difference between C and C', $|(C \setminus C') \cup (C' \setminus C)|$ is bounded by $2degree(p^*)$.

Proof: We consider the sets $C \setminus C'$ and $C' \setminus C$ separately and show that the size of each set is bounded by $degree(p^*)$. Let p be a point and let de be a line segment that is involved in a comparison from $C - C'$. Let H_p denote the hull immediately before the insertion of p in X and let H'_p be the hull immediately before the inserting of p in X'. We have two cases depending on whether p sees or probes de.

1. p **sees** de**.** By Lemma 51, de is in H'_p unless it is buried by p^*. Since p does not see de in X', de is buried by p^*. Therefore, de is compared to p^*. We charge the comparison between p and de to the comparison between de and p^*. Since de is deleted from the hull by p, there is no other point p' that sees de in X. Therefore the comparison between p^* and de is charged once by all non-probing comparisons.

2. p **probes** de**.** By Lemma 52, we know that there is some point a such that p probes ap^*. In this case, we charge the comparison between p and de to the comparisons between p and ap^*. Since p performs exactly one probing comparison in both X and X', the comparison between p and ap^* is charged to at most once.

In the first case, we are able to charge one non-probing comparison from $C \setminus C'$ to a non-probing comparison involving p^* such that each non-probing comparison involving p^* is charged at most once. In the second case, we are able to charge one probing comparison from $C \setminus C'$ to a probing comparison involving p^* such that each probing comparison involving p^* is charged at most once. We therefore conclude that $|C \setminus C'| \leq degree(p^*)$.

Let p be a point and let de be a line segment on the hull that is involved in a comparison from $C' \setminus C$. We have two cases depending on whether p sees or probes de. If p sees de in X', then either e or e is p^*—because otherwise de is also in the hull by Lemma 51 and therefore p will see de in X. In the second cases, p probes de. By Lemma 52, we know that either e or e is equal to p^*. Therefore the total number of comparisons in $C' \setminus C$ is bounded by $degree(p^*)$, *i.e.* $|C \setminus C'| \leq degree(p^*)$.

We conclude that $|(C \setminus C') \cup (C' \setminus C)| \leq 2degree(p^*)|$.

■

The trace of the Graham Scan algorithm consists of a call to `GrahamScan` at the root and a chain of calls to `Scan`, one for each point in the input. Each call to `Scan` starts a chain of calls to `Insert`, one for each point removed from the hull. Each call to `Scan` is tagged with the vertex being inserted into the hull, and each call to `Insert` is tagged with the point being inserted and the rightmost point of the hull. Therefore both the execution order and the caller-callee relationships between function calls is determined by the ordering of the points in the sorted input, and the hull. Since inserting/deleting one point into/from the input does not exchange the order of points in the sorted input or in the hull (which is always a sublist of the input points), we have the following theorem.

Theorem 54
The Graham's Scan algorithm is monotone with respect to single insertions/deletions into/from the input.

Since the algorithms is monotone, insertions and deletions are symmetric, and the trace distance can be measured by comparing the sets of function calls.

Theorem 55
The Graham's Scan algorithm for finding planar convex hulls is $O(degree(p^*))$ stable for insertion/deletion of a point p^*.

Proof: Since insertions and deletions are symmetric for monotone programs, we will only consider insertions. Consider inserting the point p^* to the input I to obtain the input I'. Let C and C' denote the set of line-side test performed by an execution of `GrahamScan` with I and I'.

Since `GrahamScan` is called exactly once in each execution, it contributes zero to the trace distance. Since `Scan` is called exactly once for each input point it contributes a constant to the distance. For calls to `Insert`, note that each insert will be tagged with the points that are involved in the line-side test being performed. Since all memory allocated based on the points, the quantity $|(C \setminus C') \cup (C' \setminus C)|$ bounds the number of calls to `Insert` with no cognates. Since $|(C \setminus C') \cup (C' \setminus C)| \leq 2degree(p^*)$, we conclude that the algorithm is $O(degree(P^*)$ stable for insertions/deletions. ■

This together with Theorem 34 gives us the following bound on the change-propagation time for Graham Scan.

Theorem 56
Graham Scan algorithm updates its output in $O(degree(p^*) \log degree(p^*))$ time after the insertion/deletion of the point p^*.

Since each call to `Insert` either inserts a point to the current hull or deletes a point from the current hull, the total number of call to `Insert` is $2n$. Since line-side tests are performed only by `Insert`, the total number of line-side tests is bounded by $2n$. Since each line-side test involves three points, the sum of the $degree(\cdot)$'s for all points is no more than $6n$. The graham-scan algorithm is therefore expected constant stable for the deletion of a uniformly chosen point from the input. Same bound applies to an insertion when taking expectations over all points in the input. Since the degree of a point is bounded by n, we have the following theorem.

Theorem 57 (Self-Adjusting Graham's Scan)
The Graham's Scan algorithm is expected $O(1)$-stable for a uniformly random deletion/insertion into a list with n points, and handles an insertion/deletion in expected $O(\log n)$ time.

Recall that self-adjusting sorting requires expected $O(\log n)$ time. Since a single insertion/deletions changes the output of the sort by one key, the two algorithms when combined yield an expected $O(\log n)$-time self-adjusting convex-hull algorithm.

Chapter 10

Tree Contraction and Dynamic Trees

This chapter shows that the randomized tree contraction algorithm of Miller and Reif is expected $O(\log n)$ stable [62]. The self-adjusting tree-contraction solves a generalization of the Sleator and Tarjan's dynamic-trees problem in expected $O(\log n)$ time [91, 92]. Section 10.1 describes tree contraction and how it can be implemented. Section 10.3 proves the expected $O(\log n)$ stability bound for tree contraction for edge insertions/deletions.

10.1 Tree Contraction

The tree-contraction algorithm of Miller and Reif takes a t-ary forest, F, and contracts each tree in the forest to a single vertex in a number of rounds. In each round, each tree is contracted by applying the rake and compress operations. The *rake* operations delete all leaves of the tree (if the tree is a single edge, then only one leaf is deleted). The *compress* operations delete an independent set of degree-two vertices. By an independent set, we mean a set of vertices none of which are adjacent to each other. All rake and compress operations within a round are local and are applied "in parallel"—all decisions are based on the state when the round starts. The algorithm terminates when no edges remain.

Various versions of tree contraction have been proposed depending on how they select the independent set of degree-two vertices to compress. There are two basic approaches, one deterministic and the other randomized. We use a randomized version. In *randomized tree contraction*, each vertex flips a coin in each round and a degree-two vertex is compressed if it flips a head, its two neighbors both flip tails, and neither neighbor is a leaf. This compress rule is a slight generalization of the original rule by Miller and Reif [62], which only applies to rooted trees.

Using tree contraction the programmer can perform various computations on trees by associating data with edges and vertices and defining how data is accumulated during rake and compress [62, 64]. In Chapter 17, we consider a broad range of applications that are considered in the context of the dynamic-trees problem [91, 38, 96, 46, 11, 5, 95], and show how they can be computed by using rake and compress operations. Since the rake and compress operations are entirely determined by the structure of the tree and not the data we assume, for this section, that trees come with no associated data. Due to the orthogonality between rake/compress operations and the data operations, all results hold in the presence of data.

```
Tree_Contract_MR (V_s, E_s)
   while E_s ≠ ∅ do
      (V_t, E_t) ← CreateForest(V_s, E_s)
      for each v ∈ V_s do
         Contract(v, (V_s, E_s), (V_t, E_t))
         V_s ← V_t
         E_s ← E_t

Contract(v, (V_s, E_s), (V_t, E_t))
   if degree(v) = 1 then
      // rake
      (u, v) ∈ E_s
      if degree(u) > 1 or u > v then
         DeleteNode(v, (V_s, E_s), (V_t, E_t))
      else
         CopyNode(v, (V_s, E_s), (V_t, E_t))
   else if degree(v) = 2 then
      // compress
      (u, v) ∈ E ∧ (w, v) ∈ E
      if degree(u) > 1 and degree(w) > 1 and flips(u, v, w) = (T, H, T) then
         DeleteNode(v, (V_s, E_s), (V_t, E_t))
         u_t = next(u)
         w_t = next(w)
         E_t ← E_t ∪ {(u_t, w_t)}
      else
         CopyNode(v, (V_s, E_s), (V_t, E_t))
   else
      CopyNode(v, (V_s, E_s), (V_t, E_t))

CreateForest((V_s, E_s))
   (V_t, E_t) ← (∅, ∅)
   for ∀v ∈ V_s do
      u ← new node
      V_t ← V_t ∪ {u}
      next(v) ← u

CopyNode(v, (V_s, E_s), (V_t, E_t))
   v_t ← next(v)
   for ∀(u, v) ∈ E_s do
      u_t ← next(u)
      E_t ← E_t ∪ {(u_t, v_t)}

DeleteNode(v, (V_s, E_s), (V_t, E_t))
   v_t ← next(v)
   V_t ← V_t \ {v_t}
   for ∀(u_t, v_t) ∈ E_t do
      E_t ← E_t \ {(u_t, v_t)}
```

Figure 10.1: Randomized tree-contraction algorithm of Miller and Reif.

Figure 10.1 shows the pseudo-code for a sequential version of the randomized tree contraction algorithm of Miller Reif. The algorithm takes a source forest (V_s, E_s) and contracts the forest into a forest consisting of disconnected vertices in a number of round. In each round, the algorithm creates an empty target forest (V_t, E_t) by copying each vertex in the source forest. The $next$ function maps each vertex to its copy in the next round. Initially the target forest contains no edges. The algorithm then contracts each vertex by calling `Contract`. The `Contract` function processes the vertex v based on its degree. If v has degree one, then it may be raked, if it has degree two, then it may be compressed. When a vertex is not raked or compressed, it is copied to the target forest by calling `CopyNode` function. This function inserts the edges between the vertex and all its neighbors in the target forest. During a rake or compress, a vertex is deleted by `DeleteNode` function. This function deletes all the edges incident on the vertex from the target forest.

$degree(v) = 1$**:** The algorithm checks if the neighbor u of v has degree one. If not, then v is raked from the target forest. If the u has degree one, then v is raked if it is greater than u (we assume an arbitrary total order on the vertices). If v is not raked, then it is copied to the next round.

$degree(v) = 2$**:** The algorithm checks the degrees of the neighbors, u and w, of v. If u and w both have degree two or more, and the they both flips tails and v flips heads, then the algorithm compresses v by deleting it from the target forest and inserting an edge between u and w. If v is not compressed, then it is copied to the next round.

$degree(v) > 2$**:** In this case, v is copied to the next round.

Miller and Reif originally studied tree contraction in the context of parallel computation for directed trees. The code presented here is a sequential version of their algorithm extended to support undirected trees. To make sure that every location is written at most once, the pseudo-code considered here makes a copy of the forest at each round. All decisions on the rake and compress operations at some round are performed "in parallel" based on the forest at the beginning of that round.

Miller and Reif showed that their algorithm for rooted trees requires an expected logarithmic number of rounds and linear work. The key theorem behind these bounds shows that a constant fraction of all vertices is deleted in each round with high probability [63, 62]. These theorems are easily extended to the slight generalization of the compress rule presented here by only modifying the constant factors.

10.2 Self-Adjusting Tree Contraction

This section describes an implementation of the tree-contraction algorithm in the closure model, by specifying data structures for the pseudo code presented in Figure 10.1. The main data structure is an adjacency-list representation for forests, where a forest is represented as a list of vertices. Each vertex maintains its *neighbors*, an array of pointers to the adjacent vertices, its *degree*, a pointer *next* to its copy in the next round, a *deleted* flag, a unique *identity* (an integer), and a round number. Each element of the neighbor array of v consists of a pointer to a neighbor u and an integer, called *backIndex* equal to the index (position) of the of the neighbor pointer from u to v.

Figure 10.2 shows the tree-contraction algorithm on the closure model. The algorithm specializes the algorithm given in Figure 10.1 with the adjacency-list representation. The algorithm repeatedly applies the

```
function Tree_Contract (c_s)
   if HasEdges(c_s, false) then
      c_t ← CreateForest(c_t)
      ContractAll(c_s)
      Tree_Contract(c_t)
   else return c_s

function ContractAll (c)
   case a of
      nil => return
   |  cons(v, t) => Contract(v); ContractAll(*t)

function Contract(v)
   if degree(v) = 1 then // rake
      [(u, _)] ← GetNeighbors(v)
      if degree(u) > 1 or id(u) > id(v) then deleted(next(v)) ← true
      else CopyVertex(v, next(v), 0)
   else if degree(v) = 2 then // compress
      [(u, i_uv), (w, i_wv)] ← GetNeighbors(v)
      if degree(u) > 1 and degree(w) > 1 and flips(u, v, w) = (T, H, T) then
         deleted(next(v)) ← true
         neighbors(u)[i_uv] ← (w, i_wv), neighbors(w)[i_wv] ← (u, i_uv)
      else CopyVertex(v, next(v), 0)
   else CopyVertex(v, next(v), 0)

function HasEdges(c, flag) =
   case c of
      nil => return flag
   |  cons(v, t) => if degree(v) > 0 then HasEdges(*t, true)
                    else HasEdges(*t, flag)

function CreateForest(c) =
   case c of
      nil => return nil
   |  cons(v, t) =>
         cc ← CreateForest(*t)
         if deleted(v) then return cc
         else  l ← alloc[id(v), round(v)](1), *l ← cc
               u ← EmptyVertex(id(v))
               round(u) ← round(v) + 1
               next(v) ← u
               return cons(u, l)

function CopyVertex(v, v_t, i) =
   if (i < max_degree) then
      CopyVertex (v, v_t, i + 1)
      if neighbor(v)[i] ≠ null then
         (pu, i_uv) ← neighbor(v)[i]
         u_t ← next(*pu)
         neighbor(u_t)[i_uv] ← (v_t, i)
```

Figure 10.2: Randomized tree-contraction in the closure model.

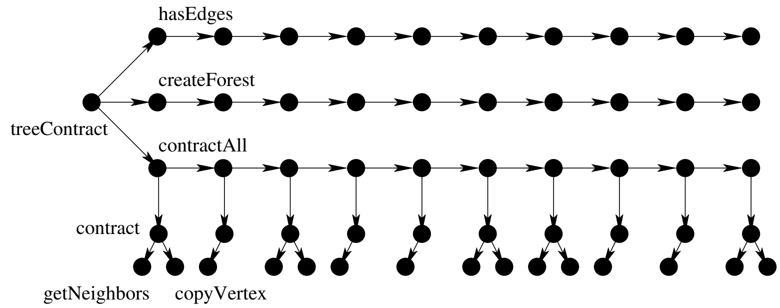


Figure 10.3: An example trace of tree-contraction at some round.

function `Contract` to each vertex in rounds until the forest reduces to a disconnected set of vertices. The function `HasEdges` determines if there are any edges in the forest.

Each round creates an empty target forest of disconnected vertices and applies `Contract` to each vertex in the source. The `Contract` function first checks the degree of the vertex. If the vertex can be raked or compressed, then the `deleted` flag of its copy is set to $true$. If the vertex is to be compressed, then an edge is inserted between the neighbors of the compressed vertex. If the vertex is not raked or compressed, then it is copied to the target forest by calling the `CopyVertex` function. The `CopyVertex` function copies a vertex v by walking over the neighbors of v and making the copies of each neighbor point to the copy of v. For the purposes of brevity, the code does not show how the degree is computed. The degree is computed by the `CopyVertex` function while copying the neighbors. For each neighbor, the function checks if it will be raked in the next round—this is determined by checking the degree of the vertex and its id—and subtract one from the current degree for each raked neighbor.

To ensure that each location is written at most once, the code relies on a liveness flag. When a vertex is raked or compressed, its copy in the target forest is marked deleted by setting its $deleted$ flag to $true$, instead of removing the vertex from the target. Removing the vertex from the target would have required writing tail element of the preceding vertex for a second time. Deleted vertices are removed from the forest by not copying them when creating the target forest.

To generate random coin flips, we use a family H of 3-wise independent hash functions mapping $\{1 \ldots n\}$ to $\{0,1\}$. We randomly select one member of H per round. Since there are families H with $|H|$ polynomial in n [97], we need $O(\log^2 n)$ random bits in expectation. As discussed in Section 8.1 we analyze trace stability assuming fixed selection of random bits and take expectations over all selections of bits. For fixed random bits, vertex i will generate the same coin flip on round j for any input. This ensures that the trace remains stable in different executions.

10.3 Trace Stability

This section shows that the tree-contraction algorithm is expected $O(\log n)$ stable for single edge insertions/deletions.

We partition the trace into rounds. Each round consists of a sequence of calls to `hasEdges` function followed by a possibly empty sequence of calls to `createForest` and `contractAll` functions. Each call to `contractAll` performs a single call to `contract`. Each call to `contract` performs at most one call to `getNeighbors` and at most one call to `copyVertex`. Figure 10.3 shows an hypothetical traces of tree contraction at some round where the forest contains ten vertices. In the figure, the vertices representing the calls to `copyVertex` represent a constant number of recursive calls to this function—the number of calls is determined by the maximum degree that each vertex may have.

The tree-contraction algorithm allocates all memory locations using unique labels based on the identity of the vertex being considered and its round number. Since each function call takes as argument the vertex or a list cell that contains a vertex, and since each function call is called on a vertex in each round at most once, the algorithm is concise. To see that the algorithm is monotone, note that the list of vertices input to some round is a sublist of the vertices in the previous round. Furthermore, a single insertion or deletion of an edge does not exchange the order of vertices in the original inputs lists. It is straightforward to show that the algorithm is monotone for single insertions and deletions based on these two facts.

Theorem 58
The tree-contraction algorithm is monotone for the class of single edge insertions/deletions.

Consider a forest $F = (V, E)$ with n vertices and execute tree-contraction on F. We identify vertices by their identities—two vertices are considered the same if they have the same identity. We denote the contracted forest at round i as $F^i = (V^i, E^i)$. Throughout this section, the term *"at round i"* means, *"at the beginning of round i"* We say that a vertex v is *live* at round i, if $v \in V^i$. A vertex is live at round i, if it has not been deleted (compressed or raked) in some previous round. We define the *configuration* of a vertex v at round $i \geq 1$

$$\kappa_F^i(v) = \begin{cases} \{(u, \sigma(u)) \mid (v, u) \in E^i\} & v \in V^i \\ \textsf{deleted} & v \notin V^i \end{cases}$$

Here $\sigma(u)$ is the singleton status of vertex u. The singleton status of a vertex at some round is *true* if the vertex has degree one at that round and *false* otherwise.

The motivation behind defining the configuration of a vertex is to measure the distance between two trace by comparing the configurations of vertices. The configuration of a vertex v is defined to be a superset of the tags of all function calls where v is passed as an argument or is stored in the first cell of the adjacency list. Since the tree-contraction is monotone and every function call in the trace has constant weight (takes constant time to execute), to measure the trace distance, it suffices to count the number of traces whose configurations differ in two contractions.

Consider some forest $G = (V, E \setminus \{(u, v)\})$ obtained from F by deleting the edge (u, v) and let $\mathcal{X}_F$ and $\mathcal{X}_G$ be the *contraction* with F and G respectively. By a *contraction*, we mean an execution of the tree-contraction algorithm on some forest. Let T_F and T_G be traces for the contractions $\mathcal{X}_F$ and $\mathcal{X}_G$. Define the set of vertices V of F and G as $V = \{v_1, \ldots, v_n\}$. The distance between T_F and T_G is within a constant factor of the difference between $\mathcal{X}_F$ and $\mathcal{X}_G$, that is $\delta(T_F, T_G) = O\left(\sum_{i=1}^{k \log n} \sum_{j=1}^{n} \textsf{neq}(\kappa_F^i(v_j), \kappa_G^i(v_j))\right)$, where $\textsf{neq}(\cdot, \cdot)$ is one if the configurations are not equal and zero otherwise. To bound the trace distance it, therefore, suffices to count the places where two configurations do not match in $\mathcal{X}_F$ and $\mathcal{X}_G$.

We say that a vertex v *becomes affected in round* i, if i is the earliest round for which $\kappa_F^{i+1}(v) \neq \kappa_G^{i+1}(v)$. Once a vertex becomes affected, it remains *affected* in any subsequent round. Only input changes will make a vertex affected at round one. We say that a vertex v is a *frontier* at round i, if v is affected and is adjacent to a unaffected vertex at round i. We denote the set of affected vertices at round i as A^i—note that A^i includes all affected vertices live or deleted. For any $A \subseteq A^i$, we denote the forests induced by A on F^i and G^i as F^i_A and G^i_A respectively. Since all deleted vertices have the same configuration $\delta(T_F, T_G) = O\left(\sum_{i=1}^{k \log n} |F^i_{A^i}| + |G^i_{A^i}|\right)$.

An *affected component* at round i, $\mathcal{AC}^i$ is defined as a maximal set satisfying (1) $\mathcal{AC}^i \subseteq A^i$, and (2) $F^i_{\mathcal{AC}^i}$ and $G^i_{\mathcal{AC}^i}$ are trees.

To prove that tree contraction is expected $O(\log n)$ stable we will show that the expected size of $F^i_{A^i}$ and $G^i_{A^i}$ is constant at any round i. The first lemma establishes two useful properties that we use throughout the analysis.

Lemma 59
(1) A frontier is live and is adjacent to the same set of unaffected vertices at any round i in both $\mathcal{X}_F$ and $\mathcal{X}_G$. (2) If a vertex becomes affected in any round i, then it is either adjacent to a frontier or it has neighbor that is adjacent to a frontier at that round.

Proof: The first property follows from the facts that (A) a frontier is adjacent to a unaffected vertex at that round, and (B) unaffected vertices have the same configuration in both $\mathcal{X}_F$ and $\mathcal{X}_G$. For the second property, consider some vertex v that is unaffected at round i. If v's neighbors are all unaffected, then v's neighbors at round $i+1$ will be the same in both $\mathcal{X}_F$ and $\mathcal{X}_G$. Thus, if v becomes affected in some round, then either it is adjacent to an affected vertex u, or v has neighbor u whose singleton status differs in $\mathcal{X}_F$ and $\mathcal{X}_G$ in round $(i+1)$, which can happen only if u is adjacent to an affected vertex. ■

We now partition A^i into two affected components $\mathcal{AC}^i_u$ and $\mathcal{AC}^i_v$, $A^i = \mathcal{AC}^i_u \cup \mathcal{AC}^i_v$, such that $G^i_{A^i} = G^i_{\mathcal{AC}^i_u} \cup G^i_{\mathcal{AC}^i_v}$ and $F^i_{A^i} = F^i_{\mathcal{AC}^i_u \cup \mathcal{AC}^i_v}$. Affected components correspond to the end-points of the deleted edge (u, v). Note $G^i_{\mathcal{AC}^i_u}$ and $G^i_{\mathcal{AC}^i_v}$ are disconnected.

Lemma 60
At round one, each of $\mathcal{AC}^1_u$ and $\mathcal{AC}^1_v$ contain at most two vertices and at most one frontier.

Proof: Deletion of (u, v) makes u and v affected. If u does not become a leaf, then its neighbors remain unaffected. If u becomes a leaf, then its neighbor u' (if it exists) becomes affected. Since u and v are not connected in G, the set $\{u\}$ or if u' is also affected $\{u, u'\}$ is an affected component, this set will be $\mathcal{AC}^1_u$. Either u or if u' exists, then u' may be a frontier. The set $\mathcal{AC}^1_v$ is built by a similar argument. ■

Lemma 61
Assume that $\mathcal{AC}^i_u$ and $\mathcal{AC}^i_v$ are the only affected components in round i. There are exactly two affected components in round $i+1$, $\mathcal{AC}^{i+1}_u$ and $\mathcal{AC}^{i+1}_v$.

Proof: By Lemma 59, we consider vertices that become affected due to each frontier. Let U and V be the set of vertices that become affected due to a frontier in $\mathcal{AC}^i_u$ and $\mathcal{AC}^i_v$ respectively and define $\mathcal{AC}^{i+1}_u = \mathcal{AC}^i_u \cup U$ and $\mathcal{AC}^{i+1}_v = \mathcal{AC}^i_v \cup V$. This accounts for all affected vertices $A^{i+1} = A^i \cup U \cup V$. By Lemma 59 the vertices of U and V are connected to some frontier by a path. Since tree-contraction preserves connectivity, $F^{i+1}_{\mathcal{AC}^{i+1}_u}, F^{i+1}_{\mathcal{AC}^{i+1}_v}$ and $G^{i+1}_{\mathcal{AC}^{i+1}_u}, G^{i+1}_{\mathcal{AC}^{i+1}_v}$ are trees, and $G^{i+1}_{\mathcal{AC}^{i+1}_u}, G^{i+1}_{\mathcal{AC}^{i+1}_v}$ remain disconnected. ■

By induction on i, using Lemmas 60 and 61, the number of affected components is exactly two. We now show that each affected component has at most two frontiers. The argument applies to both components, thus we consider some affected component $\mathcal{AC}$.

Lemma 62
Suppose there is exactly one frontier in $\mathcal{AC}^i$. At most two vertices become affected in round i due to contraction of vertices in $\mathcal{AC}^i$ and there are at most two frontiers in $\mathcal{AC}^{i+1}$.

Proof: Let x be the sole frontier at round i. Since x is adjacent to a unaffected vertex, it has degree at least one. Furthermore x cannot have degree one, because otherwise its leaf status would be different in two contraction making its neighbor affected. Thus x has degree two or greater.

Suppose that x is compressed in round i in $\mathcal{X}_F$ or $\mathcal{X}_G$. Then x has at most two unaffected neighbors y and z, which may become affected. Since x is compressed, y will have degree at least one after the compress, and thus no (unaffected) neighbor of y will become affected. Same argument holds for z. Now suppose that x is not deleted in either $\mathcal{X}_F$ or $\mathcal{X}_G$. If x does not become a leaf, then no vertices become affected. If x becomes a leaf, then it has at most one unaffected neighbor, which may become affected. Thus, at most two vertices become affected or frontier. ■

Lemma 63
Suppose there are exactly two frontiers in $\mathcal{AC}^i$. At most two vertices become affected in round i due to the contraction of vertices in $\mathcal{AC}^i$ and there are at most two frontiers in $\mathcal{AC}^{i+1}$.

Proof: Let x and y be two frontiers. Since x and y are in the same affected component, they are connected by a path of affected vertices and each is also adjacent to a unaffected vertex. Thus each has degree at least two in both contractions at round i. Assume that x is compressed in either contraction. Then x has at most one unaffected neighbor w, which may become affected. Since w has degree at least one after the compress, no (unaffected) neighbor of w will become affected. If x is not deleted in either contraction, then no vertices will become affected, because x cannot become a leaf—a path to y remains, because y cannot be raked. Therefore, at most one vertex will become affected and will possibly become a frontier. The same argument applies to y. ■

Lemma 64
The total number of affected vertices in F^i and G^i is $O(1)$ in the expected case.

Proof: Consider applying tree contraction on F^i and note that $\mathcal{F}^i_{A^i}$ will contract by a constant factor when we disregard the frontier vertices, by an argument similar to that of Miller and Reif [62]—based on independence of randomness between different rounds. The number of affected components is exactly two and each component has at most two frontiers and cause two vertices become affected (by Lemmas 60, 62, and 63). Thus, there are at most four frontiers and four new vertices may become affected in round i. Thus, we have $E[|F^{i+1}_{A^{i+1}}|] \leq (1-c)|\mathcal{F}^i_{A^i}| + 8$ and $E[|F^{i+1}_{A^{i+1}}|] \leq (1-c)E[|\mathcal{F}^i_{A^i}|] + 8$.

Since the number of affected vertices in the first round is at most 4, $E[|F_A^i|] = O(1)$, for any i. A similar argument holds for G^i. ■

For our main result, we rely on the following theorem. This theorem is a simple extension of Theorem 3.5 from Miller and Reif's original work [63].

Theorem 65
There exists come constants a, b and n_0, such that for all $n \geq n_0$, randomized tree contraction reduces a forest with n vertices into a single vertex in $a \log n + b$ rounds with probability of failure at most $1/n$.

Theorem 66
Tree contraction is expected $O(\log n)$ stable for a single edge insertion or deletion.

Proof: By Theorem 65, we know that tree contraction takes $a \log n + b$ rounds for sufficiently large n with probability of failure no more than $1/n$. Let D be the random variable denoting the number of affected vertices in a contraction and let R be the random variable denoting the number of rounds. We will bound the expectation of D by conditioning it on the number of rounds. In particular, we know that

$$E[D] = \sum_{r=0}^{\infty} E[D|R = r] \times \Pr[R = r]$$

Note now that that $E[D|R = r] = r$, because, by Lemma 64, the expected number of live affected vertices per round is constant independent of the number of rounds. We therefore have

$$E[D] = \sum_{r=0}^{\infty} r \times \Pr[R = r].$$

Let $L = a \log n + b$; we have

$$E[D] = \sum_{i=0}^{\infty} \sum_{r=L^i}^{L^{i+1}} r \times \Pr[R = r] \leq \sum_{i=0}^{\infty} L^{i+1} \sum_{r=L^i}^{L^{i+1}} \Pr[R = r] \leq \sum_{i=0}^{\infty} L^{i+1} (\frac{1}{n})^i \leq L \times \sum_{i=0}^{\infty} (\frac{L}{n})^i.$$

Since $\sum_{i=0}^{\infty} (\frac{L}{n})^i = O(1)$, $E[D] = O(L)$. We therefore conclude that the expected number of affected vertices is $O(\log n)$. ■

It is a property of the tree-contraction algorithm that the order in which the vertices of the forest are contracted within each round does not affect the result. Therefore a FIFO queue can be used during change propagation instead of a general-purpose priority queue. We therefore have the following theorem for dynamic trees problem of Sleator and Tarjan [91].

Theorem 67 (Dynamic Trees)
Self=adjusting tree contraction algorithm adjusts to a single edge insertion/deletion in expected $O(\log n)$ time. Self-adjusting tree contraction thus solves the dynamic trees problem in expected $O(\log n)$ time.

Part IV

Language Techniques

Introduction

This part of the thesis presents language facilities for writing self-adjusting programs. We consider three different purely functional languages, called the Adaptive Functional Language (AFL), the Memoizing Functional Language (MFL), the Self-Adjusting functional Language (SLf, read "self"), and an imperative language where memory locations can be written multiple times. The AFL (Chapter 11) and MFL (Chapter 12) languages are based on dynamic dependence graphs, and memoization. The SLf language combines and extends AFL and MFL to provide support for non-strict dependences. We also present an imperative language, called SLi, that extends SLf language with support for multiple writes (side-effects).

Chapter 11 introduces the AFL (Adaptive Functional Language) language. The AFL language enables the programmer to transform an ordinary, non-self-adjusting program into a self-adjusting program by making simple, methodical changes to the code. As an AFL program runs, a run-time system builds the dynamic dependence graph of the execution. The DDG is then used to adjust the computation to external changes. A key property of the AFL language is that it supports *selective dependence tracking*. The programmer decides what parts of the input will be "changeable", and the language ensures that all dependences pertaining to the changeable data is tracked. The dependences pertaining to the "stable" parts of the input, *i.e.*, the parts that cannot change, are not tracked.

We present a static and dynamic semantics for the AFL language and prove that the language is type safe. We also formalize the change-propagation algorithm for dynamic dependence graphs and prove that it is correct. The correctness theorem shows that change propagation is identical to a from-scratch execution—except, of course, it is faster. We also describe an implementation of the language as an ML library and give the code for this library. The implementation is based on the data structures described in Chapter 6.

Chapter 12 describes a language for *selective memoization*. Selective memoization enables the programmer to apply memoization efficiently by providing control over 1) the cost of equality test, 2) the input-output dependences, 3) space consumption. The most important aspect of selective memoization is the mechanisms for supporting memoization with precise input-output dependences. We study selective memoization in the context of a purely functional language, called MFL, and present a static and dynamic semantics for the language. Based on the semantics, we prove the correctness of the selective memoization techniques. We also describe an efficient implementation of the MFL language as an SML library and presents the code for the implementation.

Chapter 13 describes a language, called SLf, based on memoized dynamic dependence graphs and the memoized change-propagation algorithm. The SLf language combines the AFL and the MFL languages and extends them with constructs to distinguish between *strict* and *non-strict dependences*. The language enables the programmer to transform an ordinary, non-self adjusting program into a self-adjusting program by making small methodical changes to the code. The key properties of the SLf language are that 1) it yields efficient self-adjusting programs, and 2) it enables determining the time complexity of change-propagation by using trace-stability techniques. In Part V we present an experimental evaluation of the SLf language by considering a number of applications.

Chapter 14 presents an imperative language for self-adjusting computation where memory locations can be written multiple times. The key idea is to make computations persistent by keeping track of all writes to memory by versioning [31]. This chapter shows that our techniques are also applicable in the imperative setting.

Chapter 11

Adaptive Functional Programming

This chapter describes language techniques for writing self-adjusting programs by applying dynamic dependence graphs selectively. Since the techniques described in this chapter rely only on ordinary (non-memoized) dynamic dependence graphs, we use the term *adaptive* when referring to programs based on these techniques. We reserve the word *self-adjusting* to refer to programs based on memoized dependence graphs.

We describe a language, called AFL (Adaptive Functional Language), where every program is adaptive. We give the static semantics (type system), and the dynamic semantics of the AFL language and prove that the language is safe. Based on the static and dynamic semantics of AFL, we formalize the change-propagation algorithm for dynamic dependence graphs and prove that it is correct. We also presents the complete code for an implementation of the language as an ML library.

Using the ML library, Section 11.2 illustrates the main ideas of adaptive functional programming. We describe first how to transform an ordinary quick sort written in the ML language into an adaptive quick sort by applying a methodical transformation. We then describe the particular version of the dynamic dependence graphs and the change-propagation algorithm used by the library. We finish by presenting the code for our library.

Section 11.3 defines an adaptive functional programming language, called AFL, as an extension of a simple call-by-value functional language with adaptivity primitives. The static semantics of AFL show that AFL is consistent with purely functional programming. Since the language involves a particular form of references, yet it is purely functional, the type safety proof for the language is intricate. The dynamic semantics of AFL is given by an evaluation relation that maintains a record of the adaptive aspects of the computation, called a trace. The trace is used by the change propagation algorithm to adapt the computation to external changes.

Section 11.5 gives a semantics to the change propagation algorithm based on the dynamic semantics of AFL. The change propagation algorithm interprets a trace to determine the expressions that need to be re-evaluated and to determine the order in which the re-evaluation takes place. We prove that the change-propagation algorithm is correct by showing that it yields essentially the same result as a complete re-execution on the changed inputs.

11.1 Introduction

The proposed mechanism extends call-by-value functional languages with a small set of primitives to support adaptive functional programming. Apart from requiring that the host language be purely functional, we make no other restriction on its expressive power. In particular our mechanism is compatible with the full range of effect-free constructs found in ML. The proposed mechanism has these strengths:

- **Generality:** It applies to any purely functional program. The programmer can build adaptivity into an application in a natural and modular way.

- **Flexibility:** It enables the programmer to control the amount of adaptivity. For example, a programmer can choose to make only one portion or aspect of a system adaptive, leaving the others to be implemented conventionally.

- **Simplicity:** It requires small changes to existing code. For example, the adaptive version of Quicksort presented in the next section requires only minor changes to the standard implementation.

- **Efficiency:** The mechanism admits a simple implementation with constant-time overhead. The computational complexity of an adaptive program can be determined using standard analytical techniques

The adaptivity mechanism is based on the idea of a *modifiable reference* (or *modifiable*, for short) and three operations for creating (`mod`), reading (`read`), and writing (`write`) modifiables. A modifiable allows the system to record the dependence of one computation on the value of another. A modifiable reference is essentially a write-once reference cell that records the value of an expression whose value may change as a (direct or indirect) result of changes to the inputs. Any expression whose value can change must store its value in a modifiable reference; such an expression is said to be *changeable*. Expressions that are not changeable are said to be *stable*; stable expressions are not associated with modifiables.

Any expression that depends on the value of a changeable expression must express this dependence by explicitly reading the contents of the modifiable storing the value of that changeable expression. This establishes a data dependence between the expression reading that modifiable, called the *reader*, and the expression that determines the value of that modifiable, the *writer*. Since the value of the modifiable may change as a result of changes to the input, the reader must itself be deemed a changeable expression. This means that a reader cannot be considered stable, but may only appear as part of a changeable expression whose value is stored in some other modifiable.

By choosing the extent to which modifiables are used in a program, the programmer can control the extent to which it is able to adapt to change. For example, a programmer may wish to make a list manipulation program adaptive to insertions into and deletions from the list, but not under changes to the individual elements of the list. This can be represented in our framework by making only the "tail" elements of a list adaptive, leaving the "head" elements stable. However, once certain aspects are made changeable, all parts of the program that depend on those aspects are, by implication, also changeable.

The key to adapting the output to change of input is to record the dependencies between readers and writers that arise during the initial evaluation. These dependencies are maintained by using a version of the *dynamic-dependence-graph (DDG)* data structure presented in Chapter 4. The key difference between this version and the standard DDGs is that, this version only tracks modifiable references and reads, instead of

```
signature ADAPTIVE =
  sig
    type 'a mod
    type 'a dest
    type changeable

    val mod: ('a * 'a -> bool) -> ('a dest -> changeable) -> 'a mod
    val read: 'a mod * ('a -> changeable) -> changeable
    val write: 'a dest * 'a -> changeable

    val init: unit -> unit
    val change: 'a mod * 'a -> unit
    val propagate: unit -> unit
  end
```

Figure 11.1: Signature of the adaptive library.

all memory locations and all function calls. This is how AFL enables applying dynamic-dependence-graph-based dependence tracking selectively.

11.2 A Framework for Adaptive Computing

We give an overview of our techniques based on our ML library and an adaptive version of Quicksort.

11.2.1 The ML library

The signature of our adaptive library for ML is given in Figure 11.1. The library provides functions to create (`mod`), to read from (`read`), and to write to (`write`) modifiables, as well as meta-functions to initialize the library (`init`), change input values (`change`) and propagate changes to the output (`propagate`). The meta-functions are described later in this section. The library distinguishes between two "handles" to each modifiable: a *source* of type `'a mod` for reading from, and a *destination* of type `'a dest` for writing to. When a modifiable is created, correct usage of the library requires that it only be accessed as a destination until it is written, and then only be accessed as a source.[1] All changeable expressions have type `changeable`, and are used in a "destination passing" style—they do not return a value, but rather take a destination to which they write a value. Correct usage requires that a changeable expression ends with a `write`—we define "ends with" more precisely when we discuss time stamps. The destination written will be referred to as the *target* destination. The type changeable has no interpretable value.

The `mod` takes two parameters, a conservative comparison function and an *initializer*. A conservative comparison function returns `false` when the values are different but may return `true` or `false` when the values are the same. This function is used by the change-propagation algorithm to avoid unnecessary

[1]The library does not enforce this restriction statically, but can enforce it with run-time checks. In the following discussion we will use the term "correct usage" to describe similar restrictions in which run-time checks are needed to check correctness. The language described in Section 11.3 enforces all these restrictions statically using a modal type system.

propagation. The `mod` function creates a modifiable and applies the initializer to the new modifiable's destination. The initializer is responsible for writing the modifiable. Its body is therefore a changeable expression, and correct usage requires that the body's target match the initializer's argument. When the initializer completes, `mod` returns the source handle of the modifiable it created.

The `read` takes the source of a modifiable and a *reader*, a function whose body is changeable. The `read` accesses the contents of the modifiable and applies the reader to it. Any application of `read` is itself a changeable expression since the value being read could change. If a call R_a to `read` is within the dynamic scope of another call R_b to `read`, we say that R_a is *contained* within R_b. This relation defines a hierarchy on the reads, which we will refer to as the *containment hierarchy* (of reads).

11.2.2 Making an Application Adaptive

The transformation of a non-adaptive program into an adaptive program involves two steps. First, the input data structures are made "modifiable" by placing desired elements in modifiables. Second, the original program is updated by making the reads of modifiables explicit and placing the results of each expression that depends on a modifiable into another modifiable. This means that all values that directly or indirectly depend on modifiable inputs are placed in modifiables. The changes to the program are therefore determined by what parts of the input data structure are made modifiable.

As an example, consider the code for a standard Quicksort, `qsort`, and an adaptive Quicksort, `qsort'`, as shown in Figure 11.2. To avoid linear-time concatenations, `qsort` uses an accumulator to store the sorted tail of the input list. The transformation is done in two steps. First, we make the lists "modifiable" by placing the tail of each list element into a modifiable as shown in lines 1,2,3 in Figure 11.2. (If desired, each element of the list could have been made modifiable as well; this would allow changing an element without changing the list structurally). The resulting structure, *a modifiable list*, allows the user to insert and delete items to and from the list. Second, we change the program so that the values placed in modifiables are accessed explicitly via a `read`. The adaptive Quicksort uses a `read` (line 21) to determine whether the input list `l` is empty and writes the result to a destination `d` (line 23). This destination belongs to the modifiable that is created by a call to `mod` (through `modl`) in line 28 or 33. These modifiables form the output list, which now is a modifiable list. The function `filter` is similarly transformed into an adaptive one, `filter'` (lines 6-18). The `modl` function takes an initializer and passes it to the `mod` function with a constant-time, conservative comparison function for lists. The comparison function returns `true`, if and only if both lists are `NIL` and returns `false` otherwise. This comparison function is sufficiently powerful to prove the $O(\log n)$ bound for adaptive Quicksort.

11.2.3 Adaptivity

An adaptive programs allows the programmer to change the input to the program and update the result by running change propagation. This process can be repeated as desired. The library provides the meta-function `change` to change the value of a modifiable and the meta-function `propagate` to propagate these changes to the output. Figure 11.3 illustrates an example. The function `fromList` converts a list to a modifiable list, returning both the modifiable list and its last element. The `test` function first performs an initial evaluation of the adaptive Quicksort by converting the input list `lst` to a modifiable list `l` and sorting it into `r`. It then changes the input by adding a new key `v` to the end of `l`. To update the output `r`, `test` calls `propagate`.

```
datatype 'a list =
    nil
  | cons of ('a * 'a list)



fun fil test l =
let
  fun f(l) =
    case l of
      nil => nil
    | cons(h,t) =>
      if test(h) then
        cons(h,f t)
      else
        f(t)
in
  f(l)
end

fun qsort(l) =
let
  fun qs(l,r) =
    case l of
      nil => r
    | cons(h,t) =>
    let
      val l = fil (fn x => x<h) t
      val g = fil (fn x => x>=h) t
      val gs = qs(g,r)
    in
      qs(l,cons(h,gs))
    end
in
  qs(l,nil)
end
```

```
datatype 'a list' =
    NIL
  | CONS of ('a * 'a list' mod)

fun modl f =
  mod (fn (NIL,NIL) => true
        |    _       => false) f

fun fil' test l =
let
  fun f(l,d) = read(l, fn l' =>
    case l' of
      NIL => write(d, NIL)
    | CONS(h,t) =>
      if test(h) then write(d,
        CONS(h,modl(fn d => f(t,d))))
      else
        f(t,d))
in
  modl(fn d => f(l, d))
end

fun qsort'(l) =
let
  fun qs(l,r,d) = read(l, fn l' =>
    case l' of
      NIL => write(d, r)
    | CONS(h,t) =>
    let
      val l = fil (fn x => x<h) t
      val g = fil (fn x => x>=h) t
      val gs = modl(fn d => qs(g,r,d))
    in
      qs(l,CONS(h,gs),d)
    end)
in
  modl(fn d => qs(l,NIL,d))
end
```

Figure 11.2: The complete code for non-adaptive (left) and adaptive (right) Quicksort.

The update will result in a list identical to what would have been returned if `v` was added to the end of `l` before the call to `qsort`. In general, any number of inputs could be changed before running `propagate`.

```
fun new(v) = modl(fn d => write(d,v))

fun fromList(l) =
  case l of
    nil  => let val m = new(NIL) in (m,m) end
  | h::t => let val (l,last) = fromList(t) in
              (new(CONS(h,l)),last)
            end

fun test(lst,v) =
let val _ = init()
    val (l,last) = fromList(lst)
    val r = qsort'(l)
in
  (change(last,CONS(v,new(NIL)));
   propagate();
   r)
end
```

Figure 11.3: Example of changing input and change propagation for Quicksort.

11.2.4 Dynamic Dependence Graphs

The crucial issue is to support change propagation efficiently. To do this, an adaptive program, as it evaluates, creates a record of the adaptive activity in the form of a dynamic dependence graph. Dynamic dependence graphs are defined and formalized in Chapter 4. In this chapter, we use a version of DDGs that is similar to the data structure described in Chapter 4. The key difference between this version and the standard DDGs is that, this version only tracks modifiable references and reads, instead of all memory locations and all function calls. The correspondence between modifiable reference and memory locations, and between reads and function calls is not *ad hoc*. This correspondence show how AFL enables applying dependence tracking selectively. For the rest of this chapter, the terms "dynamic dependence graph" and DDG will refer to this particular version unless otherwise stated.

In a dynamic dependence graph, each node represents a modifiable and each edge represents a read. An evaluation of `mod` adds a node, and an evaluation of `read` adds an edge to the graph. In a `read`, the node being read becomes the source, and the target of the read (the modifiable that the reader finished by writing to) becomes the target. We also tag the edges with the reader function, which is a *closure* (*i.e.*, a function and an environment consisting of values for its free variables). We say that a node (and corresponding modifiable) is an *input* if it has no incoming edges.

When the input to an adaptive program changes, a change propagation algorithm updates the output and the dynamic dependence graph by propagating changes through the graph and re-executing the reads affected by the change. When re-evaluated on the changed source, a read can, due-to conditionals, create completely new reads than it previously did. It is therefore critical that the reads created by the previous execution of a read are deleted from the graph. This is done by maintaining a *containment hierarchy* between reads. A read e is *contained* within another read e' if e was created during the execution of e'. During change propagation, the reads contained in a re-evaluated read are removed from the graph.

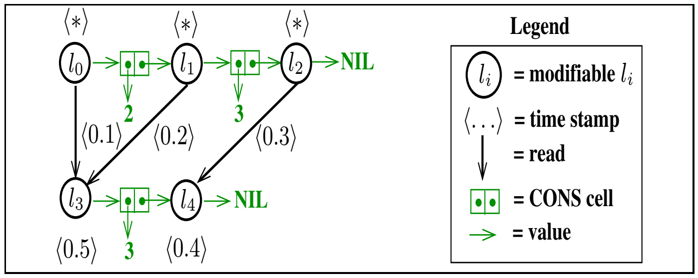


Figure 11.4: The DDG for an application of `filter′` to the modifiable list `2::3::nil`.

Containment hierarchy is represented using time-stamps generated by a virtual clock (or order-maintenance) data structure (Section 6.1.1). Each edge and node in the dynamic dependence graph is tagged with a *time-stamp* corresponding to its execution "time" in the sequential execution order. Time stamps are generated by the `mod` and `read` expressions. The time stamp of an edge is generated by the corresponding `read`, before the reader is evaluated, and the time stamp of a node is generated by the `mod` after the initializer is evaluated (the time corresponds to the time that the node is initialized). Correct usage of the library requires that the order of time stamps is independent of whether the `write` or `mod` generate the time stamp for the corresponding node. This is what we mean by saying that a changeable expression must end with a `write` to its target.

The time stamp of an edge is called its *start time* and the time stamp of the target of the edge is called the edge's *stop time*. The start and the stop time of the edge define the *time interval* of the edge. Time intervals are then used to identify the containment relationship of reads: a read R_a is contained in a read R_b if and only if the time interval of the edge associated with R_a is within the time interval of the edge associated with R_b. For now, we will represent time stamps with real numbers.

As an example for dynamic dependence graphs, consider the adaptive filter function `filter′` shown in Figure 11.2. The function takes another function `f` and a modifiable list `l` as parameters and outputs a modifiable list that contains the items of `l` satisfying `f`. Figure 11.4 shows the dependence graph for an evaluation of `filter′` with the function `(fn x => x > 2)` and a modifiable input list of `2::3::nil`. The output is the modifiable list `3::nil`. Although not shown in the figure, each edge is also tagged with a reader. In this example, all edges have an instance of reader `(fn l′ => case l′ of ...)` (lines 8-15 of `qsort′` in Figure 11.2). The time stamps for input nodes are not relevant, and are marked with stars in Figure 11.4. We note that readers are closures, *i.e.*, code with captured environments. In particular, each of the readers in our example have their source and their target in their environment.

11.2.5 Change Propagation

Given a dynamic dependence graph and a set of changed input modifiables, the *change-propagation algorithm* updates the DDG and the output by propagating changes in the DDG. The change-propagation algorithm for DDGs is described in Section 4.4. In this section, we describe a version of that algorithm to the particular version of DDGs that we use here.

The idea behind the change-propagation algorithm is to re-evaluate the reads that are affected by the

Propagate Changes

I `is the set of changed inputs`
$(V, E) = G$ `is an DDG`
1 $Q := \bigcup_{v \in I}$ `outEdges`(v)
2 `while` Q `is not empty`
3 $\quad e :=$ `deleteMin`(Q)
4 $\quad (T_s, T_e) :=$ `timeInterval`(e)
5 $\quad V := V - \{v \in V | T_s < T(v) < T_e\}$
6 $\quad E' := \{e' \in E | T_s < T(e') < T_e\}$
7 $\quad E := E - E'$
8 $\quad Q := Q - E'$
9 $\quad v' :=$ `apply(reader(`e`), val(src(`e`)))` `in time` (T_s, T_e)
10 $\quad$ `if` $v' \neq$ `val(target(`e`))` `then`
11 $\qquad$ `val(target(`e`))` = v'
12 $\qquad Q := Q +$ `outEdges(target(`e`))`

Figure 11.5: The change-propagation algorithm.

input change in the sequential execution order. Re-executing the reads in the sequential execution order ensures that the source of an edge (read) is updated before the re-execution of that read. We say that an edge or read, is *affected* if the source of the edge changes value. For correctness, it is critical that the edges that are created by an edge e, *i.e.*, the edges contained in e, are deleted from the graph when e is re-evaluated. We say that an edge is *obsolete* if it is contained within an affected edge.

Figure 11.5 shows the change-propagation algorithm. The algorithm maintains a priority queue of affected edges. The queue is prioritized on the time stamp of each edge, and is initialized with the out-edges of the changed input values. Each iteration of the while loop processes one affected edge—each iteration is called an *edge update*. An edge update re-evaluates the reader associated with the edge. Re-evaluation makes any code that was within the reader's dynamic scope obsolete. A key aspect of the algorithm is that when an edge is updated, all nodes and edges that are contained within that edge are deleted from both the graph and queue. This prevents the reader of an obsolete edge from being re-evaluated. After the reader is re-evaluated the algorithm checks if the value of the target has changed (line 10) by using the conservative comparison function passed to `mod`. If the target has changed, the out-edges of the target are added to the queue to propagate that change.

As an example, consider an initial evaluation of `filter` whose dependence graph is shown in Figure 11.4. Now, suppose we change the modifiable input list from `2::3::nil` to `2::4::7::nil` by creating the modifiable list `4::7::nil` and changing the value of modifiable l_1 to this list. The top left frame in Figure 11.6 shows the input change. Now, we run the change-propagation algorithm to update the output. First, we insert the sole outgoing edge of l_1, namely (l_1,l_3), into the queue. Since this is the only (hence, the earliest) edge in the queue, we remove it from the queue and establish the current time-interval as $\langle 0.2\rangle$-$\langle 0.5\rangle$. Next, we delete all the nodes and edges contained in this edge from the DDG and from the queue (which is empty) as shown by the top right frame in Figure 11.6. Then we redo the read by re-

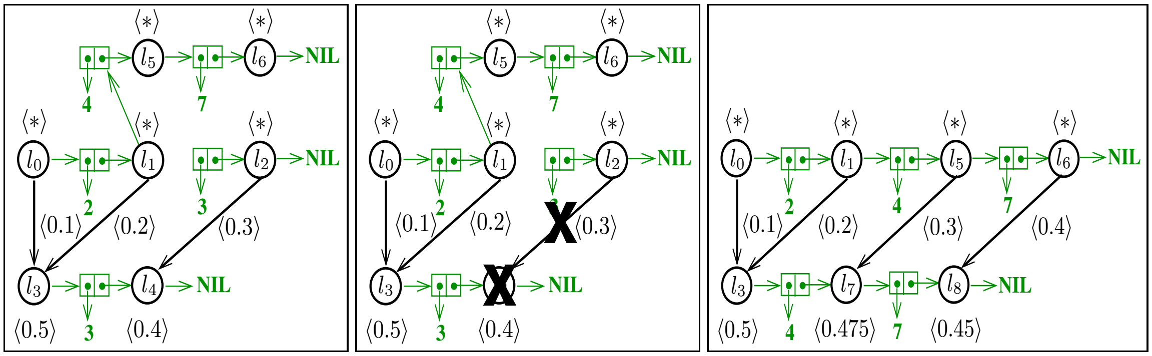


Figure 11.6: Snapshots of the DDG during change propagation.

```
1  fun factorial (n:int mod, p:bool mod) =
2    let
3      fun fact (n:int mod) =
4        if (n = 0) then 1
5        else n * fact(n-1)
6    in
7      modl (fn d =>
8      read (p, fn p' =>
9        if (not p') then write (d,1)
10       else read (n, fn n' => write (d, fact n'))))
11 end
```

Figure 11.7: The adaptive code for `factorial`.

evaluating the reader `(fn l' => case l' of ...)` (8-15 in Figure 11.2) in the current time interval $\langle 0.2\rangle$-$\langle 0.5\rangle$. The reader walks through the modifiable list `4::7::nil` as it filters the items and writes the head of the result list to l_3, as shown in the bottom frame in Figure 11.6. This creates two new edges, which are given the time stamps, $\langle 0.3\rangle$, and $\langle 0.4\rangle$. The targets of these edges, l_7 and l_8, are assigned the time stamps, $\langle 0.475\rangle$, and $\langle 0.45\rangle$, matching the order that they were initialized (these time stamps are otherwise chosen arbitrarily to fit in the range $\langle 0.4\rangle$-$\langle 0.5\rangle$). Note that after change propagation, the modifiables l_2 and l_4 become unreachable and can be garbage collected.

The reason for deleting obsolete reads and never re-evaluating them is that they may be inconsistent with a from-scratch evaluation of the adaptive program on the changed input. Re-evaluating an obsolete read can therefore change the semantics and the performance of the program. For example, it can cause non-termination, or raise an exception (assuming the language supports exceptions as ML does). As an example, consider the `factorial` function shown in Figure 11.7. The program takes an integer modifiable `n` and a boolean modifiable `p` whose value is `true` if the value of `n` is positive and `false` otherwise. Consider evaluating the function with a positive `n` with `p` set to `true`. Now change `n` to negative two and `p` to `false`. This change will make the read on line 8 affected and therefore the read on line 10 will be obsolete. With our change propagation algorithm, the read on line 8 will be re-evaluated and one will be written to the result modifiable—the obsolete on line 10 will not be re-evaluated. Note that re-evaluating the obsolete read on line 10 will call the function `fact` on negative two and will therefore cause non-termination.

11.2.6 The ML Implementation

We present an implementation of our adaptive mechanism in ML. The implementation is based on a library for virtual clocks and a standard priority queue. In the virtual-clock interface (shown in Figure 11.8), `delete` deletes all time stamps between two given time stamps and `isDeleted` returns `true` if the time stamp has been deleted and `false` otherwise.

```
signature VIRTUAL_CLOCK = sig
  type t

  val init :  unit -> t           (* Initialize *)
  val compare:  t*t -> order      (* Compare two nodes *)
  val insert :  t ref -> t        (* Insert a new node *)
  val delete:  t*t -> unit        (* Delete an interval *)
  val isDeleted:  t -> bool       (* Is the node deleted?  *)
end
```

Figure 11.8: The signature for virtual clocks.

Figure 11.9 shows the code for the ML implementation. The implementation differs somewhat from the earlier description. The `edge` and `node` types correspond to edges and nodes in the DDG. The reader and time-interval are represented explicitly in the `edge` type, but the source and destination are implicit in the reader. In particular the reader starts by reading the source, and ends by writing to the destination. The node consists of the corresponding modifiable's value (`value`), its out-edges (`out`), and a write function (`wrt`) that implements writes or changes to the modifiable. A time stamp is not needed since edges keep both start and stop times. The `timeNow` is used to help generate the sequential time stamps, which are generated for the edge on line 27 and for the node on line 22 by the write operation.

Some of the tasks assigned to the change-propagate loop in Figure 11.5 are performed by the write operation in the ML code. This includes the functionality of lines 10 and 12 in Figure 11.5, which are executed by lines 17 and 20 in the ML code. Another important difference is that the deletion of contained edges is done lazily. Instead of deleting edges from the queue and from the graph immediately, the time stamp of the edge is marked as affected (by being removed from the ordered-list data structure), and is deleted when it is next encountered. This can be seen in line 38.

We note that the implementation given does not include sufficient run-time checks to verify "correct usage". For example, the code does not verify that an initializer writes its intended destination. The code, however, does check for a read before write.

Recall that there is strong correspondence between the version of DDGs used here and the standard DDG data structure. The version of DDGs that we use here track modifiables instead of all memory locations, and reads instead of all function calls. Based on this correspondence, we can show that the presented implementation has constant-time overhead and supports change-propagation within the same time bounds as shown in Chapter 6.

```
structure Adaptive :> ADAPTIVE = struct
  type changeable = unit
  exception unsetMod

  type edge = {reader:(unit -> unit), timeInterval:(Time.t * Time.t)}
  type 'a node = {value:(unit -> 'a) ref, wrt:('a->unit) ref, out:edge list ref}
  type 'a mod = 'a node
  type 'a dest = 'a node

  val timeNow = ref(Time.init())
  val PQ = ref(Q.empty)             (* The priority queue *)

  fun init() = (timeNow := Time.init(); PQ := Q.empty)

  fun mod cmp f =
  let val value = ref(fn() => raise unsetMod)
      val wrt = ref(fn(v) => raise unsetMod)
      val out = ref(nil)
      val m = {value=value, wrt=wrt, out=out}
      fun change t v =
        (if cmp(v,(!value)()) then ()
         else (value := (fn() => v);
               List.app (fn x => PQ := Q.insert(x,!PQ)) (!out);
               out := nil);
         timeNow := t)
      fun write(v) = (value := (fn() => v);wrt := change(Time.insert(timeNow))
      val _ = wrt := write
  in f(m); m end

  fun write({wrt, ...} :  'a dest, v) = (!wrt)(v)

  fun read({value, out, ...} :  'a mod, f) =
  let val start = Time.insert(timeNow)
      fun run() =
        (f((!value)());
         out := {reader=run,timeInterval=(start,(!timeNow))}::(!out))
  in run() end

  fun change(l:  'a mod, v) = write(l, v)

  fun propagate'() =
    if (Q.isEmpty(!PQ)) then ()
    else let val (edge, pq) = Q.deleteMin(!PQ)
             val _ = PQ := pq
             val {reader=f,timeInterval=(start,stop)} = edge
         in
           if (Time.isDeleted start) then
             propagate'()                   (* Obsolete read, discard.*)
           else
             (Time.delete(start,stop);      (* Delete interval *)
              timeNow := start;
              f();                          (* Rerun the read *)
              propagate'())
         end

  fun propagate() = let val ctime = !timeNow in
    (propagate'(); timeNow := ctime)
  end
end
```

Figure 11.9: The implementation of the adaptive library.

11.3 An Adaptive Functional Language

The first half of this chapter described the adaptivity mechanism in an informal, algorithmic setting. The purpose was to introduce the basic concepts of adaptivity and show that the mechanism can be implemented efficiently. We now turn to the question of whether the proposed mechanism is sound. To this end, we present a small, purely functional language, called AFL, with primitives for adaptive computation. AFL ensures correct usage of the adaptivity mechanism statically by using a modal type system and employing implicit "destination passing."

The adaptivity mechanisms of AFL are similar to those of the adaptive library presented in Section 11.2. The chief difference is that the target of a changeable expression is implicit in AFL. Implicit passing of destinations is critical for ensuring various safety properties of the language.

AFL does not include analogues of the meta-operations for making and propagating changes as in the ML library. Instead, we give a direct presentation of the change-propagation algorithm in Section 11.5, which is defined in terms of the dynamic semantics of AFL given here. As with the ML implementation, the dynamic semantics must keep a record of the adaptive aspects of the computation. Rather than use DDG's, however, the semantics maintains this information in the form of a trace, which guides the change propagation algorithm. The trace representation simplifies the proof of correctness of the change propagation algorithm given in Section 11.5.

11.3.1 Abstract Syntax

The abstract syntax of AFL is given in Figure 11.10. We use the meta-variables x, y, and z (and variants) to range over an unspecified set of variables, and the meta-variable l (and variants) to range over a separate, unspecified set of locations—the locations are modifiable references. The syntax of AFL is restricted to "2/3-cps", or "named form", to streamline the presentation of the dynamic semantics.

The types of AFL include the base types `int` and `bool`; the stable function type, $\tau_1 \xrightarrow{\mathtt{S}} \tau_2$; the changeable function type, $\tau_1 \xrightarrow{\mathtt{C}} \tau_2$; and the type τ `mod` of modifiable references of type τ. Extending AFL with product, sum, recursive, or polymorphic types presents no fundamental difficulties, but they are omitted here for the sake of brevity.

Expressions are classified into two categories, the *stable* and the *changeable*. The value of a stable expression is not sensitive to modifications to the inputs, whereas the value of a changeable expression may, directly or indirectly, be affected by them. The familiar mechanisms of functional programming are embedded in AFL as stable expressions. These include basic types such as integers and booleans, and a sequential `let` construct for ordering evaluation. Ordinary functions arise in AFL as *stable functions*. The body of a stable function must be a stable expression; the application of a stable function is correspondingly stable. The stable expression $\mathtt{mod}_\tau\ e_c$ allocates a new modifiable reference whose value is determined by the changeable expression e_c. Note that the modifiable itself is stable, even though its contents is subject to change.

Changeable expressions are written in destination-passing style, with an implicit target. The changeable expression $\mathtt{write}_\tau(v)$ writes the value v of type τ into the target. The changeable expression $\mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ e_c\ \mathtt{end}$ binds the contents of the modifiable v to the variable x, then continues evaluation of e_c. A `read` is considered changeable because the contents of the modifiable on which it depends

Types	τ	$::=$	$\texttt{int} \mid \texttt{bool} \mid \tau\,\texttt{mod} \mid \tau_1 \xrightarrow{\mathsf{S}} \tau_2 \mid \tau_1 \xrightarrow{\mathsf{C}} \tau_2$
Values	v	$::=$	$c \mid x \mid l \mid \texttt{fun}_{\mathsf{S}}\, f(x:\tau_1):\tau_2\ \texttt{is}\ e_s\ \texttt{end} \mid$ $\texttt{fun}_{\mathsf{C}}\, f(x:\tau_1):\tau_2\ \texttt{is}\ e_c\ \texttt{end}$
Operators	o	$::=$	$\texttt{not} \mid + \mid - \mid = \mid < \mid \ldots$
Constants	c	$::=$	$n \mid \texttt{true} \mid \texttt{false}$
Expressions	e	$::=$	$e_s \mid e_c$
Stable Expressions	e_s	$::=$	$v \mid o\,(v_1,\ldots,v_n) \mid \texttt{apply}_{\mathsf{S}}\,(v_1,v_2) \mid$ $\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_s'\ \texttt{end} \mid \texttt{mod}_\tau\ e_c \mid$ $\texttt{if}\ v\ \texttt{then}\ e_s\ \texttt{else}\ e_s'$
Changeable Expressions	e_c	$::=$	$\texttt{write}_\tau\,(v) \mid \texttt{apply}_{\mathsf{C}}\,(v_1,v_2) \mid$ $\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_c\ \texttt{end} \mid$ $\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c\ \texttt{end} \mid$ $\texttt{if}\ v\ \texttt{then}\ e_c\ \texttt{else}\ e_c'$

Figure 11.10: The abstract syntax of AFL.

```
fun sum (l:int modlist):int =
  let fun sum' (l:int modlist, s:int, dest:int mod) =
        read (l, fn l' =>
          case l' of
            NIL => write (dest,s)
          | CONS(h,t) => sum' (t, s+h, dest))
  in mod (fn d => sum' (l,0,d)) end

funS sum (l:int modlist):int mod is
  let sum'' be funC sum' (l:int modlist, s:int):int is
        read l as l' in
          case l' of
            NIL => writeint(s)
          | CONS(h,t) => applyC (sum',(t,s+h))
        end
  in modint applyC (sum'',(l,0)) end
```

Figure 11.11: Function sum written with the ML library (top), and in AFL (bottom).

is subject to change. A changeable function itself is stable, but its body is changeable; correspondingly, the application of a changeable function is a changeable expression. The sequential `let` construct allows for the inclusion of stable sub-computations in changeable mode. Finally, conditionals with changeable branches are themselves changeable.

As an example, consider a function that sums up the keys in a modifiable list. Such a function could be written by traversing the list and accumulating a sum, which is written to the destination at the end. The code for this function using our ML library (Section 11.2) is shown in Figure 11.11 on the left. Note that all recursive calls to the function `sum'` share the same destination. The code for the `sum` function in AFL is shown in Figure 11.11 on the right assuming constructs for supporting lists and pattern matching. The critical difference between the two implementations is that in AFL, destinations are passed implicitly and that `sum'` function is a changeable function. Since `sum'` is changeable all recursive calls to it share the same destination that is created in the body of `sum`.

The advantage to sharing of destinations is performance. Consider for example calling `sum` on some list and changing the list by an insertion or deletion at the end. Propagating this change will take constant time and the result will be updated in constant time. If instead, each recursive call to `sum'` created its own destination and copied the result from the recursive call to its destination, then this change will propagate up the recursive-call tree and will take linear time. This is the motivation for including changeable functions in the AFL language.

11.3.2 Static Semantics

The AFL type system is inspired by the type theory of modal logic given by Pfenning and Davies [79]. We distinguish two modes, the *stable* and the *changeable*, corresponding to the distinction between terms and expressions, respectively, in Pfenning and Davies' work. However, they have no analogue of our changeable function type, and do not give an operational interpretation of their type system.

The judgement $\Lambda; \Gamma \vdash_{\mathsf{S}} e : \tau$ states that e is a well-formed stable expression of type τ, relative to Λ and Γ. The judgement $\Lambda; \Gamma \vdash_{\mathsf{C}} e : \tau$ states that e is a well-formed changeable expression of type τ, relative to Λ and Γ. Here Λ is a *location typing* and Γ is a *variable typing*; these are finite functions assigning types to locations and variables, respectively. (In Section 11.4 we will impose additional structure on location typings that will not affect the definition of the static semantics.)

The typing judgements for stable and changeable expressions are shown in Figures 11.12 and 11.13 respectively. For primitive functions, we assume a typing relation o. For stable expression, the interesting rules are the `mod` and the changeable functions. The bodies of these expressions are changeable expressions and therefore they are typed in the changeable mode. For changeable expressions, the interesting rule is the `let` rule. The body of `let` is a changeable expression and thus typed in the changeable mode; the expression bound, however, is a stable expression and thus typed in the stable mode. The `mod` and `let` rules therefore provide inclusion between two modes.

11.3.3 Dynamic Semantics

The evaluation judgements of AFL have one of two forms. The judgement $\sigma, e \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s$ states that evaluation of the stable expression e, relative to the input store σ, yields the value v, the trace $\mathtt{T}_s$, and the updated store σ'. The judgement $\sigma, l \leftarrow e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}_c$ states that evaluation of the changeable expression e, relative to the input store σ, writes its value to the target l, and yields the trace $\mathtt{T}_c$ and the updated store σ'.

In the dynamic semantics, a *store*, σ, is a finite function mapping each location in its domain, $\mathrm{dom}(\sigma)$, to either a value v or a "hole" $\square$. The *defined domain*, $\mathrm{def}(\sigma)$, of σ consists of those locations in $\mathrm{dom}(\sigma)$ not mapped to $\square$ by σ. When a location is created, it is assigned the value $\square$ to reserve that location while

$$\frac{}{\Lambda;\Gamma \vdash_{\mathsf{S}} n : \mathtt{int}}\ \textbf{(number)} \qquad \frac{}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{true} : \mathtt{bool}}\ \textbf{(true)} \qquad \frac{}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{false} : \mathtt{bool}}\ \textbf{(false)}$$

$$\frac{(\Lambda(l) = \tau)}{\Lambda;\Gamma \vdash_{\mathsf{S}} l : \tau\ \mathtt{mod}}\ \textbf{(location)} \qquad \frac{(\Gamma(x) = \tau)}{\Lambda;\Gamma \vdash_{\mathsf{S}} x : \tau}\ \textbf{(variable)}$$

$$\frac{\Lambda;\Gamma,\ f : \tau_1 \xrightarrow{\mathsf{S}} \tau_2,\ x : \tau_1 \vdash_{\mathsf{S}} e : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{fun}_{\mathsf{S}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end} : (\tau_1 \xrightarrow{\mathsf{S}} \tau_2)}\ \textbf{(fun/stable)}$$

$$\frac{\Lambda;\Gamma,\ f : \tau_1 \xrightarrow{\mathsf{C}} \tau_2,\ x : \tau_1 \vdash_{\mathsf{C}} e : \tau_2}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{fun}_{\mathsf{C}}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end} : (\tau_1 \xrightarrow{\mathsf{C}} \tau_2)}\ \textbf{(fun/changeable)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_i : \tau_i \quad (1 \leq i \leq n) \quad \vdash_o o : (\tau_1, \ldots, \tau_n)\ \tau}{\Lambda;\Gamma \vdash_{\mathsf{S}} o\,\mathtt{(}v_1, \ldots, v_n\mathtt{)}\ : \tau}\ \textbf{(primitive)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} x : \mathtt{bool} \quad \Lambda;\Gamma \vdash_{\mathsf{S}} e : \tau \quad \Lambda;\Gamma \vdash_{\mathsf{S}} e' : \tau}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{if}\,x\,\mathtt{then}\,e\,\mathtt{else}\,e' : \tau}\ \textbf{(if)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_1 : (\tau_1 \xrightarrow{\mathsf{S}} \tau_2) \quad \Lambda;\Gamma \vdash_{\mathsf{S}} v_2 : \tau_1}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{apply}_{\mathsf{S}}\,\mathtt{(}v_1, v_2\mathtt{)}\ : \tau_2}\ \textbf{(apply)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} e : \tau \quad \Lambda;\Gamma, x : \tau \vdash_{\mathsf{S}} e' : \tau'}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{let}\,x\,\mathtt{be}\,e\,\mathtt{in}\,e'\,\mathtt{end} : \tau'}\ \textbf{(let)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{C}} e : \tau}{\Lambda;\Gamma \vdash_{\mathsf{S}} \mathtt{mod}_\tau\ e : \tau\ \mathtt{mod}}\ \textbf{(modifiable)}$$

Figure 11.12: Typing of stable expressions.

its value is being determined. With a store σ, we associate a location typing Λ and write $\sigma : \Lambda$, if the store satisfies the typing Λ. This is defined formally in Section 11.4.

A *trace* is a finite data structure recording the adaptive aspects of evaluation. The abstract syntax of traces is given by the following grammar:

$$\begin{array}{llcl}
\textit{Trace} & \mathtt{T} & ::= & \mathtt{T}_s \mid \mathtt{T}_c \\
\textit{Stable} & \mathtt{T}_s & ::= & \epsilon \mid \langle \mathtt{T}_c \rangle_{l:\tau} \mid \mathtt{T}_s \ ;\ \mathtt{T}_s \\
\textit{Changeable} & \mathtt{T}_c & ::= & \mathtt{W}_\tau \mid R_l^{x.e}(\mathtt{T}_c) \mid \mathtt{T}_s \ ;\ \mathtt{T}_c
\end{array}$$

When writing traces, we adopt the convention that "**;**" is right-associative.

A stable trace records the sequence of allocations of modifiables that arise during the evaluation of a stable expression. The trace $\langle \mathtt{T}_c \rangle_{l:\tau}$ records the allocation of the modifiable, l, its type, τ, and the trace of

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{write}_\tau\mathtt{(}v\mathtt{)} : \tau}\ \textbf{(write)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} x : \mathtt{bool} \quad \Lambda;\Gamma \vdash_{\mathsf{C}} e : \tau \quad \Lambda;\Gamma \vdash_{\mathsf{C}} e' : \tau}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{if}\, x\, \mathtt{then}\, e\, \mathtt{else}\, e' : \tau}\ \textbf{(if)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v_1 : (\tau_1 \xrightarrow{\mathsf{C}} \tau_2) \quad \Lambda;\Gamma \vdash_{\mathsf{S}} v_2 : \tau_1}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{apply}_{\mathsf{C}}\mathtt{(}v_1, v_2\mathtt{)} : \tau_2}\ \textbf{(apply)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} e : \tau \quad \Lambda;\Gamma, x : \tau \vdash_{\mathsf{C}} e' : \tau'}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{let}\, x\, \mathtt{be}\, e\, \mathtt{in}\, e'\, \mathtt{end} : \tau'}\ \textbf{(let)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathsf{S}} v : \tau\, \mathtt{mod} \quad \Lambda;\Gamma, x : \tau \vdash_{\mathsf{C}} e : \tau'}{\Lambda;\Gamma \vdash_{\mathsf{C}} \mathtt{read}\, v\, \mathtt{as}\, x\, \mathtt{in}\, e\, \mathtt{end} : \tau'}\ \textbf{(read)}$$

Figure 11.13: Typing of changeable expressions.

the initialization code for l. The trace $\mathtt{T}_s$; $\mathtt{T}'_s$ results from evaluation of a `let` expression in stable mode, the first trace resulting from the bound expression, the second from its body.

A changeable trace has one of three forms. A write, $\mathtt{W}_\tau$, records the storage of a value of type τ in the target. A sequence $\mathtt{T}_s$; $\mathtt{T}_c$ records the evaluation of a `let` expression in changeable mode, with $\mathtt{T}_s$ corresponding to the bound stable expression, and $\mathtt{T}_c$ corresponding to its body. A read $R_l^{x.e}(\mathtt{T}_c)$ trace specifies the location read, l, the context of use of its value, $x.e$, and the trace, $\mathtt{T}_c$, of the remainder of evaluation with the scope of that read. This records the dependency of the target on the value of the location read.

We define the *domain* $\mathrm{dom}(\mathtt{T})$ of a trace $\mathtt{T}$ as the set of locations read or written in the trace $\mathtt{T}$. The *defined domain* $\mathrm{def}(\mathtt{T})$ of a trace $\mathtt{T}$ is the set of locations written in the trace $\mathtt{T}$. Formally the domain and the defined domain of traces are defined as

$$\begin{array}{lcl@{\qquad}lcl}
\mathrm{def}(\varepsilon) & = & \emptyset & \mathrm{dom}(\varepsilon) & = & \emptyset \\
\mathrm{def}(\langle \mathtt{T}_c \rangle_{l:\tau}) & = & \mathrm{def}(\mathtt{T}_c) \cup \{l\} & \mathrm{dom}(\langle \mathtt{T}_c \rangle_{l:\tau}) & = & \mathrm{dom}(\mathtt{T}_c) \cup \{l\} \\
\mathrm{def}(\mathtt{T}_s\, ;\, \mathtt{T}'_s) & = & \mathrm{def}(\mathtt{T}_s) \cup \mathrm{def}(\mathtt{T}'_s) & \mathrm{dom}(\mathtt{T}_s\, ;\, \mathtt{T}'_s) & = & \mathrm{dom}(\mathtt{T}_s) \cup \mathrm{dom}(\mathtt{T}'_s) \\
\mathrm{def}(\mathtt{W}_\tau) & = & \emptyset & \mathrm{dom}(\mathtt{W}_\tau) & = & \emptyset \\
\mathrm{def}(R_l^{x.e}(\mathtt{T}_c)) & = & \mathrm{def}(\mathtt{T}_c) & \mathrm{dom}(R_l^{x.e}(\mathtt{T}_c)) & = & \mathrm{dom}(\mathtt{T}_c) \cup \{l\} \\
\mathrm{def}(\mathtt{T}_s\, ;\, \mathtt{T}_c) & = & \mathrm{def}(\mathtt{T}_s) \cup \mathrm{def}(\mathtt{T}_c) & \mathrm{dom}(\mathtt{T}_s\, ;\, \mathtt{T}_c) & = & \mathrm{dom}(\mathtt{T}_s) \cup \mathrm{dom}(\mathtt{T}_c).
\end{array}$$

The dynamic dependency graphs described in Section 11.2 may be seen as an efficient representation of traces. Time stamps may be assigned to each read and write operation in the trace in left-to-right order. These correspond to the time stamps in the DDG representation. The containment hierarchy is directly represented by the structure of the trace. Specifically, the trace $\mathtt{T}_c$ (and any read in $\mathtt{T}_c$) is contained within

$$\frac{}{\sigma, v \Downarrow^{\mathsf{S}} v, \sigma, \varepsilon}\ \textbf{(value)}$$

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

$$\frac{\sigma, e \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}{\sigma, \mathtt{if\,true\,then}\,e\,\mathtt{else}\,e' \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}\ \textbf{(if/true)}$$

$$\frac{\sigma, e' \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}{\sigma, \mathtt{if\,false\,then}\,e\,\mathtt{else}\,e' \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}\ \textbf{(if/false)}$$

$$\frac{\begin{array}{c}(v_1 = \mathtt{fun}_{\mathsf{S}}\ f(x:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end})\\ \sigma, [v_1/f, v_2/x]\ e \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s\end{array}}{\sigma, \mathtt{apply}_{\mathsf{S}}\,(v_1, v_2)\ \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s}\ \textbf{(apply)}$$

$$\frac{\begin{array}{lll}\sigma, e & \Downarrow^{\mathsf{S}} & v, \sigma', \mathtt{T}_s\\ \sigma', [v/x]e' & \Downarrow^{\mathsf{S}} & v', \sigma'', \mathtt{T}'_s\end{array}}{\sigma, \mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ e'\ \mathtt{end} \Downarrow^{\mathsf{S}} v', \sigma'', (\mathtt{T}_s\ ;\ \mathtt{T}'_s)}\ \textbf{(let)}$$

$$\frac{\sigma[l \to \square], l \leftarrow e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}_c \quad (l \not\in \mathrm{dom}(\sigma))}{\sigma, \mathtt{mod}_\tau\ e \Downarrow^{\mathsf{S}} l, \sigma', \langle \mathtt{T}_c \rangle_{l:\tau}}\ \textbf{(mod)}$$

Figure 11.14: Evaluation of stable expressions.

the read trace $R_l^{x.e}(\mathtt{T}_c)$.

Stable Evaluation. The evaluation rules for stable expressions are given in Figure 11.14. Most of the rules are standard for a store-passing semantics. For example, the `let` rule sequences evaluation of its two expressions, and performs binding by substitution. Less conventionally, it yields a trace consisting of the sequential composition of the traces of its sub-expressions.

The most interesting rule is the evaluation of $\mathtt{mod}_\tau\ e$. Given a store σ, a fresh location l is allocated and initialized to $\square$ prior to evaluation of e. The sub-expression e is evaluated in changeable mode, with l as the target. Pre-allocating l ensures that the target of e is not accidentally re-used during evaluation; the static semantics ensures that l cannot be read before its contents is set to some value v.

Each location allocated during the evaluation a stable expression is recorded in the trace and is written to: If $\sigma, e \Downarrow^{\mathsf{S}} v, \sigma', \mathtt{T}_s$, then $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma) \cup \mathrm{def}(\mathtt{T}_s)$, and $\mathrm{def}(\sigma') = \mathrm{def}(\sigma) \cup \mathrm{def}(\mathtt{T}_s)$. Furthermore, all locations read during evaluation are defined in the store, $\mathrm{dom}(\mathtt{T}_s) \subseteq \mathrm{def}(\sigma')$.

Changeable Evaluation. The evaluation rules for changeable expressions are given in Figure 11.15. The `let` rule is similar to the corresponding rule in stable mode, except that the bound expression, e, is evaluated in stable mode, whereas the body, e', is evaluated in changeable mode. The `read` expression sub-

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

$$\frac{\sigma, l \leftarrow e \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c}{\sigma, l \leftarrow \texttt{if}\,\texttt{true}\,\texttt{then}\,e\,\texttt{else}\,e' \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c} \ \textbf{(if/true)}$$

$$\frac{\sigma, l \leftarrow e' \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c}{\sigma, l \leftarrow \texttt{if}\,\texttt{false}\,\texttt{then}\,e\,\texttt{else}\,e' \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c} \ \textbf{(if/false)}$$

$$\frac{\begin{array}{c}(v_1 = \texttt{fun}_{\mathsf{C}}\, f(x:\tau_1) : \tau_2 \ \texttt{is}\ e\ \texttt{end}) \\ \sigma, l \leftarrow [v_1/f, v_2/x]\, e \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c\end{array}}{\sigma, l \leftarrow \texttt{apply}_{\mathsf{C}}(v_1, v_2) \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c} \ \textbf{(apply)}$$

$$\frac{\begin{array}{lll}\sigma, e & \Downarrow^{\mathsf{S}} & v, \sigma', \texttt{T}_s \\ \sigma', l \leftarrow [v/x]e' & \Downarrow^{\mathsf{C}} & \sigma'', \texttt{T}_c\end{array}}{\sigma, l \leftarrow \texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ e'\ \texttt{end} \ \Downarrow^{\mathsf{C}} \ \sigma'', (\texttt{T}_s\ ;\ \texttt{T}_c)} \ \textbf{(let)}$$

$$\frac{\sigma, l' \leftarrow [\sigma(l)/x]\, e \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c}{\sigma, l' \leftarrow \texttt{read}\ l\ \texttt{as}\ x\ \texttt{in}\ e\ \texttt{end} \ \Downarrow^{\mathsf{C}} \ \sigma', R_l^{x.e}(\texttt{T}_c)} \ \textbf{(read)}$$

Figure 11.15: Evaluation of changeable expressions.

stitutes the binding of location l in the store σ for the variable x in e, and continues evaluation in changeable mode. The read is recorded in the trace, along with the expression that employs the value read. The `write` rule simply assigns its argument to the target. Finally, application of a changeable function passes the target of the caller to the callee, avoiding the need to allocate a fresh target for the callee and a corresponding read to return its value to the caller.

Each location allocated during the evaluation a changeable expression is recorded in the trace and is written; the destination is also written: If $\sigma, l \leftarrow e \ \Downarrow^{\mathsf{C}} \ \sigma', \texttt{T}_c$, then $\text{dom}(\sigma') = \text{dom}(\sigma) \cup \text{def}(\texttt{T}_c)$, and $\text{def}(\sigma') = \text{def}(\sigma) \cup \text{def}(\texttt{T}_c) \cup \{l\}$. Furthermore, all locations read during evaluation are defined in the store, $\text{dom}(\texttt{T}_c) \subseteq \text{def}(\sigma')$.

11.4 Type Safety of AFL

The static semantics of AFL ensures these five properties of its dynamic semantics: (1) each modifiable is written exactly once; (2) no modifiable is read before it is written; (3) dependencies are not lost, *i.e.*, if a value depends on a modifiable, then its value is also placed in a modifiable; (4) the store is acyclic and (5) the data dependences (dynamic dependence graph) is acyclic. These properties are critical for correctness of the adaptivity mechanisms. The last two properties show that AFL is consistent with pure functional programming by ensuring that no cycles arise during evaluation.

The write-once property (1) and no-lost-dependencies property (3) are relatively easy to observe. A write can only take place in the changeable mode and can write to the current destination. Since being the changeable mode requires the creation of a new destination by the `mod` construct, and only the current destination can be written, each modifiable is written exactly once. For property 3, note that dependencies are created by read operations, which take place in the changeable mode, are recorded in the trace, and end with a write. Thus, dependences are recorded and the result of a read is always written to a destination. The proof that the store is acyclic is more involved. We order locations (modifiables) of the store with respect to the times that they are written and require that the value of each expression typecheck with respect to the locations written before that expression. The total order directly implies that the store is acyclic (property 4), *i.e.*, no two locations refer to each other. The restriction that an expression typechecks with respect to the previously written locations ensures that no location is read before it is written (property 2). This fact along with the total ordering on locations implies that there are no cyclic dependences, *i.e.*, the dynamic dependence graph is acyclic (property 5).

The proof of type safety for AFL hinges on a type preservation theorem for the dynamic semantics. Since the dynamic semantics of AFL is given by an evaluation relation, rather than a transition system, the proof of type safety is indirect. First, we prove the type preservation theorem stating that the outcome of evaluation is type consistent, provided that the inputs are. Second, we prove a canonical forms lemma characterizing the "shapes" of closed values of each type. Third, we augment the dynamic semantics with rules stating that evaluation "goes wrong" in the case that the principal argument of an elimination form is non-canonical. Finally, we argue that, by the first two results, these rules can never apply to a well-typed program. Since the last two steps are routine, given the first two, we concentrate on preservation and canonical forms.

11.4.1 Location Typings

For the safety proof we will enrich location typings with a total ordering on locations that respects the order that they are written to. A location typing, Λ, consists of three parts:

1. A finite set, $\mathrm{dom}(\Lambda)$, of locations, called the *domain* of the store typing.
2. A finite function, also written Λ, assigning types to the locations in $\mathrm{dom}(\Lambda)$.
3. A linear ordering $\leq_\Lambda$ of $\mathrm{dom}(\Lambda)$.

The relation $l <_\Lambda l'$ holds if and only if $l \leq_\Lambda l'$ and $l \neq l'$. The restriction, $\leq_\Lambda \restriction L$, of $\leq_\Lambda$ to a subset $L \subseteq \mathrm{dom}(\Lambda)$ is the intersection $\leq_\Lambda \cap (L \times L)$.

As can be expected, stores are extended with respect to the total order: the *ordered extension*, $\Lambda[l'{:}\tau'{<}l]$, of a location typing Λ assigns the type τ' to the location $l' \notin \mathrm{dom}(\Lambda)$ and places l' immediately before $l \in \mathrm{dom}(\Lambda)$ such that

1. $\mathrm{dom}(\Lambda') = \mathrm{dom}(\Lambda) \cup \{\, l' \,\}$;
2. $\Lambda'(l'') = \begin{cases} \tau' & \textit{if } l'' = l' \\ \Lambda(l'') & \textit{otherwise;} \end{cases}$
3. (a) $l' \leq_{\Lambda'} l$;

(b) if $l'' \leq_{\Lambda} l$, then $l'' \leq_{\Lambda'} l'$;

(c) if $l'' \leq_{\Lambda} l'''$, then $l'' \leq_{\Lambda'} l'''$.

Location typings are partially ordered with respect to a containment relation by defining $\Lambda \sqsubseteq \Lambda'$ if and only if

1. $\mathrm{dom}(\Lambda) \subseteq \mathrm{dom}(\Lambda')$;
2. if $l \in \mathrm{dom}(\Lambda)$, then $\Lambda'(l) = \Lambda(l)$;
3. $\leq_{\Lambda'} \upharpoonright \mathrm{dom}(\Lambda) =\leq_{\Lambda}$.

Ordered extensions yield bigger stores: if $l \in \mathrm{dom}(\Lambda)$ and $l' \notin \mathrm{dom}(\Lambda)$, then $\Lambda \sqsubseteq \Lambda[l'{:}\tau'{<}l]$.

Forcing an ordering on the locations of a store suffices to show that the store is acyclic. It does not however help ensure that locations are not read before they are written. We therefore restrict an expression to depend only on those locations that have been written. How can we know what locations are written? At any point in evaluation, we call the last allocated but not yet written location the *cursor* and require that an expression depend only on locations prior to the cursor. The cursor will be maintained such that all locations that precede the cursor have been written and those that come after have not. We therefore define the *restriction* $\Lambda \upharpoonright l$, of a location typing Λ to a location $l \in \mathrm{dom}(\Lambda)$ is defined as the location typing Λ' such that

1. $\mathrm{dom}(\Lambda') = \{\, l' \in \mathrm{dom}(\Lambda) \mid l' <_{\Lambda} l \,\}$;
2. if $l' <_{\Lambda} l$, then $\Lambda'(l') = \Lambda(l')$;
3. $\leq_{\Lambda'} = \leq_{\Lambda} \upharpoonright \mathrm{dom}(\Lambda')$.

Note that if $\Lambda \sqsubseteq \Lambda'$ and $l \in \mathrm{dom}(\Lambda)$, then $\Lambda \upharpoonright l \sqsubseteq \Lambda' \upharpoonright l$.

Definition 68 (Store Typing)
A store σ may be assigned a location typing Λ, written $\sigma : \Lambda$, if and only if the following two conditions are satisfied.

1. *$\mathrm{dom}(\sigma) = \mathrm{dom}(\Lambda)$.*
2. *for each $l \in \mathrm{def}(\sigma)$, $\Lambda \upharpoonright l \vdash_{S} \sigma(l) : \Lambda(l)$.*

The location typing, Λ, imposes a linear ordering on the locations in the store, σ, such that the values in σ store have the types assigned to them by Λ, relative to the types of its preceding locations in the ordering.

11.4.2 Trace Typing

The formulation of the type safety theorem requires a notion of typing for traces. The judgement $\Lambda, l_0 \vdash_{\mathtt{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$ states that the stable trace $\mathtt{T}_s$ is well-formed relative to the input location typing Λ and the cursor $l_0 \in \mathrm{dom}(\Lambda')$. The output location typing Λ' is an extension of Λ with typings for the locations allocated by

$$\frac{}{\Lambda, l_0 \vdash_{\mathsf{S}} \varepsilon \rightsquigarrow \Lambda} \qquad \frac{\Lambda, l_0 \vdash_{\mathsf{S}} \mathtt{T}_s \rightsquigarrow \Lambda' \quad \Lambda', l_0 \vdash_{\mathsf{S}} \mathtt{T}'_s \rightsquigarrow \Lambda''}{\Lambda, l_0 \vdash_{\mathsf{S}} \mathtt{T}_s \,;\, \mathtt{T}'_s \rightsquigarrow \Lambda''}$$

$$\frac{\Lambda[l{:}\tau{<}l_0], l \vdash_{\mathsf{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda' \quad (l \notin \mathrm{dom}(\Lambda))}{\Lambda, l_0 \vdash_{\mathsf{S}} \langle \mathtt{T}_c \rangle_{l:\tau} \rightsquigarrow \Lambda'}$$

$$\frac{}{\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{W}_\tau : \tau \rightsquigarrow \Lambda} \qquad \frac{\Lambda, l_0 \vdash_{\mathsf{S}} \mathtt{T}_s \rightsquigarrow \Lambda' \quad \Lambda', l_0 \vdash_{\mathsf{S}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda''}{\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_s \,;\, \mathtt{T}_c : \tau \rightsquigarrow \Lambda''}$$

$$\frac{\Lambda \upharpoonright l_0 ; x{:}\tau \vdash_{\mathsf{C}} e : \tau' \quad \Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau' \rightsquigarrow \Lambda' \quad (l <_\Lambda l_0,\ \Lambda(l) = \tau)}{\Lambda, l_0 \vdash_{\mathsf{C}} R_l^{x.e}(\mathtt{T}_c) : \tau' \rightsquigarrow \Lambda'}$$

Figure 11.16: Typing of Traces.

the trace; these will all be ordered prior to the cursor. When Λ' is not important, we simply write $\Lambda \vdash_{\mathsf{S}} \mathtt{T}_s\ \mathsf{ok}$ to mean that $\Lambda \vdash_{\mathsf{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$ for some Λ'.

Similarly, the judgement $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$ states that the changeable trace $\mathtt{T}_c$ is well-formed relative to Λ and $l_0 \in \mathrm{dom}(\Lambda)$. As with stable traces, Λ' is an extension of Λ with the newly-allocated locations of the trace. When Λ' is not important, we write $\Lambda \vdash_{\mathsf{C}} \mathtt{T}_c : \tau$ for $\Lambda \vdash_{\mathsf{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$ for some Λ'.

The rules for deriving these judgements are given in Figure 11.16. The input location typing specifies the active locations, of which only those prior to the cursor are eligible as subjects of a read; this ensure a location is not read before it is written. The cursor changes when processing an allocation trace to make the allocated location active, but unreadable, thereby ensuring that no location is read before it is allocated. The output location typing determines the ordering of locations allocated by the trace relative to the ordering of the input locations. Specifically, the ordering of the newly allocated locations is determined by the trace, and is such that they are all ordered to occur immediately prior to the cursor. The ordering so determined is essentially the same as that used in the implementation described in Section 11.2.

The following invariants hold for traces and trace typings.

1. $\forall l.\, l \in \mathrm{def}(\mathtt{T})$, l is written exactly once: l appears once in a write position in $\mathtt{T}$ of the form $\langle \mathtt{T}_c \rangle_{l:\tau}$ for some $\mathtt{T}_c$.
2. If $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$, then $\mathrm{dom}(\Lambda') = \mathrm{dom}(\Lambda) \cup \mathrm{def}(\mathtt{T}_c)$ and $\mathrm{dom}(\mathtt{T}_c) \subseteq \mathrm{dom}(\Lambda')$.
3. If $\Lambda, l_0 \vdash_{\mathsf{S}} \mathtt{T}_s \rightsquigarrow \Lambda'$, then $\mathrm{dom}(\Lambda') = \mathrm{dom}(\Lambda) \cup \mathrm{def}(\mathtt{T}_s)$ and $\mathrm{dom}(\mathtt{T}_s) \subseteq \mathrm{dom}(\Lambda')$.

11.4.3 Type Preservation

For the proof of type safety we shall make use of a few technical lemmas. First, typing is preserved by addition of typings of "irrelevant" locations and variables.

Lemma 69 (Weakening)
If $\Lambda \sqsubseteq \Lambda'$ and $\Gamma \subseteq \Gamma'$, then

1. *if $\Lambda; \Gamma \vdash_{\mathcal{S}} e : \tau$, then $\Lambda'; \Gamma' \vdash_{\mathcal{S}} e : \tau$;*

2. *if* $\Lambda; \Gamma \vdash_C e : \tau$, *then* $\Lambda'; \Gamma' \vdash_C e : \tau$;

3. *if* $\Lambda \vdash_S T_s$ ok, *then* $\Lambda' \vdash_S T_s$ ok;

4. *if* $\Lambda \vdash_C T_c : \tau$, *then* $\Lambda' \vdash_C T_c : \tau$.

Second, typing is preserved by substitution of a value for a free variable of the same type as the value.

Lemma 70 (Value Substitution)
Suppose that $\Lambda; \Gamma \vdash_S v : \tau$.

1. *If* $\Lambda; \Gamma, x{:}\tau \vdash_S e' : \tau'$, *then* $\Lambda; \Gamma \vdash_S [v/x]e' : \tau'$.

2. *If* $\Lambda; \Gamma, x{:}\tau \vdash_C e' : \tau'$, *then* $\Lambda; \Gamma \vdash_C [v/x]e' : \tau'$.

The type preservation theorem for AFL states that the result of evaluation of a well-typed expression is itself well-typed. The location l_0, called the cursor, is the current allocation point. All locations prior to the cursor are written to, and location following the cursor are allocated but not yet written. All new locations are allocated prior to l_0 in the ordering and the newly allocated location becomes the cursor. The theorem requires that the input expression be well-typed relative to those locations preceding the cursor so as to preclude forward references to locations that have been allocated, but not yet initialized. In exchange the result is assured to be sensible relative to those locations prior to the cursor, all of which are allocated and initialized. This ensures that no location is read before it has been allocated and initialized.

Theorem 71 (Type Preservation)

1. *If*

   (a) $\sigma, e \Downarrow^S v, \sigma', T_s$,

   (b) $\sigma : \Lambda$,

   (c) $l_0 \in dom(\Lambda)$,

   (d) $l <_\Lambda l_0$ *implies* $l \in def(\sigma)$,

   (e) $\Lambda \upharpoonright l_0 \vdash_S e : \tau$,

   then there exists $\Lambda' \sqsupseteq \Lambda$ *such that*

   (f) $\Lambda' \upharpoonright l_0 \vdash_S v : \tau$,

   (g) $\sigma' : \Lambda'$, *and*

   (h) $\Lambda, l_0 \vdash_S T_s \leadsto \Lambda'$.

2. *If*

   (a) $\sigma, l_0 \leftarrow e \Downarrow^C \sigma', T_c$,

   (b) $\sigma : \Lambda$,

   (c) $\Lambda(l_0) = \tau_0$,

   (d) $l <_\Lambda l_0$ *implies* $l \in def(\sigma)$,

(e) $\Lambda \upharpoonright l_0 \vdash_C e : \tau_0$,

then

(f) $l_0 \in \mathit{def}(\sigma')$,

and there exists $\Lambda' \sqsupseteq \Lambda$ *such that*

(g) $\sigma' : \Lambda'$, *and*

(h) $\Lambda, l_0 \vdash_C T_c : \tau_0 \leadsto \Lambda'$.

Proof: Simultaneously, by induction on evaluation. We will consider several illustrative cases.

- Suppose that

  (1a) $\sigma, \mathtt{mod}_\tau\ e \ \Downarrow^{\mathsf{S}} l, \sigma'', \langle \mathtt{T}_c \rangle_{l:\tau}$;

  (1b) $\sigma : \Lambda$;

  (1c) $l_0 \in \mathrm{dom}(\Lambda)$;

  (1d) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$;

  (1e) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} \mathtt{mod}_\tau\ e : \tau\ \mathtt{mod}$.

  Since the typing and evaluation rules are syntax-directed, it follows that

  (1a(i)) $\sigma[l \to \Box], l \leftarrow e \ \Downarrow^{\mathsf{C}} \ \sigma'', \mathtt{T}_c$, where $l \notin \mathrm{dom}(\sigma)$, and

  (1e(i)) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{C}} e : \tau$.

  Note that $l \notin \mathrm{dom}(\Lambda)$, by (1b). Let $\sigma' = \sigma[l \to \Box]$ and let $\Lambda' = \Lambda[l{:}\tau{<}l_0]$. Note that $\Lambda \sqsubseteq \Lambda'$ and that $\Lambda'(l) = \tau$.

  Then we have

  (2a$'$) $\sigma', l \leftarrow e \ \Downarrow^{\mathsf{C}} \ \sigma'', \mathtt{T}_c$, by (1a(i));

  (2b$'$) $\sigma' : \Lambda'$. Since $\sigma : \Lambda$ by (1b), we have $\Lambda \upharpoonright l' \vdash_{\mathsf{S}} \sigma(l') : \Lambda(l')$ for every $l' \in \mathrm{def}(\sigma)$. Now $\mathrm{def}(\sigma') = \mathrm{def}(\sigma)$, so for every $l' \in \mathrm{def}(\sigma')$, $\sigma'(l') = \sigma(l')$ and $\Lambda'(l') = \Lambda(l')$. Therefore, by Lemma 69, we have $\Lambda' \upharpoonright l' \vdash_{\mathsf{S}} \sigma'(l') : \Lambda'(l')$, as required.

  (2c$'$) $\Lambda'(l) = \tau$, by definition;

  (2d$'$) $l' <_{\Lambda'} l$ implies $l' \in \mathrm{def}(\sigma')$, since $l' <_{\Lambda'} l$ implies $l' <_\Lambda l_0$ and (1d);

  (2e$'$) $\Lambda' \upharpoonright l \vdash_{\mathsf{C}} e : \tau$ since $\Lambda' \upharpoonright l = \Lambda \upharpoonright l_0$ and (1e(i)).

  Therefore, by induction,

  (2f$'$) $l \in \mathrm{def}(\sigma'')$;

  and there exists $\Lambda'' \sqsupseteq \Lambda'$ such that

  (2g$'$) $\sigma'' : \Lambda''$;

(2h′) $\Lambda', l \vdash_{\mathsf{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda''$.

Hence we have

(1f) $\Lambda'' \upharpoonright l_0 \vdash_{\mathsf{S}} l : \tau$, since $(\Lambda'' \upharpoonright l_0)(l) = \Lambda'(l) = \tau$;

(1g) $\sigma'' : \Lambda''$ by (2g′);

(1h) $\Lambda, l_0 \vdash_{\mathsf{S}} \langle \mathtt{T}_c \rangle_{l:\tau} : \tau \rightsquigarrow \Lambda''$, by (2h′).

- Suppose that

(2a) $\sigma, l_0 \leftarrow \mathtt{write}_\tau\mathtt{(}v\mathtt{)} \;\Downarrow^{\mathsf{C}}\; \sigma[l_0 \leftarrow v], \mathtt{W}_\tau$;

(2b) $\sigma : \Lambda$;

(2c) $\Lambda(l_0) = \tau$;

(2d) $l <_\Lambda l_0$ implies $l \in \mathrm{def}(\sigma)$;

(2e) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{C}} \mathtt{write}_\tau\mathtt{(}v\mathtt{)} : \tau$.

By the syntax-directed nature of the typing rules it follows that

(2e(i)) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} v : \tau$.

Let $\Lambda' = \Lambda$ and $\sigma' = \sigma[l_0 \leftarrow v]$. Then we have:

(2f) $l_0 \in \mathrm{def}(\sigma')$, by definition of σ';

(2g) $\sigma' : \Lambda'$, since $\sigma : \Lambda$ by (2b), $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} v : \tau$ by (2e(i)), and $\Lambda(l_0) = \tau$ by (2c).

(2h) $\Lambda', l_0 \vdash_{\mathsf{C}} \mathtt{W}_\tau : \tau \rightsquigarrow \Lambda'$ by definition.

- Suppose that

(2a) $\sigma, l_0 \leftarrow \mathtt{read}\, l \,\mathtt{as}\, x \,\mathtt{in}\, e \,\mathtt{end} \;\Downarrow^{\mathsf{C}}\; \sigma', R_l^{x.e}(\mathtt{T}_c)$;

(2b) $\sigma : \Lambda$;

(2c) $\Lambda(l_0) = \tau_0$;

(2d) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$;

(2e) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{C}} \mathtt{read}\, l \,\mathtt{as}\, x \,\mathtt{in}\, e \,\mathtt{end} : \tau_0$.

By the syntax-directed nature of the evaluation and typing rules, it follows that

(2a(i)) $\sigma, l_0 \leftarrow [\sigma(l)/x]\, e \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c$;

(2e(i)) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} l : \tau\,\mathtt{mod}$, hence $(\Lambda \upharpoonright l_0)(l) = \Lambda(l) = \tau$, and so $l <_\Lambda l_0$ and $\Lambda(l) = \tau$;

(2e(ii)) $\Lambda \upharpoonright l_0; x{:}\tau \vdash_{\mathsf{C}} e : \tau_0$.

Since $l <_\Lambda l_0$, it follows that $\Lambda \upharpoonright l \sqsubseteq \Lambda \upharpoonright l_0$.

Therefore,

(2a′) $\sigma, l_0 \leftarrow [\sigma(l)/x]\, e \;\Downarrow^{\mathsf{C}}\; \sigma', \mathtt{T}_c$ by (2a(i));

(2b′) $\sigma : \Lambda$ by (2b);

(2c′) $\Lambda(l_0) = \tau_0$ by (2c);

(2d′) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$ by (2d).

Furthermore, by (2b′), we have $\Lambda \upharpoonright l \vdash_{\mathsf{S}} \sigma(l) : \Lambda(l)$, hence $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} \sigma(l) : \Lambda(l)$ and so by Lemma 70 and 2e(ii),

(2e′) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{C}} [\sigma(l)/x]e : \tau_0$.

It follows by induction that

(2f′) $l_0 \in \mathrm{def}(\sigma')$

and there exists $\Lambda' \sqsupseteq \Lambda$ such that

(2g′) $\sigma' : \Lambda'$;

(2h′) $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau \rightsquigarrow \Lambda'$.

Therefore we have

(2f) $l_0 \in \mathrm{def}(\sigma')$ by (2f′);

(2g) $\sigma' : \Lambda'$ by (2g′);

(2h) $\Lambda, l_0 \vdash_{\mathsf{C}} R_l^{x.e}(\mathtt{T}_c) : \tau_0 \rightsquigarrow \Lambda'$, since

(a) $\Lambda \upharpoonright l_0, x{:}\Lambda(l) \vdash_{\mathsf{C}} e : \tau_0$ by (2e(ii));

(b) $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau_0 \rightsquigarrow \Lambda'$ by (2h′);

(c) $l \leq_\Lambda l_0$ by (2e(i)).

■

11.4.4 Type Safety for AFL

Type safety follows from the canonical forms lemma, which characterizes the shapes of closed values of each type.

Lemma 72 (Canonical Forms)
Suppose that $\Lambda \vdash_S v : \tau$. Then

- *If $\tau = \mathtt{int}$, then v is a numeric constant.*
- *If $\tau = \mathtt{bool}$, then v is either $\mathtt{true}$ or $\mathtt{false}$.*
- *If $\tau = \tau_1 \xrightarrow{C} \tau_2$, then $v = \mathtt{fun}_C\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end}$ with $\Lambda; f{:}\tau_1 \xrightarrow{C} \tau_2, x{:}\tau_1 \vdash_C e : \tau_2$.*
- *If $\tau = \tau_1 \xrightarrow{S} \tau_2$, then $v = \mathtt{fun}_S\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end}$ with $\Lambda; f{:}\tau_1 \xrightarrow{S} \tau_2, x{:}\tau_1 \vdash_S e : \tau_2$.*

- *If* $\tau = \tau'$ mod*, then* $v = l$ *for some* $l \in dom(\Lambda)$ *such that* $\Lambda(l) = \tau'$.

Theorem 73 (Type Safety)
Well-typed programs do not "go wrong".

Proof (sketch): Instrument the dynamic semantics with rules that "go wrong" in the case of a non-canonical principal argument to an elimination form. Then show that no such rule applies to a well-typed program, by appeal to type preservation and the canonical forms lemma. ■

11.5 Change Propagation

We formalize the change-propagation algorithm and the notion of an input change and prove the type safety and correctness of the change propagation algorithm.

We represent an input change via difference stores. A *difference store* is a finite map assigning values to locations. Unlike a store, a difference store may contain "dangling" locations that are not defined within the difference store. The process of modifying a store with a difference store is defined as

Definition 74 (Store Modification)
Let $\sigma : \Lambda$ *be a well-typed store for some* Λ *and let* δ *be a difference store. The modification of* σ *by* δ*, written* $\sigma \oplus \delta$*, is a well-typed store* $\sigma' : \Lambda'$ *for some* $\Lambda' \sqsupseteq \Lambda$ *and is defined as*

$$\sigma' = \sigma \oplus \delta = \delta \cup \{\, (l, \sigma(l)) \mid l \notin dom(\delta) \text{ and } l \in dom(\sigma) \,\}.$$

Note that the definition requires the result store be well typed and the types of modified locations be preserved.

Modifying a store σ with a difference store δ changes the locations in the set $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\delta)$. This set is called the *changed set* and denoted by χ. A location in the changed set is said to be *changed*.

Figure 11.17 shows the a formal version of the change-propagation algorithm that was stated in algorithmic terms in Section 11.2. In the rest of this section, the term "change-propagation algorithm" refers to the formal version. The change-propagation algorithm takes a modified store, a trace obtained by evaluating an AFL program with respect to the original store, and a changed set with respect to some difference store. The algorithm scans the trace as it seeks for reads of changed locations. When such a read is found, the body of the read is re-evaluated with the new value to update the trace and the store. Since re-evaluation can change the target of the re-evaluated read, the target is added to the changed set. Note that the change-propagation algorithm scans the trace once in the order that it was originally generated.

Formally, the change propagation algorithm is given by these two judgements:

1. *Stable propagation*: $\sigma, \mathtt{T}_s, \chi \overset{\mathtt{S}}{\curvearrowright} \sigma', \mathtt{T}'_s, \chi'$;
2. *Changeable propagation*: $\sigma, l \leftarrow \mathtt{T}_c, \chi \overset{\mathtt{C}}{\curvearrowright} \sigma', \mathtt{T}'_c, \chi'$;

$$\frac{}{\sigma, \varepsilon, \chi \stackrel{\mathtt{S}}{\curvearrowright} \sigma, \varepsilon, \chi}$$

$$\frac{\sigma, l \leftarrow \mathtt{T}_c, \chi \stackrel{\mathtt{C}}{\curvearrowright} \sigma', \mathtt{T}'_c, \chi'}{\sigma, \langle \mathtt{T}_c \rangle_{l:\tau}, \chi \stackrel{\mathtt{S}}{\curvearrowright} \sigma', \langle \mathtt{T}'_c \rangle_{l:\tau}, \chi'} \ \textbf{(mod)}$$

$$\frac{\begin{array}{ccc} \sigma, \mathtt{T}_s, \chi & \stackrel{\mathtt{S}}{\curvearrowright} & \sigma', \mathtt{T}''_s, \chi' \\ \sigma', \mathtt{T}'_s, \chi' & \stackrel{\mathtt{S}}{\curvearrowright} & \sigma'', \mathtt{T}'''_s, \chi'' \end{array}}{\sigma, (\mathtt{T}_s \ ; \ \mathtt{T}'_s), \chi \stackrel{\mathtt{S}}{\curvearrowright} \sigma'', (\mathtt{T}''_s \ ; \ \mathtt{T}'''_s), \chi''} \ \textbf{(let)}$$

$$\frac{}{\sigma, l \leftarrow \mathtt{W}_\tau, \chi \stackrel{\mathtt{S}}{\curvearrowright} \sigma, \mathtt{W}_\tau, \chi} \ \textbf{(write)}$$

$$\frac{\begin{array}{c} (l \notin \chi) \\ \sigma, l' \leftarrow \mathtt{T}_c, \chi \stackrel{\mathtt{C}}{\curvearrowright} \sigma', \mathtt{T}'_c, \chi' \end{array}}{\sigma, l' \leftarrow R_l^{x.e}(\mathtt{T}_c), \chi \stackrel{\mathtt{C}}{\curvearrowright} \sigma', R_l^{x.e}(\mathtt{T}'_c), \chi'} \ \textbf{(read/no change)}$$

$$\frac{\begin{array}{c} (l \in \chi) \\ \sigma, l' \leftarrow [\sigma(l)/x]e \ \Downarrow^{\mathtt{C}} \ \sigma', \mathtt{T}'_c \end{array}}{\sigma, l' \leftarrow R_l^{x.e}(\mathtt{T}_c), \chi \stackrel{\mathtt{C}}{\curvearrowright} \sigma', R_l^{x.e}(\mathtt{T}'_c), \chi \cup \{l'\}} \ \textbf{(read/change)}$$

$$\frac{\begin{array}{ccc} \sigma, \mathtt{T}_s, \chi & \stackrel{\mathtt{S}}{\curvearrowright} & \sigma', \mathtt{T}'_s, \chi' \\ \sigma', l' \leftarrow \mathtt{T}_c, \chi' & \stackrel{\mathtt{C}}{\curvearrowright} & \sigma'', \mathtt{T}'_c, \chi'' \end{array}}{\sigma, l' \leftarrow (\mathtt{T}_s \ ; \ \mathtt{T}_c), \chi \stackrel{\mathtt{C}}{\curvearrowright} \sigma'', (\mathtt{T}'_s \ ; \ \mathtt{T}'_c), \chi''} \ \textbf{(let)}$$

Figure 11.17: Change propagation rules (stable and changeable).

The judgements define the change-propagation for a stable trace, $\mathtt{T}_s$ (respectively, changeable trace, $\mathtt{T}_c$), with respect to a store, σ, and a changed set, $\chi \subseteq \mathrm{dom}(\sigma)$. For changeable propagation a target location, l, is maintained as in the changeable evaluation mode of AFL.

The rules defining the change-propagation judgements are given in Figure 11.17. Given a trace, change propagation mimics the evaluation rule of AFL that originally generated that trace. To stress this correspondence, each change-propagation rule is marked with the name of the evaluation rule to which it corresponds. For example, the propagation rule for the trace $\mathtt{T}_s\ ;\ \mathtt{T}'_s$ mimics the `let` rule of the stable mode that gives rise to this trace.

The most interesting rule is the `read` rule. This rule mimics a `read` operation, which evaluates an expression after binding its specified variable to the value of the location read. The read rule takes two different actions depending on whether the location being read is in the changed set or not. If the location is in the changed set, then the expression is re-evaluated with the new value of there location. Re-evaluation yields a revised store and a new trace. The new trace "repairs" the original trace by replacing the trace of the re-evaluated read. Since re-evaluating the read may change the value written at the target, the target location is added to the changed set. If the location being read is not in the changed set, then there is no need to re-evaluate this read and change-propagation continues to scan the rest of the trace.

The purely functional change-propagation algorithm presented here scans the whole trace. A direct implementation of this algorithm will therefore run in time linear in the size of the trace. The change-propagation algorithm, however, revises the trace by only replacing the changeable trace of re-evaluated reads. Thus, if one is content with updating the trace with side effects, then traces of re-evaluated reads can be replaced in place, while skipping all the rest of the trace. This is precisely what dynamic dependence graphs support (Section 11.2.4). The ML implementation performs change propagation using dynamic dependence graphs (Section 11.2.6).

11.5.1 Type Safety

The change-propagation algorithm also enjoys a type preservation property stating that if the initial state is well-formed, so is the result state. This ensures that the results of change propagation can subsequently be used as further inputs. For the preservation theorem to apply, the store modification must respect the typing of the store being modified.

Theorem 75 (Type Preservation)
Suppose that def$(\sigma) =$ *dom*(σ).

1. *If*

   (a) $\sigma, T_s, \chi \overset{S}{\curvearrowright} \sigma', T'_s, \chi'$,

   (b) $\sigma : \Lambda$,

   (c) $l_0 \in dom(\Lambda)$,

   (d) $\Lambda, l_0 \vdash_S T_s\ \mathsf{ok}$, *and*

   (e) $\chi \subseteq dom(\Lambda)$,

   then for some $\Lambda' \sqsupseteq \Lambda$,

   (f) $\sigma' : \Lambda'$,

   (g) $\Lambda, l_0 \vdash_S T'_s \leadsto \Lambda'$,

(h) $\chi' \subseteq dom(\Lambda)$.

2. *If*

   (a) $\sigma, l_0 \leftarrow T_c, \chi \overset{\mathsf{C}}{\curvearrowright} \sigma', T'_c, \chi'$,
   (b) $\sigma : \Lambda$,
   (c) $\Lambda(l_0) = \tau_0$,
   (d) $\Lambda, l_0 \vdash_{\mathsf{C}} T_c : \tau_0$, *and*
   (e) $\chi \subseteq dom(\Lambda)$,

   then there exists $\Lambda' \sqsupseteq \Lambda$ *such that*

   (f) $\sigma' : \Lambda'$,
   (g) $\Lambda, l_0 \vdash_{\mathsf{C}} T'_c : \tau_0 \rightsquigarrow \Lambda'$, *and*
   (h) $\chi' \subseteq dom(\Lambda)$.

Proof: By induction on the definition of the change propagation relations, making use of Theorem 71. We consider the case of a re-evaluation of a read. Suppose that $l \in \chi$ and

(2a) $\sigma, l_0 \leftarrow R_l^{x.e}(\mathtt{T}_c), \chi \overset{\mathsf{C}}{\curvearrowright} \sigma', R_l^{x.e}(\mathtt{T}'_c), \chi \cup \{l_0\}$;

(2b) $\sigma : \Lambda$;

(2c) $\Lambda(l_0) = \tau_0$;

(2d) $\Lambda, l_0 \vdash_{\mathsf{C}} R_l^{x.e}(\mathtt{T}_c) : \tau_0$;

(2e) $\chi \subseteq \mathrm{dom}(\Lambda)$.

By the syntax-directed nature of the change propagation and trace typing rules, it follows that

(2a(i)) $\sigma, l_0 \leftarrow [\sigma(l)/x]e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}'_c$;

(2b(i) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{S}} \sigma(l) : \Lambda(l)$, by (2b);

(2d(i)) $l <_{\Lambda} l_0$ and $\Lambda(l) = \tau$ for some type τ;

(2d(ii)) $\Lambda \upharpoonright l_0; x{:}\tau \vdash_{\mathsf{C}} e : \tau_0$;

(2d(iii)) $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c : \tau_0$.

Therefore

(2a′) $\sigma, l_0 \leftarrow [\sigma(l)/x]e \Downarrow^{\mathsf{C}} \sigma', \mathtt{T}'_c$ by (2a(i));

(2b′) $\sigma : \Lambda$ by (2b);

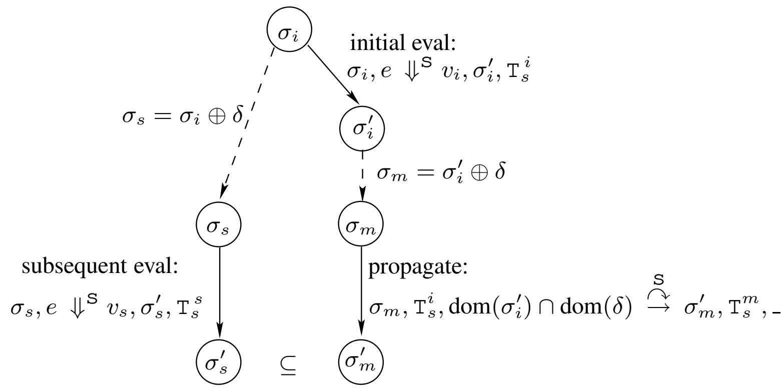


Figure 11.18: Change propagation simulates a complete re-evaluation.

(2c′) $\Lambda(l_0) = \tau_0$ by (2c);

(2d′) $l' <_\Lambda l_0$ implies $l' \in \mathrm{def}(\sigma)$ by assumption that $\mathrm{def}(\sigma) = \mathrm{dom}(\sigma)$;

(2e′) $\Lambda \upharpoonright l_0 \vdash_{\mathsf{C}} [\sigma(l)/x]e : \tau_0$ by (2d(ii)), (2b(i)), and Lemma 70.

Hence, by Theorem 71,

(2f′) $l \in \mathrm{def}(\sigma')$;

and there exists $\Lambda' \sqsupseteq \Lambda$ such that

(2g′) $\sigma' : \Lambda'$;

(2h′) $\Lambda, l_0 \vdash_{\mathsf{C}} \mathtt{T}_c' : \tau_0 \leadsto \Lambda'$.

Consequently,

(2f) $\sigma' : \Lambda'$ by (2g′);

(2g) $\Lambda, l_0 \vdash_{\mathsf{C}} R_l^{x.e}(\mathtt{T}_c') : \tau_0$ by (2d(i) and (ii)), (2h′), and Lemma 69;

(2h) $C \cup \{ l_0 \} \subseteq \mathrm{dom}(\Lambda')$ since $l_0 \in \mathrm{dom}(\Lambda)$ and $\Lambda' \sqsupseteq \Lambda$.

■

11.5.2 Correctness

Change propagation simulates a complete re-evaluation by re-evaluating only the affected sub-expressions of an AFL program. This sections shows that change propagation is correct by proving that it yields the same output and the trace as a complete re-evaluation.

Consider evaluating an adaptive program (a stable expression) e with respect to an initial store σ_i; call this the *initial evaluation*. As shown in Figure 11.18, the initial evaluation yields a value v_i, an extended store σ_i', and a trace $\mathtt{T}_s^i$. Now modify the initial store with a difference store δ as $\sigma_s = \sigma_i \oplus \delta$ and re-evaluate the program with this store in a *subsequent evaluation*. To simulate the subsequent evaluation via a change propagation apply the modifications δ to σ_i' and let σ_m denote the modified store, *i.e.*, $\sigma_m = \sigma_i' \oplus \delta$. Now perform change propagation with respect to σ_m, the trace of the initial evaluation $\mathtt{T}_s^i$, and the set of changed locations $\mathrm{dom}(\sigma_i') \cap \mathrm{dom}(\delta)$. Change propagation yields a revised store σ_m' and trace $\mathtt{T}_s^m$. For the change-propagation to work properly, δ must change only input locations, i.e., $\mathrm{dom}(\sigma_i') \cap \mathrm{dom}(\delta) \subseteq \mathrm{dom}(\sigma_i)$.

To prove correctness, we compare the store and the trace obtained by the subsequent evaluation, σ_s' and $\mathtt{T}_s^s$, to those obtained by the change propagation, σ_m' and $\mathtt{T}_s^m$, (see Figure 11.18). Since locations (names) are chosen arbitrarily during evaluation, subsequent evaluation and change propagation can choose different locations (names). We therefore show that the traces are identical modulo the choice of locations and the the store σ_s' is contained in the store σ_m' modulo the choice of locations.

To study relations between traces and stores modulo the choice of locations we use an equivalence relation for stores and traces that matches different locations via a partial bijection. A *partial bijection* is a one-to-one mapping from a set of locations D to a set of locations R that may not map all the locations in D.

Definition 76 (Partial Bijection)
B is a partial bijection from set D to set R if it satisfies the following:

1. $B \subseteq \{\, (a,b) \mid a \in D, b \in R \,\}$,
2. *if* $(a,b) \in B$ *and* $(a,b') \in B$ *then* $b = b'$,
3. *if* $(a,b) \in B$ *and* $(a',b) \in B$ *then* $a = a'$.

The value of a location l under the partial bijection B is denoted by $B(l)$.

A partial bijection, B, can be applied to an expression e, to a store σ, or to a trace $\mathtt{T}$, denoted $B[e]$, $B[\sigma]$, and $B[\mathtt{T}]$, by replacing, whenever defined, each location l with $B(l)$:

Definition 77 (Applications of Partial Bijections)

Expression: *The application of a partial bijection B to an expression e yields another expression obtained by substituting each location l in e with $B(l)$ (when defined) as shown in Figure 11.19.*

Hole: *The application of a partial bijection to a hole yields a hole, $B[\square] = \square$.*

Store: *The application of a partial bijection B to a store σ yields another store $B[\sigma]$, defined as $B[\sigma] = \{\, (B[l], B[\sigma(l)]) \mid l \in \mathrm{dom}(\sigma) \,\}$.*

Trace: *The application of a partial bijection to a trace is defined as*

$$
\begin{array}{rcl}
B[\epsilon] & = & \epsilon \\
B[\langle \mathtt{T}_c \rangle_{l:\tau}] & = & \langle B[\mathtt{T}_c] \rangle_{B[l]:\tau} \\
B[\mathtt{T}_s \,;\, \mathtt{T}_s] & = & B[\mathtt{T}_s] \,;\, B[\mathtt{T}_s] \\
B[\mathtt{W}_\tau] & = & \mathtt{W}_\tau \\
B[R_l^{x.e}(\mathtt{T}_c)] & = & R_{B[l]}^{B[x].B[e]}(B[\mathtt{T}_c]) \\
B[\mathtt{T}_s \,;\, \mathtt{T}_c] & = & B[\mathtt{T}_s] \,;\, B[\mathtt{T}_c].
\end{array}
$$

$$
\begin{array}{rcl}
 & B[c] = c, \quad B[x] = x & \\
B[l] & = & \begin{cases} l' & \mathit{if}\ (l,l') \in B \\ l & \mathit{otherwise} \end{cases} \\
B[\texttt{fun}_\texttt{S}\ f(x:\tau):\tau'\ \texttt{is}\ e\ \texttt{end}] & = & \texttt{fun}_\texttt{S}\ f(x:\tau):\tau'\ \texttt{is}\ B[e_s]\ \texttt{end} \\
B[\texttt{fun}_\texttt{C}\ f(x:\tau):\tau'\ \texttt{is}\ e\ \texttt{end}] & = & \texttt{fun}_\texttt{C}\ f(x:\tau):\tau'\ \texttt{is}\ B[e_c]\ \texttt{end}
\end{array}
$$

$$
\begin{array}{rcl}
B[o\,\texttt{(}v_1,\ldots,v_n\texttt{)}] & = & o\,\texttt{(}B[v_1],\ldots,B[v_n]\texttt{)} \\
B[\texttt{apply}_\texttt{S}\,\texttt{(}v_1,v_2\texttt{)}] & = & \texttt{apply}_\texttt{S}\,\texttt{(}B[v_1],B[v_2]\texttt{)} \\
B[\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e'_s\ \texttt{end}] & = & \texttt{let}\ x\ \texttt{be}\ B[e_s]\ \texttt{in}\ B[e'_s]\ \texttt{end} \\
B[\texttt{if}\ v\ \texttt{then}\ e_s\ \texttt{else}\ e'_s] & = & \texttt{if}\ B[v]\ \texttt{then}\ B[e_s]\ \texttt{else}\ B[e'_s] \\
B[\texttt{mod}_\tau\ e_c] & = & \texttt{mod}_\tau\ B[e_c] \\
 & & \\
B[\texttt{apply}_\texttt{C}\,\texttt{(}v_1,v_2\texttt{)}] & = & \texttt{apply}_\texttt{C}\,\texttt{(}B[v_1],B[v_2]\texttt{)} \\
B[\texttt{let}\ x\ \texttt{be}\ e_s\ \texttt{in}\ e_c\ \texttt{end}] & = & \texttt{let}\ x\ \texttt{be}\ B[e_s]\ \texttt{in}\ B[e_c]\ \texttt{end} \\
B[\texttt{if}\ v\ \texttt{then}\ e_c\ \texttt{else}\ e'_c] & = & \texttt{if}\ B[v]\ \texttt{then}\ B[e_c]\ \texttt{else}\ B[e'_c] \\
B[\texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e_c\ \texttt{end}] & = & \texttt{read}\ B[v]\ \texttt{as}\ x\ \texttt{in}\ B[e_c]\ \texttt{end} \\
B[\texttt{write}_\tau\,\texttt{(}v\texttt{)}] & = & \texttt{write}_\tau\,\texttt{(}B[v]\texttt{)}
\end{array}
$$

Figure 11.19: Application of a partial bijection B to values, and stable and changeable expression.

Destination: *Application of a partial bijection to an expression with a destination is defined as* $B[l \leftarrow e] = B[l] \leftarrow B[e]$.

In the correctness theorem, we will show that the traces obtained by change propagation and the subsequent evaluation are identical under some partial bijection B; referring back to Figure 11.18, we will show that $B[\mathtt{T}_s^m] = \mathtt{T}_s^s$. The relationship between the store σ'_s of the subsequent evaluation and σ'_m of change propagation are more subtle. Since the change propagation is performed on the store that the initial evaluation yields σ'_i, and no allocated location is ever deleted, the store after the change propagation will contain leftover but now unused (garbage) locations from the initial evaluation. We will therefore show that $B[\sigma'_m]$ contains σ'_s.

Definition 78 (Store Containment)
We say that a store, σ, is contained *in another σ', written $\sigma \sqsubseteq \sigma'$, if*

1. $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$, and
2. $\forall l, l \in \mathrm{def}(\sigma), \sigma(l) = \sigma'(l)$.

We now state and prove the correctness theorem. The theorem hinges on a lemma (Lemma 81) for expressions that are equal up to a partial bijection—such expressions arise due to substitution of a variable by two different locations. The correctness theorem, however, is only concerned with *equal* programs or stable expressions, *i.e.*, expressions that are syntactically identical. We note that the equivalence relation between expressions does not "deference" locations: two expressions are equal even though the stores can contain different values for the same location.

Theorem 79 (Correctness)
Let e be a well-typed program with respect to a store typing Λ, $\sigma_i : \Lambda$ be an initial store such that def$(\sigma_i) =$ dom(σ_i), δ be a difference store, $\sigma_s = \sigma_i \oplus \delta$, and $\sigma_m = \sigma'_i \oplus \delta$ as shown in Figure 11.18. If

(A1) $\sigma_i, e \Downarrow^{S} v_i, \sigma'_i, \mathtt{T}^i_s,$ *(initial evaluation)*

(A2) $\sigma_s, e \Downarrow^{S} v_s, \sigma'_s, \mathtt{T}^s_s,$ *(subsequent evaluation)*

(A3) dom$(\sigma'_i) \cap$ dom$(\delta) \subseteq$ dom(σ_i)

then the following holds:

1. $\sigma_m, \mathtt{T}^i_s, (\text{dom}(\sigma'_i) \cap \text{dom}(\delta)) \overset{\overset{S}{\curvearrowright}}{\rightarrow} \sigma'_m, \mathtt{T}^m_s, \_,$
2. *there is a partial bijection $\mathcal{B}$ such that*
   (a) $\mathcal{B}[v_i] = v_s,$
   (b) $\mathcal{B}[\mathtt{T}^m_s] = \mathtt{T}^s_s,$
   (c) $\mathcal{B}[\sigma'_m] \sqsupseteq \sigma'_s.$

Proof: The proof is by an application of Lemma 81. To apply the lemma, define the partial bijection B (of Lemma 81) to be the identity function with domain dom(σ_i). The following hold:

1. dom$(\sigma_m) \supseteq$ dom(σ'_i) (follows by the definition of the store modification).
2. $\forall l, l \in (\text{def}(\sigma'_i) - \text{def}(\sigma_i)), \sigma_m(l) = \sigma'_i(l)$ (follows by assumption three (A3) and the definition of the store modification).
3. $B[\sigma_m] \sqsupseteq \sigma_s$ (since B is the identity, this reduces to $\sigma_m \sqsupseteq \sigma_s$, which holds because $\sigma_s = \sigma_i \oplus \delta$, and $\sigma_m = \sigma'_i \oplus \delta$ and $\sigma'_i \sqsupseteq \sigma_i$).

Applying Lemma 81 with changed locations dom$(\sigma'_i) \cap$ dom$(\delta) = \{l \mid l \in \text{dom}(\sigma_i) \wedge \sigma_i[l] \neq \sigma_s[l]\}$ yields a partial bijection B' such that

1. $B'[v_i] = v_s,$
2. $B'[\mathtt{T}^m_s] = \mathtt{T}^s_s,$
3. $B'[\sigma'_m] \sqsupseteq \sigma'_s.$

Thus taking $\mathcal{B} = B'$ proves the theorem.

■

We turn our attention to the main lemma. Lemma 81 considers initial and subsequent evaluations of expressions that are equivalent under some partial bijection and shows that the subsequent evaluation is identical to change propagation under an extended partial bijection. The need to consider expressions that are equivalent under some partial bijection arises because arbitrarily chosen locations (names) can be substituted for the same variable in two different evaluations. We start by defining the notion of a changed set, the set of changed locations of a store, with respect to some modified store and a partial bijection.

Definition 80 (Changed Set)
Given two stores σ and σ', and a partial bijection B from dom(σ) to the dom(σ') the set of changed locations *of σ is*

$$\chi\,(B, \sigma, \sigma')\, = \{\, l \mid l \in \mathit{dom}(\sigma), B[\sigma(l)] \neq \sigma'(B[l])\,\}.$$

The main lemma consists of two symmetric parts for stable and changeable expressions. For each kind of expression, it shows that the trace and the store of a subsequent evaluation of an expression under some partial bijection is identical to those that would have obtained by change propagation under some extended partial bijection. The lemma constructs a partial bijection by mapping locations created by change propagation to those created by the subsequent evaluation. We will assume, without loss of generality that the expression are well-typed with respect to the stores that they are evaluated with. Indeed, the correctness theorem (Theorem 79) requires that the expressions and store modifications be well typed.

The proof assumes that a changeable expression evaluates to a different value when re-evaluated, *i.e.*, the value of the destination location changes. This assumption causes no loss of generality, and can be eliminated by additional machinery to enable comparison of the old and the new values of the destination location.

Lemma 81 (Change Propagation)
Consider the stores σ_i and σ_s and B be a partial bijection from dom(σ_i) to dom(σ_s). The following hold:

- *If*

$$\sigma_i, l \leftarrow e \Downarrow^{C} \sigma_i', \mathcal{T}_c^i; \quad \text{and}$$
$$\sigma_s, B[l \leftarrow e] \Downarrow^{C} \sigma_s', \mathcal{T}_c^s$$

  then for any store σ_m satisfying

  1. *dom$(\sigma_m) \supseteq$ dom(σ_i'),*
  2. *$\forall l, l \in (\mathit{def}(\sigma_i') - \mathit{def}(\sigma_i)), \sigma_m(l) = \sigma_i'(l)$, and*
  3. *$B[\sigma_m] \sqsupseteq \sigma_s$,*

  there exists a partial bijection B' such that

$$\sigma_m, l \leftarrow \mathcal{T}_c^i, \chi\,(B, \sigma_i, \sigma_s)\, \overset{C}{\curvearrowright} \sigma_m', \mathcal{T}_c^m, \chi; \quad \text{where}$$

  1. *$B' \supseteq B$*
  2. *dom(B') = dom$(B) \cup$ def$(\mathcal{T}_c^m)$,*
  3. *$B'[\sigma_m'] \sqsupseteq \sigma_s'$,*
  4. *$B'[\mathcal{T}_c^m] = \mathcal{T}_c^s$, and*
  5. *$\chi = \chi\,(B', \sigma_i', \sigma_s')$.*

- *If*

$$\sigma_i, e \Downarrow^{S} v_i, \sigma_i', \mathcal{T}_s^i; \quad \text{and}$$
$$\sigma_s, B[e] \Downarrow^{S} v_i', \sigma_s', \mathcal{T}_s^s,$$

  then for any store σ_m satisfying

1. *dom*$(\sigma_m) \supseteq$ *dom*(σ'_i),
2. $\forall l, l \in ($*def*$(\sigma'_i) -$ *def*$(\sigma_i)), \sigma_m(l) = \sigma'_i(l)$, *and*
3. $B[\sigma_m] \sqsupseteq \sigma_s$,

there exists a partial bijection B' *such that*

$$\sigma_m, \mathtt{T}^i_s, \chi\,\mathtt{(}B, \sigma_i, \sigma_s\mathtt{)} \overset{\mathtt{S}}{\curvearrowright} \sigma'_m, \mathtt{T}^m_s, \chi; \quad \textit{where}$$

1. $B' \supseteq B$,
2. *dom*$(B') =$ *dom*$(B) \cup$ *def*$(\mathtt{T}^m_s)$,
3. $B'[v_i] = v'_i$,
4. $B'[\sigma'_m] \sqsupseteq \sigma'_s$,
5. $B'[\mathtt{T}^m_s] = \mathtt{T}^s_s$, *and*
6. $\chi = \chi\,\mathtt{(}B', \sigma'_i, \sigma'_s\mathtt{)}$.

Proof: The proof is by simultaneous induction on the evaluation. Among the changeable expressions, the most interesting are `write`, `let`, and `read`. Among the stable expression, the most interesting are the `let` and `mod`.

We refer to the following properties of modified store σ_m as the *modified-store properties*:

1. $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma'_i)$,
2. $\forall l, l \in (\mathrm{def}(\sigma'_i) - \mathrm{def}(\sigma_i)), \sigma_m(l) = \sigma'_i(l)$, and
3. $B[\sigma_m] \sqsupseteq \sigma_s$,

- **Write:** Suppose

$$\sigma_i, l \leftarrow \mathtt{write}_\tau\mathtt{(}v\mathtt{)} \Downarrow^{\mathtt{C}} \sigma_i[l \to v], \mathtt{W}_\tau; \quad \text{and}$$
$$\sigma_s, B[l \leftarrow \mathtt{write}_\tau\mathtt{(}v\mathtt{)}] \Downarrow^{\mathtt{C}} \sigma_s[B[l] \to B[v]], \mathtt{W}_\tau$$

then for store σ_m satisfying the modified-store properties we have

$$\sigma_m, l \leftarrow \mathtt{W}_\tau, \chi\,\mathtt{(}B, \sigma_i, \sigma_s\mathtt{)} \overset{\mathtt{C}}{\curvearrowright} \sigma_m, \mathtt{W}_\tau, \chi\,\mathtt{(}B, \sigma_i, \sigma_s\mathtt{)}.$$

The partial bijection B satisfies the following properties:

1. $B \supseteq B$
2. $\mathrm{dom}(B) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{W}_\tau)$
3. $B[\sigma_m] \sqsupseteq \sigma_s[B[l] \to B[v]]$: We know that $B[\sigma_m] \sqsupseteq \sigma_s$ and thus we must show that $B[l]$ is mapped to $B(v)$ in $B[\sigma'_m]$. Observe that $\sigma_m(l) = (\sigma_i[l \to v])(l) = v$ by Modified-Store Property 2, thus $B[\sigma_m](B[l]) = B[v]$.
4. $B[\mathtt{W}_\tau] = \mathtt{W}_\tau$

5. $\chi\,(B, \sigma_i, \sigma_s) = \chi\,(B, \sigma_i[l \to v], \sigma_s[B[l] \to B[v]])$, by definition.

Thus, pick $B' = B$ for this case.

- **Apply (Changeable):** Suppose that

$$\frac{\text{(I.1)}\quad \sigma_i, l \leftarrow [v/x, \mathtt{fun_C}\; f(x:\tau_1):\tau_2\; \mathtt{is}\; e\; \mathtt{end}/f]\; e\; \Downarrow^{\mathsf{C}}\; \sigma_i', \mathtt{T}_c^i}{\text{(I.2)}\quad \sigma_i, l \leftarrow \mathtt{apply_C}\,(\mathtt{fun_C}\; f(x:\tau_1):\tau_2\; \mathtt{is}\; e\; \mathtt{end}, v)\; \Downarrow^{\mathsf{C}}\; \sigma_i', \mathtt{T}_c^i}$$

$$\frac{\text{(S.1)}\quad \sigma_s, B[l] \leftarrow [B[v]/x, B[\mathtt{fun_C}\; f(x:\tau_1):\tau_2\; \mathtt{is}\; e\; \mathtt{end}]/f]\; e]\; \Downarrow^{\mathsf{C}}\; \sigma_s', \mathtt{T}_c^s}{\text{(S.2)}\quad \sigma_s, B[l \leftarrow \mathtt{apply_C}\,(\mathtt{fun_C}\; f(x:\tau_1):\tau_2\; \mathtt{is}\; e\; \mathtt{end}, v)]\; \Downarrow^{\mathsf{C}}\; \sigma_s', \mathtt{T}_c^s}$$

Consider evaluations (I.1) and (S.1) and a store σ_m that satisfies the modified-store properties. By induction we have a partial bijection B_0 and

$$\sigma_m, l \leftarrow \mathtt{T}_c^i, \chi\,(B, \sigma_i, \sigma_s) \overset{\mathsf{C}}{\curvearrowright} \sigma_m, \mathtt{T}_c^m, \chi,$$

where

1. $B_0 \supseteq B$,
2. $\mathrm{dom}(B_0) = \mathrm{dom}(B) \cup \mathrm{def}(\mathtt{T}_c^m)$,
3. $B_0[\mathtt{T}_c^m] = \mathtt{T}_c^s$, and
4. $B_0[\sigma_m] \sqsupseteq \sigma_s'$.
5. $\chi = \chi\,(B_0, \sigma_i', \sigma_s')$,

Since (I.2) and (S.2) return the trace and store returned by (I.1) and (S.1), we pick $B' = B_0$ for this case.

- **Let:**

$$\frac{\begin{array}{lll}\text{(I.1)} & \sigma_i, e & \Downarrow^{\mathsf{S}} \quad v_i, \sigma_i', \mathtt{T}_s^i \\ \text{(I.2)} & \sigma_i', l \leftarrow [v_i/x]e' & \Downarrow^{\mathsf{C}} \quad \sigma_i'', \mathtt{T}_c^i\end{array}}{\text{(I.3)}\quad \sigma_i, l \leftarrow \mathtt{let}\; x\; \mathtt{be}\; e\; \mathtt{in}\; e'\; \mathtt{end}\; \Downarrow^{\mathsf{C}}\; \sigma_i'', (\mathtt{T}_s^i\; ;\; \mathtt{T}_c^i)}$$

$$\frac{\begin{array}{lll}\text{(S.1)} & \sigma_s, B[e] & \Downarrow^{\mathsf{S}} \quad v_s, \sigma_s', \mathtt{T}_s^s \\ \text{(S.2)} & \sigma_s', B[l] \leftarrow [v_s/x]B[e'] & \Downarrow^{\mathsf{C}} \quad \sigma_s'', \mathtt{T}_c^s\end{array}}{\text{(S.3)}\quad \sigma_s, B[l \leftarrow \mathtt{let}\; x\; \mathtt{be}\; e\; \mathtt{in}\; e'\; \mathtt{end}]\; \Downarrow^{\mathsf{C}}\; \sigma_s'', (\mathtt{T}_s^s\; ;\; \mathtt{T}_c^s)}$$

Consider any store σ_m that satisfies the modified-store properties. The following judgement shows a change propagation applied with the store σ_m on the output trace $\mathtt{T}_s^i\; ;\; \mathtt{T}_c^i$.

$$\frac{\begin{array}{lll}(\text{P.1}) & \sigma_m, \mathtt{T}_s^i, \chi\,(B, \sigma_i, \sigma_s) & \overset{\mathtt{S}}{\curvearrowright}\!\!\to\ \sigma'_m, \mathtt{T}_s^m, \chi \\ (\text{P.2}) & \sigma'_m, l \leftarrow \mathtt{T}_c^i, \chi & \overset{\mathtt{C}}{\curvearrowright}\!\!\to\ \sigma''_m, \mathtt{T}_c^m, \chi'\end{array}}{(\text{P.3}) \quad \sigma_m, l \leftarrow (\mathtt{T}_s^i \,;\, \mathtt{T}_c^i), \chi\,(B, \sigma_i, \sigma_s) \overset{\mathtt{C}}{\curvearrowright}\!\!\to \sigma''_m, (\mathtt{T}_s^m \,;\, \mathtt{T}_c^m), \chi'}$$

We apply the induction hypothesis on (I.1) (S.1) and (P.1) to obtain a partial bijection B_0 such that

1. $B_0 \supseteq B$,
2. $\text{dom}(B_0) = \text{dom}(B) \cup \text{def}(\mathtt{T}_s^m)$,
3. $v_s = B_0[v_i]$,
4. $B_0[\sigma'_m] \sqsupseteq \sigma'_s$,
5. $B_0[\mathtt{T}_s^m] = \mathtt{T}_s^s$, and
6. $\chi = \chi\,(B_0, \sigma'_i, \sigma'_s)$.

Using these properties, we now show that we can apply the induction hypothesis on (I.2) and (S.2) with the partial bijection B_0.

- $B_0[l \leftarrow [v_i/x]e'] = B[l] \leftarrow [v_s/x]B[e']$:
  By Properties 1 and 2 it follows that $B[l] = B_0[l]$.
  By Property 3, $B_0[v_i] = v_s$.
  To show that $B[e'] = B_0[e']$, note that $\text{locs}(e) \subseteq \text{dom}(\sigma_i) \subseteq \text{dom}(\sigma_m)$ (since e is well typed with respect to σ_i). It follows that $\text{dom}(\mathtt{T}_s^m) \cap \text{locs}(e) = \emptyset$. Since $\text{dom}(B_0) = \text{dom}(B) \cup \text{def}(\mathtt{T}_s^m)$, $B[e'] = B_0[e']$.
- $\chi = \chi\,(B_0, \sigma'_i, \sigma''_i)$. This is true by Property 6.
- σ'_m satisfies the modified-store properties:
  1. $\text{dom}(\sigma'_m) \supseteq \text{dom}(\sigma'_i)$
     This is true because $\text{dom}(\sigma'_m) \supseteq \text{dom}(\sigma_m) \supseteq \text{dom}(\sigma''_i) \supseteq \text{dom}(\sigma'_i)$.
  2. $\forall l, l \in (\text{def}(\sigma''_i) - \text{def}(\sigma'_i)), \sigma'_m(l) = \sigma''_i(l)$
     To show that $\forall l, l \in (\text{def}(\sigma''_i) - \text{def}(\sigma'_i)), \sigma'_m(l) = \sigma''_i(l)$, observe that
     (a) $\forall l, l \in (\text{def}(\sigma''_i) - \text{def}(\sigma_i)), \sigma_m(l) = \sigma''_i(l)$,
     (b) $\text{def}(\sigma''_i) - \text{def}(\sigma'_i) = \text{def}(\mathtt{T}_c^i) \cup \{l\}$,
     (c) $\text{def}(\mathtt{T}_s^i) \cap (\text{def}(\sigma''_i) - \text{def}(\sigma'_i)) = \emptyset$,
     and that the evaluation (P.1) changes values of locations only in $\text{def}(\mathtt{T}_s^i)$.
  3. $B_0[\sigma'_m] \sqsupseteq \sigma'_s$, this follows by Property 4.

Now, we can apply the induction hypothesis on (I.2) (S.2) to obtain a partial bijection B_1 such that

1′. $B_1 \supseteq B_0$,

2′. $\text{dom}(B_1) = \text{dom}(B_0) \cup \text{def}(\mathtt{T}_c^m)$,

3′. $B_1[\sigma''_m] \sqsupseteq \sigma''_s$,

4′. $B_1[\mathtt{T}_c^m] = \mathtt{T}_c$, and

5′. $\chi' = \chi\,(B_1, \sigma''_i, \sigma''_s)$.

Based on these, we have

1″. $B_1 \supseteq B$.
This holds because $B_1 \supseteq B_0 \supseteq B$.

2″. $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_s^m\ ;\ \mathtt{T}_c^m)$.
We know that $\text{dom}(B_1) = \text{dom}(B_0) \cup \text{def}(\mathtt{T}_c^m)$ and $\text{dom}(B_0) = \text{dom}(B) \cup \text{def}(\mathtt{T}_s^m)$. Thus we have $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_s^m) \cup \text{def}(\mathtt{T}_c^m) = \text{dom}(B) \cup \text{def}(\mathtt{T}_s^m\ ;\ \mathtt{T}_c^m)$.

3″. $B_1[\sigma''_m] \sqsupseteq \sigma''_s$.
This follows by Property 3′.

4″. $B_1[\mathtt{T}_s^m\ ;\ \mathtt{T}_c^m] = \mathtt{T}_s^s\ ;\ \mathtt{T}_c^s$.
This holds if and only if $B_1[\mathtt{T}_s^m] = \mathtt{T}_s^s$ and $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$.
We know that $B_0[\mathtt{T}_s^m] = \mathtt{T}_s^s$ and since $\text{dom}(B_1) = \text{dom}(B_0) \cup \text{def}(\mathtt{T}_c^m)$ and $\text{def}(\mathtt{T}_s^m) \cap \text{def}(\mathtt{T}_c^m) = \emptyset$, we have $B_1[\mathtt{T}_s^m] = \mathtt{T}_s^s$. We also know that $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$ by Property 4′.

5″. $\chi' = \chi\,(B_1, \sigma''_i, \sigma''_s)$,
This follows by Property 5′.

Thus we pick $B' = B_1$.

- **Read:** Assume that we have:

$$\frac{\text{(I.1)} \qquad \sigma_i, l' \leftarrow [\sigma_i(l)/x]e \ \Downarrow^{\mathsf{C}}\ \sigma'_i, \mathtt{T}_c^i}{\text{(I.2)} \qquad \sigma_i, l' \leftarrow \mathtt{read}\, l\, \mathtt{as}\, x\, \mathtt{in}\, e\, \mathtt{end} \ \Downarrow^{\mathsf{C}}\ \sigma'_i, R_l^{x.e}(\mathtt{T}_c^i)}$$

$$\frac{\text{(S.1)} \qquad \sigma_s, B[l'] \leftarrow [\sigma_s(B[l])/x]B[e] \ \Downarrow^{\mathsf{C}}\ \sigma'_s, \mathtt{T}_c^s}{\text{(S.2)} \quad \sigma_s, B[l' \leftarrow \mathtt{read}\, l\, \mathtt{as}\, x\, \mathtt{in}\, e\, \mathtt{end}] \ \Downarrow^{\mathsf{C}}\ \sigma'_s, R_{B[l]}^{x.B[e]}(\mathtt{T}_c^s)}$$

Consider a store σ_m that satisfies the modified-store properties. Then we have two cases for the corresponding change-propagation evaluation. In the first case $l \notin \chi$ and we have:

$$\frac{\text{(P.1)} \qquad \sigma_m, l' \leftarrow \mathtt{T}_c^i, \chi\,(B, \sigma_i, \sigma_s) \ \overset{\mathsf{C}}{\curvearrowright}\ \sigma'_m, \mathtt{T}_c^m, \chi}{\text{(P.2)} \quad \sigma_m, l' \leftarrow R_l^{x.e}(\mathtt{T}_c^i), \chi\,(B, \sigma_i, \sigma_s) \ \overset{\mathsf{C}}{\curvearrowright}\ \sigma'_m, R_l^{x.e}(\mathtt{T}_c^m), \chi} \ (l \notin \chi)$$

In this case, we apply the induction hypothesis on (I.1), (S.1), and (P.1) with the partial bijection B to obtain a partial bijection B_0 such that

1. $B_0 \supseteq B$,
2. $\text{dom}(B_0) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$,
3. $B_0[\sigma'_m] \sqsupseteq \sigma'_s$,
4. $B_0[\mathtt{T}_c^m] = \mathtt{T}_c^s$, and
5. $\chi = \chi\,(B_0, \sigma'_i, \sigma'_s)$.

Furthermore, the following hold for B_0,

1. $\text{dom}(B_0) = \text{dom}(B) \cup \text{def}(R_l^{x.e}(\mathtt{T}_c^m))$.
   This follows by Property 2 and because $\text{def}(R_l^{x.e}(\mathtt{T}_c^m)) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$,
2. $B_0[R_l^{x.e}(\mathtt{T}_c^m)] = R_{B[l]}^{x.B[e]}(\mathtt{T}_c^s)$.
   We have $B_0[R_l^{x.e}(\mathtt{T}_c^m)] = R_{B_0[l]}^{x.B_0[e]}(B_0[\mathtt{T}_c^m]) = R_{B_0[l]}^{x.B_0[e]}(\mathtt{T}_c^s)$, because of (c). Thus we need to show that $B_0[l] = B[l]$ and $B_0[e] = B[e]$. This is true because,
   (a) $l \notin \text{def}(\mathtt{T}_c^m)$ and thus $B[l] = B_0[l]$, and
   (b) $\forall l, l \in \text{locs}(e)$ we have $l \in \text{dom}(\sigma_m)$ and thus $l \notin \text{def}(\mathtt{T}_c^m)$, which implies that $B[l] = B_0[l]$, and $B[e] = B_0[e]$.

Thus we pick $B' = B_0$.

In the second case, we have $l \in \chi$ and the read $R_l^{x.e}$ is re-evaluated.

$$\frac{\text{(P.4)} \qquad \sigma_m, l' \leftarrow [\sigma_m(l)/x]e \;\Downarrow^{\mathsf{C}}\; \sigma'_m, \mathtt{T}_c^m}{\text{(P.5)} \qquad \sigma_m, l' \leftarrow R_l^{x.e}(\mathtt{T}_c), \chi\,(B, \sigma_i, \sigma_s) \;\overset{\mathsf{C}}{\curvearrowright}\; \sigma'_m, R_l^{x.e}(\mathtt{T}_c^m), \chi\,(B, \sigma_i, \sigma_s) \cup \{l'\}} \;(l \in \chi)$$

Since $B[\sigma_m] \sqsupseteq \sigma_s$, the evaluation in (P.4) is identical to the evaluation in (S.1) and thus, there is a bijection $B_1 \supseteq B$ such that $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$ and $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$. Thus we have

1. $B_1 \supseteq B$,
2. $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$,
3. $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$,
4. $\chi\,(B, \sigma_i, \sigma_s) \cup \{l\} = \chi\,(B_1, \sigma'_i, \sigma'_s)$
   To show this, observe that
   (a) $\text{dom}(\sigma'_i) \cap \text{def}(\mathtt{T}_c^m) = \emptyset$, because $\text{dom}(\sigma_m) \supseteq \text{dom}(\sigma'_i)$.
   (b) $\chi\,(B_1, \sigma'_i, \sigma'_s) = \chi\,(B, \sigma'_i, \sigma'_s)$, because $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\mathtt{T}_c^m)$.
   (c) $\chi\,(B, \sigma'_i, \sigma'_s) = \chi\,(B, \sigma_i, \sigma_s) \cup \{l'\}$, because $\text{dom}(B) \subseteq \text{dom}(\sigma_i)$. (Assuming, without loss of generality, that the value of l' changes because of the re-evaluation).
5. $B_1[\sigma'_m] \sqsupseteq \sigma'_s$
   We know that $B_1[\sigma_m] \sqsupseteq \sigma_s$. Furthermore $B_1[\sigma'_m - \sigma_m] = \sigma'_s - \sigma_s$ and thus, $B_1[\sigma'_m] \sqsupseteq \sigma'_s$.

Thus pick $B' = B_1$.

- **Value:** Suppose that

$$\sigma_i, v \Downarrow^{\mathsf{S}} v, \sigma_i, \varepsilon$$
$$\sigma_s, B[v] \Downarrow^{\mathsf{S}} B[v], \sigma_s, \varepsilon.$$

Let σ_m be any store that satisfies the modified-store properties. We have

$$\sigma_m, \varepsilon, \chi\,(B, \sigma_i, \sigma_s) \overset{\mathsf{S}}{\curvearrowright} \sigma_m, \varepsilon, \chi\,(B, \sigma_i, \sigma_s)$$

where

1. $B \supseteq B$.
2. $\mathrm{dom}(B) = \mathrm{dom}(B) \cup \mathrm{def}(\varepsilon)$.
3. $B[\sigma_m] \sqsupseteq \sigma_s$, by Modified-Store Property 3.
4. $B[\varepsilon] = \varepsilon$.
5. $\chi\,(B, \sigma_i, \sigma_s) = \chi\,(B, \sigma_i, \sigma_s)$.

Thus pick $B' = B$.

- **Apply (Stable):** This is similar to the apply in the changeable mode.

- **Mod:** Suppose that

$$\begin{array}{ll} \text{(I.1)} & \sigma_i[l_i \to \square], l_i \leftarrow e \Downarrow^{\mathsf{C}} \sigma'_i, \mathtt{T}_c^{\,i} \\ \hline \text{(I.2)} & \sigma_i, \mathtt{mod}_\tau\ e \Downarrow^{\mathsf{S}} l_i, \sigma'_i, \langle \mathtt{T}_c^{\,i} \rangle_{l_i:\tau} \end{array} \; l_i \not\in \mathrm{dom}(\sigma_i)$$

$$\begin{array}{ll} \text{(S.1)} & \sigma_s[l_s \to \square], l_s \leftarrow B[e] \Downarrow^{\mathsf{C}} \sigma'_s, \mathtt{T}_c^{\,s} \\ \hline \text{(S.2)} & \sigma_s, B[\mathtt{mod}_\tau\ e] \Downarrow^{\mathsf{S}} l_s, \sigma'_s, \langle \mathtt{T}_c^{\,s} \rangle_{l_s:\tau} \end{array} \; l_s \not\in \mathrm{dom}(\sigma_s).$$

Let σ_m be a store that satisfies the modified-store properties. Then we have

$$\begin{array}{ll} \text{(P.1)} & \sigma_m, l_i \leftarrow \mathtt{T}_c^{\,i}, \chi\,(B, \sigma_i, \sigma_s) \overset{\mathsf{C}}{\curvearrowright} \sigma'_m, \mathtt{T}_c^{\,m}, \chi \\ \hline \text{(P.2)} & \sigma_m, \langle \mathtt{T}_c^{\,i} \rangle_{l_i:\tau}, \chi\,(B, \sigma_i, \sigma_s) \overset{\mathsf{S}}{\curvearrowright} \sigma'_m, \langle \mathtt{T}_c^{\,m} \rangle_{l_i:\tau}, \chi \end{array}$$

Consider the partial bijection $B_0 = B[l_i \mapsto l_s]$. It satisfies the following:

- $B_0[l_i \leftarrow e] = l_s \leftarrow B[e]$.
  Because $B_0(l_i) = l_s$ and $l_i \not\in \mathrm{locs}(e)$.
- $B_0[\sigma_m] \sqsupseteq \sigma_s[l_s \mapsto \square]$.
  We know that $B[\sigma_m] \sqsupseteq \sigma_s$ by Modified-Store Property 3. Since $l_s \not\in \mathrm{dom}(\sigma_s)$, we have $B_0[\sigma_m] \sqsupseteq \sigma_s$.
  Furthermore $l_s = B_0(l_i)$ and $l_i \in \mathrm{dom}(\sigma_m)$ because $\mathrm{dom}(\sigma_m) \supseteq \mathrm{dom}(\sigma'_i)$.
  Thus $B_0[\sigma_m] \sqsupseteq \sigma_s[l_s \mapsto \square]]$.

– $\forall l, l \in (\text{def}(\sigma_i') - \text{def}(\sigma_i[l_i \to \Box])), \sigma_m(l) = \sigma_i'(l)$.
Because $\forall l, l \in (\text{def}(\sigma_i') - \text{def}(\sigma_i)), \sigma_m(l) = \sigma_i'(l)$ by Modified-Store Property 2.

Thus, we can apply the induction hypothesis on (I.1), (S.1) with the partial bijection $B_0 = B[l_i \mapsto l_s]$ to obtain a partial bijection B_1 such that the following hold.

1. $B_1 \supseteq B_0$,
2. $\text{dom}(B_1) = \text{dom}(B_0) \cup \text{def}(\mathtt{T}_c^i)$,
3. $B_1[\sigma_m'] \sqsupseteq \sigma_s'$,
4. $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$, and
5. $\chi = \chi\,(B_1, \sigma_s', \sigma_i')$.

Furthermore, B_1 satisfies

1. $\text{dom}(B_1) = \text{dom}(B) \cup \text{def}(\langle \mathtt{T}_c^i \rangle_{l_i:\tau})$.
By Property 2 and because $\text{def}(\mathtt{T}_c^i) = \text{def}(\langle \mathtt{T}_c^i \rangle_{l_i:\tau})$.

2. $B_1[\langle \mathtt{T}_c^m \rangle_{l_i:\tau}] = \langle \mathtt{T}_c^s \rangle_{l_s:\tau}$.
Because $B_1[\mathtt{T}_c^m] = \mathtt{T}_c^s$ by Property 4 and $B_1(l_i) = l_s$.

Thus we can pick $B' = B_1$.

- **Let (Stable):** This is similar to the let rule in changeable mode.

■

Chapter 12

Selective Memoization

This chapter presents a framework for applying memoization selectively. The framework provides programmer control over equality, space usage, and identification of precise dependences so that memoization can be applied according to the needs of an application. Two key properties of the framework are that it is efficient and yields programs whose performance can be analyzed using standard techniques.

We describe the framework in the context of a functional language and an implementation as an SML library. The language is based on a modal type system and allows the programmer to express programs that reveal their true data dependences when executed. The SML implementation cannot support this modal type system statically, but instead employs run-time checks to ensure correct usage of primitives.

12.1 Introduction

Memoization is a fundamental and powerful technique for result re-use. It dates back a half century [15, 60, 61] and has been used extensively in many areas such as dynamic programming [8, 22, 24, 58], incremental computation [29, 83, 35, 94, 51, 1, 100, 57, 47, 3], and many others [18, 65, 52, 72, 57]. In fact, lazy evaluation provides a limited form of memoization [53].

Although memoization can dramatically improve performance and can require only small changes to the code, no language or library support for memoization has gained broad acceptance. Instead, many successful uses of memoization rely on application-specific support code. The underlying reason for this is one of control: since memoization is all about performance, the user must be able to control the performance of memoization. Many subtleties of memoization, including the cost of equality checking and the cache replacement policy for memo tables, can make the difference between exponential and linear running time.

To be general and widely applicable a memoization framework must provide control over these three areas: (1) the kind and cost of equality tests; (2) the identification of precise dependences between the input and the output of memoized code; and (3) space management. Control over equality tests is critical, because this is how re-usable results are identified. Control over identification of precise dependences is important to maximize result reuse. Being able to control when memo tables or individual entries are purged is critical, because otherwise the user will not know whether or when results are re-used.

In this chapter, we propose a framework for memoization that provides control over equality and identification of dependences, and some control over space management. We study the framework in the context of a small language called MFL and provide an implementation for it in the Standard ML language. We also prove the type safety and correctness of MFL—*i.e.,* that the semantics are preserved with respect to a non-memoized version. As an example, we show how to analyze the performance of a memoized version of Quicksort within our framework.

In the next section we describe background and related work. In Section 12.3 we introduce our framework via some examples. In Section 12.4 we formalize the MFL language and discuss its safety, correctness, and performance properties. In Section 12.5 we present a simple implementation of the framework as a Standard ML library. In Section 12.6 we discuss how the framework might be extended to allow for better control of space usage, and discuss the relationship of this work to our previous work on adaptive computation [3].

12.2 Background and Related Work

A typical memoization scheme maintains a memo table mapping argument values to previously computed results. This table is consulted before each function call to determine if the particular argument is in the table. If so, the call is skipped and the result is returned; otherwise the call is performed and its result is added to the table. The semantics and implementation of the memo lookup are critical to performance. Here we review some key issues in implementing memoization efficiently.

Equality. Any memoization scheme needs to search a memo table for a match to the current arguments. Such a search will, at minimum, require a test for equality. Typically it will also require some form of hashing. In standard language implementations testing for equality on structures, for example, can require traversing the whole structure. The cost of such an equality test can negate the advantage of memoizing and may even change the asymptotic behavior of the function. A few approaches have been proposed to alleviate this problem. The first is based on the fact that for memoization equality need not be exact—it can return unequal when two arguments are actually equal. The implementation could therefore decide to skip the test if the equality is too expensive, or could use a conservative equality test, such as "location" equality. The problem with such approaches is that whether a match is found could depend on particulars of the implementation and will surely not be evident to the programmer.

Another approach for reducing the cost of equality tests is to ensure that there is only one copy of every value, via a technique known as "hash consing" [41, 9, 93]. If there is only one copy, then equality can be implemented by comparing locations. In fact, the location can also be used as a key to a hash table. In theory, the overhead of hash-consing is constant in the expected case (expectation is over internal randomization of hash functions). The reality, however, is rather different because of large memory demands of hash-consing and its interaction with garbage collection. In fact, several researchers have argued that hash-consing is too expensive for practical purposes [81, 81, 12, 71]. As an alternative to hash consing, Pugh proposed lazy structure sharing [81]. In lazy structure sharing whenever two equal values are compared, they are made to point to the same copy to speed up subsequent comparisons. As Pugh points out, the disadvantage of this approach is that the performance depends on the order comparisons and thus it is difficult to analyze.

We note that even with hash-consing, or any other method, it remains critical to define equality on all types including reals and functions. Claiming that functions are never equivalent, for example, is not satisfactory because the result of a call involving some function as a parameter will never be re-used.

Precise Dependences. To maximize result re-use, the result of a function call must be stored with respect to its true dependences. This issue arises when the function examines only parts or an approximation of its parameter. To enable "partial" equality checks, the unexamined parts of the parameter should be disregarded. To increase the likelihood of result re-use, one should be able to match on the approximation, rather than the parameter itself. As an example, consider the code

```
fun f(x,y,z) = if (x > 0) then fy(y) else fz(z)
```

The result of `f` depends on either `(x,y)` or `(x,z)`. Also, it depends on an approximation of `x`–whether or not it is positive–rather than its exact value. Thus, the memo entry `(7,11,20)` should match `(7,11,30)` or `(4,11,50)` since, when `x` is positive, the result depends only on `y`.

Several researchers have remarked that partial matching can be very important in some applications [77, 76, 1, 47]. Abadi, Lampson, Lévy [1], and Heydon, Levin, Yu [47] have suggested program analysis methods for tracking dependences for this purpose. Although their technique is likely effective in catching potential matches, it does not provide a programmer controlled mechanism for specifying what dependences should be tracked. Also, their program analysis technique can change the asymptotic performance of a program, making it difficult to asses the effects of memoization.

Space management. Another problem with memoization is its space requirement. As a program executes, its memo tables can become large limiting the utility of memoization. To alleviate this problem, memo tables or individual entries should be disposed of under programmer control.

In some application, such as in dynamic programming, most result re-use occurs among the recursive calls of some function. Thus, the memo table of such a function can be disposed of whenever it terminates. In other applications, where result re-use is less structured, individual memo table entries should be purged according to a replacement policy [48, 81]. The problem is to determine what exact replacement policy should be used and to analyze the performance effects of the chosen policy. One widely used approach is to replace the least recently used entry. Other, more sophisticated, policies have also been suggested [81]. In general the replacement policy must be application-specific, because, for any fixed policy, there are programs whose performance is made worse by that choice [81].

12.3 A Framework for Selective Memoization

We present an overview of our framework via some examples. The framework extends a purely functional language with several constructs to support selective memoization. In this section, we use an extension to an ML-like language for the discussion. We formalize the core of this language and study its safety, soundness, and performance properties in Section 12.4.

The framework enables the programmer to determine precisely the dependences between the input and the result of a function. The main idea is to deem the parameters of a function as *resources* and provide primitives to explore incrementally any value, including the underlying value of a resource. This incremental exploration process reveals the dependences between the parameter of the function and its result.

The incremental exploration process is guided by types. If a value has the modal type $!\,\tau$, then the underlying value of type τ can be bound to an ordinary, unrestricted variable by the `let!` construct; this will create a dependence between the underlying value and the result. If a value has a product type, then its

<table>
<tr><th>Non-memoized</th><th>Memoized</th></tr>
<tr><td><pre>
fib:int -> int
fun fib (n)=

  if (n < 2) then n
  else fib(n-1) + fib(n-2)
</pre></td><td><pre>
mfib:!int -> int
mfun mfib (n')=
  let !n = n' in return (
    if (n < 2) then n
    else mfib(!(n-1)) + mfib(!(n-2))) end
</pre></td></tr>
<tr><td><pre>
f: int * int * int -> int
fun f (x, y, z)=
  if (x > 0) then
    fy y
  else
    fz z
</pre></td><td><pre>
mf:int * !int * !int -> int
mfun mf (x', y', z')=
  mif (x' > 0) then
    let !y = y' in return (fy y) end
  else
    let !z = z' in return (fz z) end
</pre></td></tr>
</table>

Figure 12.1: Fibonacci and expressing partial dependences.

two parts can be bound to two resources using the `let*` construct; this creates no dependences. If the value is a sum type, then it can be case analyzed using the `mcase` construct, which branches according to the outermost form of the value and assigns the inner value to a resource; `mcase` creates a dependence on the outer form of the value of the resource. The key aspect of the `let*` and `mcase` is that they bind resources rather than ordinary variables.

Exploring the input to a function via `let!`, `mcase`, and `let*` builds a *branch* recording the dependences between the input and the result of the function. The `let!` adds to the branch the full value, the `mcase` adds the kind of the sum, and `let*` adds nothing. Consequently, a branch contains both data dependences (from `let!`'s) and control dependences (from `mcase`'s). When a `return` is encountered, the branch recording the revealed dependences is used to key the memo table. If the result is found in the memo table, then the stored value is returned, otherwise the body of the `return` is evaluated and the memo table is updated to map the branch to the result. The type system ensures that all dependences are made explicit by precluding the use of resources within `return`'s body.

As an example consider the Fibonacci function `fib` and its memoized counterpart `mfib` shown in Figure 12.1. The memoized version, `mfib`, exposes the underlying value of its parameter, a resource, before performing the two recursive calls as usual. Since the result depends on the full value of the parameter, it has a bang type. The memoized Fibonacci function runs in linear time as opposed to exponential time when not memoized.

Partial dependences between the input and the result of a function can be captured by using the incremental exploration technique. As an example consider the function `f` shown in Figure 12.1. The function checks whether `x` is positive or not and returns `fy(y)` or `fz(z)`. Thus the result of the function depends on an approximation of `x` (its sign) and on either `y` or `z`. The memoized version `mf` captures this by first checking if `x'` is positive or not and then exposing the underlying value of `y'` or `z'` accordingly. Consequently, the result will depend on the sign of `x'` and on either `y'` or `z'`. Thus if `mf` is called with parameters $(1, 5, 7)$ first and then $(2, 5, 3)$, the result will be found in the memo the second time, because when `x'` is positive the result depends only on `y'`. Note that `mif` construct used in this example is just a special case of the more general `mcase` construct.

A critical issue for efficient memoization is the implementation of memo tables along with lookup and update operations on them. In our framework we support expected constant time memo table lookup and update operations by representing memo tables using hashing. To do this, we require that the underlying type τ of a modal type $!\tau$ be an *indexable type*. An indexable type is associated with an injective function, called an *index function*, that maps each value of that type to a unique integer; this integer is called the *index* of the value. The uniqueness property of the indices for a given type ensures that two values are equal if and only if their indices are equal. In our framework, equality is only defined for indexable types. This enables us to implement memo tables as hash tables keyed by branches consisting of indices.

We assume that each primitive type comes with an index function. For examples, for integers, the identity function can be chosen as the index function. Composite types such as lists or functions must be *boxed* to obtain an indexable type. A boxed value of type τ has type τ `box`. When a box is created, it is assigned a unique locations (or tag), and this location is used as the unique index of that boxed value. For example, we can define boxed lists as follows.

```
datatype α blist' =  NIL | CONS of α * ((α blist') box)
type α blist = (α blist') box
```

Based on boxes we implement hash-consing as a form of memoization. For example, hash-consing for boxed lists can be implemented as follows.

```
hCons:  !α * !(α blist) -> α blist
mfun hCons (h', t') =
  let !h = h' and !t = t' in
    return (box (CONS(h,t))) end
  end
```

The function takes an item and a boxed list and returns the boxed list formed by consing them. Since the function is memoized, if it is ever called with two values that are already hash-consed, then the same result will be returned. The advantage of being able to define hash-consing as a memoized function is that it can be applied selectively.

To control space usage of memo tables, our framework gives the programmer a way to dispose of memo tables by conventional scoping. In our framework, each memoized function is allocated its own memo table. Thus, when the function goes out of scope, its memo table can be garbage collected. For example, in many dynamic-programming algorithms result re-use occurs between recursive calls of the same function. In this case, the programmer can scope the memoized function inside an auxiliary function so that its memo table is discarded as soon as the auxiliary function returns. As an example, consider the standard algorithm for the Knapsack Problem `ks` and its memoized version `mks` Figure 12.2. Since result sharing mostly occurs among the recursive calls of `mks`, it can be scoped in some other function that calls `mks`; once `mks` returns its memo table will go out of scope and can be discarded.

We note that this technique gives only partial control over space usage. In particular it does not give control over when individual memo table entries are purged. In Section 12.6, we discuss how the framework might be extended so that each memo table is managed according to a programmer specified caching scheme. The basic idea is to require the programmer to supply a caching scheme as a parameter to the `mfun` and maintain the memo table according to the chosen caching scheme.

<table>
<tr><th>Non-memoized</th><th>Memoized</th></tr>
<tr><td><pre>
ks:int*((int*real) list)->int
fun ks (c,l) =

  case l of
    nil => 0
   |(w,v)::t =>
    if (c < w) then
      ks(c,t)
    else
      let v1 = ks(c,t)
          v2 = v + ks(c-w,t)
      in
        if (v1>v2) then v1
        else v2
      end
</pre></td><td><pre>
mks:!int*!((int*real) list)->int
mfun mks (c',l')
  let !c = c' and !l = l' in return (
    case (unbox l) of
      NIL => 0
    | CONS((w,v),t) =>
      if (c < w) then
        mks(!c,!t)
      else
        let v1 = mks(!c,!t)
            v2 = v + mks(!(c-w),!t)
        in
          if (v1 > v2) then v1
          else v2
        end) end
</pre></td></tr>
</table>

Figure 12.2: Memo tables for memoized Knapsack can be discarded at completion.

Memoized Quicksort. As a more sophisticated example, we consider Quicksort. Figure 12.3 shows an implementation of the Quicksort algorithm and its memoized counterpart. The algorithm first divides its input into two lists containing the keys less than the pivot, and greater than the pivot by using the filter function `fil`. It then sorts the two sublists, and returns the concatenation of the results. The memoized filter function `mfil` uses hash-consing to ensure that there is only one copy of each result list. The memoized Quicksort algorithm `mqs` exposes the underlying value of its parameter and is otherwise similar to `qs`. Note that `mqs` does not build its result via hash-consing—it can output two copies of the same result. Since in this example the output of `mqs` is not consumed by any other function, there is no need to do so. Even if the result were consumed by some other function, one can choose not to use hash-consing because operations such as insertions to and deletions from the input list will surely change the result of Quicksort.

When the memoized Quicksort algorithm is called on "similar" inputs, one would expect that some of the results would be re-used. Indeed, we show that the memoized Quicksort algorithm computes its result in expected linear time when its input is obtained from a previous input by inserting a new key at the beginning. Here the expectation is over all permutations of the input list and also the internal randomization of the hash functions used to implement the memo tables. For the analysis, we assume, without loss of generality, that all keys in the list are unique.

Theorem 82
Let L be a list and let $L' = [a, L]$. Consider running memoized Quicksort on L and then on L'. The running time of Quicksort on the modified list L' is expected $O(n)$ where n is the length of L'.

Proof: Consider the recursion tree of Quicksort with input L, denoted $Q(L)$, and label each node with the pivot of the corresponding recursive call (see Figure 12.4 for an example). Consider any pivot (key) p from L and let L_p denote the keys that precede p in L. It is easy to see that a key k is in the subtree rooted at p if and only if the following two properties are satisfied for any key $k' \in L_p$.

Non-memoized

```
fil:int->bool*int list->int list
fun fil (g:int->bool, l:int list) =
  case l of
    nil => nil
   | h::t =>
    let tt = fil(g,t) in
      if (g h) then h::tt
      else tt
    end

qs:int list->int list
fun qs (l) =

  case l of
    nil => nil
   | cons(h,t) =>
    let s = fil(fn x=>x<h,t)
        g = fil(fn x=>x>=h,t)
    in
      (qs s)@(h::(qs g))
    end
```

Memoized

```
empty = box NIL

mfil:int->bool*int blist->int blist
fun mfil (g,l) =
  case (unbox l) of
    NIL => empty
   | CONS(h,t) =>
    let tt = mfil(g,t) in
      if (g h) then hCons(h,tt)
      else tt
    end

mqs:  !(int blist)->int blist
mfun mqs (l':!int blist) =
  let !l = l' in return (
    case (unbox l) of
      NIL => NIL
     | CONS(h,t) =>
      let s = mfil(fn x=>x<h,t)
          g = mfil(fn x=>x>=h,t)
      in
        (mqs !s)@(h::(mqs !g))
      end) end
```

Figure 12.3: The Quicksort algorithm.

1. If $k' < p$ then $k > k'$, and
2. if $k' > p$ then $k < k'$.

Of the keys that are in the subtree of p, those that are less than p are in its left subtree and those greater than p are in its right subtree.

Now consider the recursion tree $Q(L')$ for $L' = [a, L]$ and let p be any pivot in $Q(L')$. Suppose $p < a$ and let k be any key in the left subtree of p in $Q(L)$. Since $k < p$, by the two properties k is in the left subtree of p in $Q(L')$. Similarly if $p > a$ then any k in the right subtree of p in $Q(L)$ is also in the right subtree of p in $Q(L')$. Since filtering preserves the respective order of keys in the input list, for any $p, p < a$, the input to the recursive call corresponding to its left child will be the same. Similarly, for $p > a$, the input to the recursive call corresponding to its right child will be the same. Thus, when sorting L' these recursive calls will find their results in the memo. Therefore only recursive calls corresponding to the root, to the children of the nodes in the rightmost spine of the left subtree of the root, and the children of the nodes in the leftmost spine of the right subtree of the root may be executed (the two spines are shown with thick lines in Figure 12.4). Furthermore, the results for the calls adjacent to the spines will be found in the memo.

Consider the calls whose results are not found in the memo. In the worst case, these will be all the calls along the two spines. Consider the sizes of inputs for the nodes on a spine and define the random variables $X_1 \ldots X_k$ such that X_i is the least number of recursive calls (nodes) performed for the input size to become

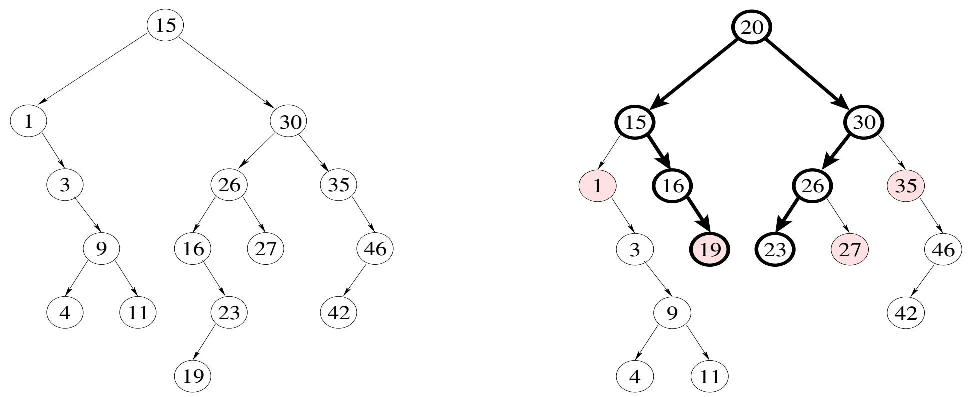


Figure 12.4: Quicksort's recursion tree with inputs $L = [15, 30, 26, 1, 3, 16, 27, 9, 35, 4, 46, 23, 11, 42, 19]$ (left) and $L' = [20, L]$ (right).

$\left(\frac{3}{4}\right)^i n$ or less after it first becomes $\left(\frac{3}{4}\right)^{(i-1)} n$ or less. Since $k \leq \lceil \log_{4/3} n \rceil$, the total and the expected number of operations along a spine are

$$\begin{aligned} C(n) &\leq \sum_{i=1}^{\lceil \log_{4/3} n \rceil} X_i \left(\frac{3}{4}\right)^{i-1} n, \text{ and} \\ E[C(n)] &\leq \sum_{i=1}^{\lceil \log_{4/3} n \rceil} E[X_i] \left(\frac{3}{4}\right)^{i-1} n. \end{aligned}$$

Since the probability that the pivot lies in the middle half of the list is $\frac{1}{2}$, $E[X_i] \leq 2$ for $i \geq 1$, and we have

$$E[C(n)] \leq \sum_{i=1}^{\lceil \log_{4/3} n \rceil} 2 \left(\frac{3}{4}\right)^{i-1} n.$$

Thus, $E[C(n)] = O(n)$ This bound holds for both spines; therefore the number of operations due to calls whose results are not found in the memo is $O(n)$. Since each operation, including hash-consing, takes expected constant time, the total time of the calls whose results are not in the memo is $O(n)$. Now, consider the calls whose results are found in the memo, each such call will be on a spine or adjacent to it, thus there are an expected $O(\log n)$ such calls. Since, the memo table lookup overhead is expected constant time the total cost for these is $O(\log n)$. We conclude that Quicksort will take expected $O(n)$ time for sorting the modified list L'. ■

It is easy to extend the theorem to show that the $O(n)$ bound holds for an insertion anywhere in the list. Although, this bound is better than a complete rerun, which would take $O(n \log n)$, we would like to achieve $O(\log n)$. In Chapter 13 we describe a combination of memoization and adaptivity reduces the expected cost of a random insertion to $O(\log n)$.

Indexable Types	η	$::=$	$1 \mid \mathtt{int} \mid \ldots$
Types	τ	$::=$	$\eta \mid\ !\,\eta \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu u.\tau \mid \tau_1 \rightarrow \tau_2$
Operators	o	$::=$	$+ \mid - \mid \ldots$
Expressions	e	$::=$	$\mathtt{return}\,(t) \mid \mathtt{let}\ !\,x{:}\eta\,\mathtt{be}\,t\,\mathtt{in}\,e\,\mathtt{end} \mid$ $\mathtt{let}\,a_1{:}\tau_1{\times}a_2{:}\tau_2\,\mathtt{be}\,t\,\mathtt{in}\,e\,\mathtt{end} \mid$ $\mathtt{mcase}\,t\,\mathtt{of}\,\mathtt{inl}\ (a_1{:}\tau_1)\ \Rightarrow\ e_1 \mid \mathtt{inr}\ (a_2{:}\tau_2)\ \Rightarrow\ e_2\,\mathtt{end}$
Terms	t	$::=$	$v \mid o\,(t_1, \ldots, t_n) \mid \langle t_1, t_2 \rangle \mid \mathtt{mfun}\,f\ (a{:}\tau_1){:}\tau_2\,\mathtt{is}\,e\,\mathtt{end} \mid$ $t_1\,t_2 \mid\ !\,t \mid \mathtt{inl}_{\tau_1+\tau_2}t \mid \mathtt{inr}_{\tau_1+\tau_2}t \mid \mathtt{roll}\,(t) \mid \mathtt{unroll}\,(t)$
Values	v	$::=$	$x \mid a \mid \star \mid n \mid\ !\,v \mid \langle v_1, v_2 \rangle \mid \mathtt{mfun}_l\,f\,(a{:}\tau_1){:}\tau_2\,\mathtt{is}\,e\,\mathtt{end}$

Figure 12.5: The abstract syntax of MFL.

12.4 The MFL Language

In this section we study a small functional language, called MFL, that supports selective memoization. MFL distinguishes memoized from non-memoized code, and is equipped with a modality for tracking dependences on data structures within memoized code. This modality is central to our approach to selective memoization, and is the focus of our attention here. The main result is a soundness theorem stating that memoization does not affect the outcome of a computation compared to a standard, non-memoizing semantics. We also show that the memoization mechanism of MFL causes a constant factor slowdown compared to a standard, non-memoizing semantics.

12.4.1 Abstract Syntax

The abstract syntax of MFL is given in Figure 12.5. The meta-variables x and y range over a countable set of *variables*. The meta-variables a and b range overf a countable set of *resources*. (The distinction will be made clear below.) The meta-variable l ranges over a countable set of *locations*. We assume that variables, resources, and locations are mutually disjoint. The binding and scope conventions for variables and resources are as would be expected from the syntactic forms. As usual we identify pieces of syntax that differ only in their choice of bound variable or resource names. A term or expression is *resource-free* if and only if it contains no free resources, and is *variable-free* if and only if it contains no free variables. A *closed* term or expression is both resource-free and variable-free; otherwise it is *open*.

The types of MFL include 1 (unit), `int`, products and sums, recursive data types $\mu u.\tau$, memoized function types, and bang types $!\,\eta$. MFL distinguishes *indexable types*, denoted η, as those that accept an injective function, called an *index function*, whose co-domain is integers. The underlying type of a bang type $!\,\eta$ is restricted to be an indexable type. For `int` type, identity serves as an index function; for 1 (unit) any constant function can be chosen as the index function. For non-primitive types an index can be supplied by boxing values of these types. Boxed values would be allocated in a store and the unique location of a box would serve as an index for the underlying value. With this extension the indexable types would be defined as $\eta ::= 1 \mid \texttt{int} \mid \tau\,\texttt{box}$. Although supporting boxed types is critical for practical purposes, we do not formalize this here to focus on the main ideas.

The syntax is structured into *terms* and *expressions*, in the terminology of Pfenning and Davies [79]. Roughly speaking, terms evaluate independently of their context, as in ordinary functional programming, whereas expressions are evaluated relative to a memo table. Thus, the body of a memoized function is an expression, whereas the function itself is a term. Note, however, that the application of a function is a *term*, not an *expression*; this corresponds to the encapsulation of memoization with the function, so that updating the memo table is benign. In a more complete language we would include case analysis and projection forms among the terms, but for the sake of simplicity we include these only as expressions. We would also include a plain function for which the body is a term. Note that every term is trivially an expression; the `return` expression is the inclusion.

12.4.2 Static Semantics

The type structure of MFL extends the framework of Pfenning and Davies [79] with a "necessitation" modality, $!\,\eta$, which is used to track data dependences for selective memoization. This modality does not correspond to a monadic interpretation of memoization effects ($\bigcirc\,\tau$ in the notation of Pfenning and Davies), though one could imagine adding such a modality to the language. The introductory and eliminatory forms for necessity are standard, namely $!\,t$ for introduction, and $\texttt{let}\ !\,x\!:\!\eta\ \texttt{be}\ t\ \texttt{in}\ e\ \texttt{end}$ for elimination.

Our modality demands that we distinguish variables from resources. Variables in MFL correspond to the "validity", or "unrestricted", context in modal logic, whereas resources in MFL correspond to the "truth", or "restricted" context. An analogy may also be made to the judgmental presentation of linear logic [78, 80]: variables correspond to the intuitionistic context, resources to the linear context.[1]

The inclusion, $\texttt{return}(t)$, of terms into expressions has no analogue in pure modal logic, but is specific to our interpretation of memoization as a computational effect. The typing rule for $\texttt{return}(t)$ requires that t be resource-free to ensure that any dependence on the argument to a memoized function is made explicit in the code before computing the return value of the function. In the first instance, resources arise as parameters to memoized functions, with further resources introduced by their incremental decomposition using $\texttt{let}\times$ and `mcase`. These additional resources track the usage of as-yet-unexplored parts of a data structure. Ultimately, the complete value of a resource may be accessed using the `let!` construct, which binds its value to a variable, which may be used without restriction. In practice this means that those parts of an argument to a memoized function on whose value the function depends will be given modal type. However, it is not essential that all resources have modal type, nor that the computation depend upon every resource that does have modal type.

The static semantics of MFL consists of a set of rules for deriving typing judgments of the form $\Gamma; \Delta \vdash t : \tau$, for terms, and $\Gamma; \Delta \vdash e : \tau$, for expressions. In these judgments Γ is a *variable type assignment*, a finite function assigning types to variables, and Δ is a *resource type assignment*, a finite function assigning types to resources. The rules for deriving these judgments are given in Figures 12.6 and 12.7.

[1]Note, however, that we impose no linearity constraints in our type system!

$$\frac{(\Gamma(x)=\tau)}{\Gamma;\Delta\vdash x{:}\tau}\ \textbf{(variable)} \qquad \frac{(\Delta(a)=\tau)}{\Gamma;\Delta\vdash a{:}\tau}\ \textbf{(resource)}$$

$$\frac{}{\Gamma;\Delta\vdash n:\texttt{int}}\ \textbf{(number)} \qquad \frac{}{\Gamma;\Delta\vdash \star:1}\ \textbf{(unit)}$$

$$\frac{\Gamma;\Delta\vdash t_i:\tau_i\ \ (1\leq i\leq n) \qquad \vdash_o o:(\tau_1,\ldots,\tau_n)\ \tau}{\Gamma;\Delta\vdash o\,\texttt{(}t_1,\ldots,t_n\texttt{)}\ :\tau}\ \textbf{(primitive)}$$

$$\frac{\Gamma;\Delta\vdash t_1:\tau_1 \quad \Gamma;\Delta\vdash t_2:\tau_2}{\Gamma;\Delta\vdash\langle t_1,t_2\rangle:\tau_1\times\tau_2}\ \textbf{(pair)}$$

$$\frac{\Gamma,\ f{:}\tau_1\rightarrow\tau_2;\Delta,a{:}\tau_1\vdash e:\tau_2}{\Gamma;\Delta\vdash\texttt{mfun}\,f\ \texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}:\tau_1\rightarrow\tau_2}\ \textbf{(fun)}$$

$$\frac{\Gamma,f{:}\tau_1\rightarrow\tau_2;\Delta,a{:}\tau_1\vdash e:\tau_2}{\Gamma;\Delta\vdash\texttt{mfun}_l\,f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}:\tau_1\rightarrow\tau_2}\ \textbf{(fun value)}$$

$$\frac{\Gamma;\Delta\vdash t_1:\tau_1\rightarrow\tau_2 \quad \Gamma;\Delta\vdash t_2:\tau_1}{\Gamma;\Delta\vdash t_1\,t_2:\tau_2}\ \textbf{(apply)}$$

$$\frac{\Gamma;\emptyset\vdash t:\eta}{\Gamma;\Delta\vdash\ \texttt{!}\,t:\texttt{!}\,\eta}\ \textbf{(bang)}$$

$$\frac{\Gamma;\Delta\vdash t:\tau_1}{\Gamma;\Delta\vdash\texttt{inl}_{\tau_1+\tau_2}t:\tau_1+\tau_2}\ \textbf{(sum/inl)} \qquad \frac{\Gamma;\Delta\vdash t:\tau_2}{\Gamma;\Delta\vdash\texttt{inr}_{\tau_1+\tau_2}t:\tau_1+\tau_2}\ \textbf{(sum/inr)}$$

$$\frac{\Gamma;\Delta\vdash t:[\mu u.\tau/u]\tau}{\Gamma;\Delta\vdash\texttt{roll(}t\texttt{)}\ :\mu u.\tau}\ \textbf{(roll)} \qquad \frac{\Gamma;\Delta\vdash t:\mu u.\tau}{\Delta\vdash\texttt{unroll(}t\texttt{)}\ :[\mu u.\tau/u]\tau}\ \textbf{(unroll)}$$

Figure 12.6: Typing judgments for terms.

12.4.3 Dynamic Semantics

The dynamic semantics of MFL formalizes selective memoization. Evaluation is parameterized by a store containing memo tables that track the behavior of functions in the program. Evaluation of a function expression causes an empty memo table to be allocated and associated with that function. Application of a memoized function is affected by, and may affect, its associated memo table. Should the function value become inaccessible, so also is its associated memo table, and hence the storage required for both can be reclaimed.

Unlike conventional memoization, however, the memo table is keyed by control flow information rather than by the values of arguments to memoized functions. This is the key to supporting selective memoization. Expression evaluation is essentially an exploration of the available resources culminating in a resource-free term that determines its value. Since the exploration is data-sensitive, only certain aspects of the resources may be relevant to a particular outcome. For example, a memoized function may take a pair of integers as

$$\frac{\Gamma; \emptyset \vdash t : \tau}{\Gamma; \Delta \vdash \texttt{return}\,(t)\ : \tau}\ \textbf{(return)}$$

$$\frac{\Gamma; \Delta \vdash t : \,!\,\eta \quad \Gamma, x{:}\eta; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \texttt{let}\ !\,x{:}\eta\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} : \tau}\ \textbf{(let!)}$$

$$\frac{\Gamma; \Delta \vdash t : \tau_1 \times \tau_2 \quad \Gamma; \Delta, a_1{:}\tau_1, a_2{:}\tau_2 \vdash e : \tau}{\Gamma; \Delta \vdash \texttt{let}\,a_1{:}\tau_1{\times}a_2{:}\tau_2\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} : \tau}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl} \Gamma; \Delta & \vdash & t : \tau_1 + \tau_2 \\ \Gamma; \Delta, a_1{:}\tau_1 & \vdash & e_1 : \tau \\ \Gamma; \Delta, a_2{:}\tau_2 & \vdash & e_2 : \tau \end{array}}{\Gamma; \Delta \vdash \texttt{mcase}\,t\,\texttt{of}\,\texttt{inl}\ (a_1{:}\tau_1)\ \Rightarrow e_1 \mid \texttt{inr}\ (a_2{:}\tau_2)\ \Rightarrow e_2\,\texttt{end} : \tau}\ \textbf{(case)}$$

Figure 12.7: Typing judgments for expressions.

argument, with the outcome determined independently of the second component in the case that the first is positive. By recording control-flow information during evaluation, we may use it to provide selective memoization.

For example, in the situation just described, all pairs of the form $\langle 0, v\rangle$ should map to the same result value, irrespective of the value v. In conventional memoization the memo table would be keyed by the pair, with the result that redundant computation is performed in the case that the function has not previously been called with v, even though the value of v is irrelevant to the result! In our framework we instead key the memo table by a "branch" that records sufficient control flow information to capture the general case. Whenever we encounter a `return` statement, we query the memo table with the current branch to determine whether this result has been computed before. If so, we return the stored value; if not, we evaluate the `return` statement, and associate that value with that branch in the memo table for future use. It is crucial that the returned term not contain any resources so that we are assured that its value does not change across calls to the function.

The dynamic semantics of MFL is given by a set of rules for deriving judgments of the form $\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'$ (for terms) and $\sigma, l{:}\beta, e \Downarrow^{\mathsf{e}} v, \sigma'$ (for expressions). The rules for deriving these judgments are given in Figure 12.8expr-dynamic. These rules make use of branches, memo tables, and stores, whose precise definitions are as follows.

A *simple branch* is a list of *simple events* corresponding to "choice points" in the evaluation of an expression.

$$\begin{array}{lrcl} \textit{Simple Event} & \varepsilon & ::= & !v \mid \texttt{inl} \mid \texttt{inr} \\ \textit{Simple Branch} & \beta & ::= & \bullet \mid \varepsilon \cdot \beta \end{array}$$

We write $\beta\frown\varepsilon$ to stand for the extension of β with the event ε at the end.

A *memo table*, θ, is a finite function mapping simple branches to values. We write $\theta[\beta \mapsto v]$, where $\beta \notin \mathrm{dom}(\theta)$, to stand for the extension of θ with the given binding for β. We write $\theta(\beta) \uparrow$ to mean that $\beta \notin \mathrm{dom}(\theta)$.

A *store*, σ, is a finite function mapping *locations*, l, to memo tables. We write $\sigma[l \mapsto \theta]$, where $l \notin$

dom(σ), to stand for the extension of σ with the given binding for l. When $l \in \mathrm{dom}(\sigma)$, we write $\sigma[l \leftarrow \theta]$ for the store σ that maps l to θ and $l' \neq l$ to $\sigma(l')$.

Term evaluation is largely standard, except for the evaluation of (memoizing) functions and applications of these to arguments. Evaluation of a memoizing function term allocates a fresh memo table, which is then associated with the function's value. Expression evaluation is initiated by an application of a memoizing function to an argument. The function value determines the memo table to be used for that call. Evaluation of the body is performed relative to that table, initiating with the null branch.

Expression evaluation is performed relative to a "current" memo table and branch. When a `return` statement is encountered, the current memo table is consulted to determine whether or not that branch has previously been taken. If so, the stored value is returned; otherwise, the argument term is evaluated, stored in the current memo table at that branch, and the value is returned. The `let!` and `mcase` expressions extend the current branch to reflect control flow. Since `let!` signals dependence on a complete value, that value is added to the branch. Case analysis, however, merely extends the branch with an indication of which case was taken. The `let`$\times$ construct does not extend the branch, because no additional information is gleaned by splitting a pair.

12.4.4 Soundness of MFL

We will prove the soundness of MFL relative to a non-memoizing semantics for the language. It is straightforward to give a purely functional semantics to MFL by an inductive definition of the relations $t \Downarrow^{\mathsf{t}}_{\mathsf{p}} v$ and $e \Downarrow^{\mathsf{e}}_{\mathsf{p}} v$, where v is a *pure value* with no location subscripts (see, for example, [79]). We will show that (Theorem 86) memoization does not affect the outcome of evaluation as compared to the non-memoized semantics. To make this precise, we must introduce some additional machinery.

The *underlying term*, t^-, of a term, t, is obtained by erasing all location subscripts on function values occurring within t. The *underlying expression*, e^-, of an expression, e, is defined in the same way. As a special case, the *underlying value*, v^-, of a value, v, is the underlying term of v regarded as a term. It is easy to check that every pure value arises as the underlying value of some impure value. Note that passage to the underlying term or expression obviously commutes with substitution. The *underlying branch*, β^-, of a simple branch, β, is obtained by replacing each event of the form $!\,v$ in β by the corresponding underlying event, $!\,(v^-)$.

The partial *access functions*, $t\,@\,\beta$ and $e\,@\,\beta$, where β is a simple branch, and t and e are variable-free (but not necessarily resource-free), are defined as follows. The definition may be justified by lexicographic induction on the structure of the branch followed by the size of the expression.

$$
\begin{array}{rcl}
t\,@\,\beta & = & e\,@\,\beta \\
\multicolumn{3}{r}{(\textit{where } t = \texttt{mfun}\,f\;\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\;\texttt{is}\;e\;\texttt{end})} \\[1em]
\texttt{return(}t\texttt{)}\;@\,\bullet & = & \texttt{return(}t\texttt{)} \\
\texttt{let}\;!\;x\texttt{:}\tau\;\texttt{be}\;t\;\texttt{in}\;e\;\texttt{end}\;@\,\beta\,\hat{}\,!\,v & = & [v/x]e\,@\,\beta \\
\texttt{let}\;a_1\texttt{:}\tau_1{\times}a_2\texttt{:}\tau_2\;\texttt{be}\;t\;\texttt{in}\;e\;\texttt{end}\;@\,\beta & = & e\,@\,\beta \\
\texttt{mcase}\;t\;\texttt{of}\;\texttt{inl (}a_1\texttt{:}\tau_1\texttt{)} \Rightarrow e_1 \mid \texttt{inr (}a_2\texttt{:}\tau_2\texttt{)} \Rightarrow e_2\;\texttt{end}\;@\,\beta\,\hat{}\,\texttt{inl} & = & e_1\,@\,\beta \\
\texttt{mcase}\;t\;\texttt{of}\;\texttt{inl (}a_1\texttt{:}\tau_1\texttt{)} \Rightarrow e_1 \mid \texttt{inr (}a_2\texttt{:}\tau_2\texttt{)} \Rightarrow e_2\;\texttt{end}\;@\,\beta\,\hat{}\,\texttt{inr} & = & e_2\,@\,\beta
\end{array}
$$

$$\frac{}{\sigma, \star \Downarrow^{\mathsf{t}} \star, \sigma}\ \textbf{(unit)} \qquad \frac{}{\sigma, n \Downarrow^{\mathsf{t}} n, \sigma}\ \textbf{(number)}$$

$$\frac{\sigma, t_1 \quad \Downarrow^{\mathsf{t}} \quad v_1, \sigma_1 \quad \ldots \quad \sigma_{n-1}, t_n \quad \Downarrow^{\mathsf{t}} \quad v_n, \sigma_n}{\sigma, o\,\texttt{(}t_1, \ldots, t_n\texttt{)} \Downarrow^{\mathsf{t}} \texttt{app}\,\texttt{(}o\texttt{,}\,\texttt{(}v_1, \ldots, v_n\texttt{)}\texttt{)}, \sigma_n}\ \textbf{(primitive)}$$

$$\frac{\begin{array}{ccc} \sigma, t_1 & \Downarrow^{\mathsf{t}} & v_1, \sigma' \\ \sigma', t_2 & \Downarrow^{\mathsf{t}} & v_2, \sigma'' \end{array}}{\sigma, \langle t_1, t_2 \rangle \Downarrow^{\mathsf{t}} \langle v_1, v_2 \rangle, \sigma''}\ \textbf{(pair)}$$

$$\frac{(l \notin \mathrm{dom}(\sigma))}{\sigma, \texttt{mfun}\, f\ \texttt{(}a\texttt{:}\tau_1\texttt{)}\,\texttt{:}\tau_2\,\texttt{is}\, e\, \texttt{end} \Downarrow^{\mathsf{t}} \texttt{mfun}_l\, f\, \texttt{(}a\texttt{:}\tau_1\texttt{)}\,\texttt{:}\tau_2\,\texttt{is}\, e\, \texttt{end}, \sigma[l \mapsto \emptyset]}\ \textbf{(fun)}$$

$$\frac{(l \in \mathrm{dom}(\sigma))}{\sigma, \texttt{mfun}_l\, f\, \texttt{(}a\texttt{:}\tau_1\texttt{)}\,\texttt{:}\tau_2\,\texttt{is}\, e\, \texttt{end} \Downarrow^{\mathsf{t}} \texttt{mfun}_l\, f\, \texttt{(}a\texttt{:}\tau_1\texttt{)}\,\texttt{:}\tau_2\,\texttt{is}\, e\, \texttt{end}, \sigma}\ \textbf{(fun val)}$$

$$\frac{\begin{array}{rcl} \sigma, t_1 & \Downarrow^{\mathsf{t}} & v_1, \sigma_1 \\ \sigma_1, t_2 & \Downarrow^{\mathsf{t}} & v_2, \sigma_2 \\ \sigma_2, l{:}\bullet, [v_1, v_2/f, a]\, e & \Downarrow^{\mathsf{e}} & v, \sigma' \\ \multicolumn{3}{l}{(v_1 = \texttt{mfun}_l\, f\, \texttt{(}a\texttt{:}\tau_1\texttt{)}\,\texttt{:}\tau_2\,\texttt{is}\, e\, \texttt{end})} \end{array}}{\sigma, t_1\, t_2 \Downarrow^{\mathsf{t}} v, \sigma'}\ \textbf{(apply)}$$

$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \texttt{!}\ t \Downarrow^{\mathsf{t}} \texttt{!}\ v, \sigma'}\ \textbf{(bang)}$$

$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \texttt{inl}_{\tau_1+\tau_2} t \Downarrow^{\mathsf{t}} \texttt{inl}_{\tau_1+\tau_2} v, \sigma'}\ \textbf{(case/inl)} \qquad \frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \texttt{inr}_{\tau_1+\tau_2} t \Downarrow^{\mathsf{t}} \texttt{inr}_{\tau_1+\tau_2} v, \sigma'}\ \textbf{(case/inr)}$$

$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \texttt{roll}\,\texttt{(}t\texttt{)} \Downarrow^{\mathsf{t}} \texttt{roll}\,\texttt{(}v\texttt{)}, \sigma'}\ \textbf{(roll)} \qquad \frac{\sigma, t \Downarrow^{\mathsf{t}} \texttt{roll}\,\texttt{(}v\texttt{)}, \sigma'}{\sigma, \texttt{unroll}\,\texttt{(}t\texttt{)} \Downarrow^{\mathsf{t}} v, \sigma'}\ \textbf{(unroll)}$$

Figure 12.8: Evaluation of terms.

This function will only be of interest in the case that $e \mathbin{@} \beta$ is a `return` expression, which, if well-typed, cannot contain free resources. Note that $(e \mathbin{@} \beta)^- = e^- \mathbin{@} \beta^-$, and similarly for values, v.

We are now in a position to justify a subtlety in the second `return` rule of the dynamic semantics, which governs the case that the returned value has not already been stored in the memo table. This rule extends, rather than updates, the memo table with a binding for the branch that determines this `return` statement within the current memoized function. But why, after evaluation of t, is this branch undefined in the revised store, σ'? If the term t were to introduce a binding for β in the memo table $\sigma(l)$, it could only do so by evaluating the very same `return` statement, which implies that there is an infinite loop, contradicting the assumption that the `return` statement has a value, v.

$$\frac{\sigma(l)(\beta) = v}{\sigma, l{:}\beta, \texttt{return}\,(t) \Downarrow^{\mathsf{e}} v, \sigma}\ \textbf{(return, found)}$$

$$\frac{\begin{array}{c}\sigma(l) = \theta \quad \theta(\beta)\uparrow \\ \sigma, t \Downarrow^{\mathsf{t}} v, \sigma' \\ \sigma'(l) = \theta'\end{array}}{\sigma, l{:}\beta, \texttt{return}\,(t) \Downarrow^{\mathsf{e}} v, \sigma'[l \leftarrow \theta'[\beta \mapsto v]]}\ \textbf{(return, not found)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & !\,v, \sigma' \\ \sigma', l{:}!\,v\cdot\beta, [v/x]e & \Downarrow^{\mathsf{t}} & v', \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{let}\ !\,x : \eta\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} \Downarrow^{\mathsf{e}} v', \sigma''}\ \textbf{(let!)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & v_1 \times v_2, \sigma' \\ \sigma', l{:}\beta, [v_1/a_1, v_2/a_2]e & \Downarrow^{\mathsf{e}} & v, \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{let}\,a_1{\times}a_2\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} \Downarrow^{\mathsf{t}} v, \sigma''}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & \texttt{inl}_{\tau_1+\tau_2} v, \sigma' \\ \sigma', l{:}\texttt{inl}\cdot\beta, [v/a_1]e_1 & \Downarrow^{\mathsf{e}} & v_1, \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{mcase}\,t\,\texttt{of}\,\texttt{inl}\ (a_1{:}\tau_1)\ \Rightarrow\ e_1 \mid \texttt{inr}\ (a_2{:}\tau_2)\ \Rightarrow\ e_2\,\texttt{end} \Downarrow^{\mathsf{t}} v_1, \sigma''}\ \textbf{(case/inl)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & \texttt{inr}_{\tau_1+\tau_2} v, \sigma' \\ \sigma', l{:}\texttt{inr}\cdot\beta, [v/a_2]e_2 & \Downarrow^{\mathsf{e}} & v_2, \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{mcase}\,t\,\texttt{of}\,\texttt{inl}\ (a_1{:}\tau_1)\ \Rightarrow\ e_1 \mid \texttt{inr}\ (a_2{:}\tau_2)\ \Rightarrow\ e_2\,\texttt{end} \Downarrow^{\mathsf{t}} v_2, \sigma''}\ \textbf{(case/inr)}$$

Figure 12.9: Evaluation of expressions.

Lemma 83
If $\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'$, $\sigma(l) \mathbin{@} \beta = \texttt{return}\,(t)$, and $\sigma(l)(\beta)$ is undefined, then $\sigma'(l)(\beta)$ is also undefined.

An *augmented branch*, γ, is an extension of the notion of branch in which we record the bindings of resource variables. Specifically, the argument used to call a memoized function is recorded, as are the bindings of resources created by pair splitting and case analysis. Augmented branches are inductively defined by the following grammar:

$$\begin{array}{lccl}\textit{Augmented Event} & \epsilon & ::= & \texttt{(}v\texttt{)} \mid !\,v \mid \langle v_1, v_2\rangle \mid \texttt{inl}\,(v) \mid \texttt{inr}\,(v) \\ \textit{Augmented Branch} & \gamma & ::= & \bullet \mid \epsilon\cdot\gamma\end{array}$$

We write $\gamma\,\widehat{}\,\epsilon$ for the extension of γ with ϵ at the end. There is an obvious *simplification* function, γ°, that yields the simple branch corresponding to an augmented branch by dropping "call" events, $\texttt{(}v\texttt{)}$, and "pair" events, $\langle v_1, v_2\rangle$, and by omitting the arguments to "injection" events, $\texttt{inl}\,(v)$, $\texttt{inr}\,(v)$. The *underlying augmented branch*, γ^-, corresponding to an augmented branch, γ, is defined by replacing each augmented event, ϵ, by its corresponding underlying augmented event, ϵ^-, which is defined in the obvious manner. Note that $(\gamma^\circ)^- = (\gamma^-)^\circ$.

The partial access functions $e \mathbin{@} \gamma$ and $t \mathbin{@} \gamma$ are defined for closed expressions e and closed terms t by the following equations:

$$\begin{array}{rcl} t\,@\,\gamma\verb|^|(v) & = & [t, v/f, a]e\,@\,\gamma \\ (\textit{where } t = \mathtt{mfun}\, f\ (a\,\mathtt{:}\,\tau_1)\,\mathtt{:}\,\tau_2\ \mathtt{is}\ e\ \mathtt{end}) & & \end{array}$$

$$\begin{array}{rcl} e\,@\,\bullet & = & e \\ \mathtt{let}\ !\,x\mathtt{:}\tau\ \mathtt{be}\ t\ \mathtt{in}\ e\ \mathtt{end}\,@\,\gamma\verb|^|!v & = & [v/x]e\,@\,\gamma \\ \mathtt{let}\ a_1\mathtt{:}\tau_1\times a_2\mathtt{:}\tau_2\ \mathtt{be}\ t\ \mathtt{in}\ e\ \mathtt{end}\,@\,\beta\verb|^|\langle v_1, v_2\rangle & = & [v_1, v_2/a_1, a_2]e\,@\,\beta \\ \mathtt{mcase}\ t\ \mathtt{of}\ \mathtt{inl}\ (a_1\mathtt{:}\tau_1)\ \Rightarrow\ e_1 \mid \mathtt{inr}\ (a_2\mathtt{:}\tau_2)\ \Rightarrow\ e_2\ \mathtt{end}\,@\,\beta\verb|^|\mathtt{inl}(v) & = & [v/a_1]e_1\,@\,\beta \\ \mathtt{mcase}\ t\ \mathtt{of}\ \mathtt{inl}\ (a_1\mathtt{:}\tau_1)\ \Rightarrow\ e_1 \mid \mathtt{inr}\ (a_2\mathtt{:}\tau_2)\ \Rightarrow\ e_2\ \mathtt{end}\,@\,\beta\verb|^|\mathtt{inr}(v) & = & [v/a_2]e_2\,@\,\beta \end{array}$$

Note that $(e\,@\,\gamma)^- = e^-\,@\,\gamma^-$, and similarly for values, v.

Augmented branches, and the associated access function, are needed for the proof of soundness. The proof maintains an augmented branch that enriches the current simple branch of the dynamic semantics. The additional information provided by augmented branches is required for the induction, but it does not affect any `return` statement it may determine.

Lemma 84
If $e\,@\,\gamma = \mathtt{return}(t)$*, then* $e\,@\,\gamma^\circ = \mathtt{return}(t)$*.*

A *function assignment*, Σ, is a finite mapping from locations to well-formed, closed, pure function values. A function assignment is *consistent with* a term, t, or expression, e, if and only if whenever $\mathtt{mfun}_l\, f\ (a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end}$ occurs in either t or e, then $\Sigma(l) = \mathtt{mfun}\, f\ (a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\ e^-\ \mathtt{end}$. Note that if a term or expression is consistent with a function assignment, then no two function values with distinct underlying values may have the same label. A function assignment is consistent with a store, σ, if and only if whenever $\sigma(l)(\beta) = v$, then Σ is consistent with v.

A store, σ, *tracks* a function assignment, Σ, if and only if Σ is consistent with σ, $\mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma)$, and for every $l \in \mathrm{dom}(\sigma)$, if $\sigma(l)(\beta) = v$, then

1. $\Sigma(l)\,@\,\beta^- = \mathtt{return}(t^-)$,
2. $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$,

Thus if a branch is assigned a value by the memo table associated with a function, it can only do so if that branch determines a `return` statement whose value is the assigned value of that branch, relative to the non-memoizing semantics.

We are now in a position to prove the soundness of MFL.

Theorem 85

1. *If* $\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'$, Σ *is consistent with* t, σ *tracks* Σ, $\emptyset;\emptyset \vdash t : \tau$, *then* $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$ *and there exists* $\Sigma' \supseteq \Sigma$ *such that* Σ' *is consistent with* v *and* σ' *tracks* Σ'.
2. *If* $\sigma, l{:}\beta, e \Downarrow^{\mathsf{e}} v, \sigma'$, Σ *is consistent with* e, σ *tracks* Σ, $\gamma^\circ = \beta$, $\Sigma(l)\,@\,\gamma^- = e^-$, *and* $\emptyset;\emptyset \vdash e : \tau$, *then there exists* $\Sigma' \supseteq \Sigma$ *such that* $e^- \Downarrow^{\mathsf{e}}_{\mathsf{p}} v^-$, Σ' *is consistent with* v, *and* σ' *tracks* Σ'.

Proof: The proof proceeds by simultaneous induction on the memoized evaluation relation. We consider here the five most important cases of the proof: function values, function terms, function application terms, and return expressions.

For function values $t = \texttt{mfun}_l\, f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}$, simply take $\Sigma' = \Sigma$ and note that $v = t$ and $\sigma' = \sigma$.

For function terms $t = \texttt{mfun}\, f\ \texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}$, note that $v = \texttt{mfun}_l\, f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}$ and $\sigma' = \sigma[l \mapsto \emptyset]$, where $l \notin \mathrm{dom}(\sigma)$. Let $\Sigma' = \Sigma[l \mapsto v^-]$, and note that since σ tracks Σ, and $\sigma(l) = \emptyset$, it follows that σ' tracks Σ'. Since Σ is consistent with t, it follows by construction that Σ' is consistent with v. Finally, since $v^- = t^-$, we have $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$, as required.

For application terms $t = t_1\, t_2$, we have by induction that $t_1{}^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v_1{}^-$ and there exists $\Sigma_1 \supseteq \Sigma$ consistent with v_1 such that σ_1 tracks Σ_1. Since $v_1 = \texttt{mfun}_l\, f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}$, it follows from consistency that $\Sigma_1(l) = v_1{}^-$. Applying induction again, we obtain that $t_2{}^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v_2{}^-$, and there exists $\Sigma_2 \supseteq \Sigma_1$ consistent with v_2 such that σ_2 tracks Σ_2. It follows that Σ_2 is consistent with $[v_1, v_2/f, a]e$. Let $\gamma = \texttt{(}v_2\texttt{)} \cdot \bullet$. Note that $\gamma^\circ = \bullet = \beta$ and we have

$$\begin{array}{rcl}
\Sigma_2(l) \mathbin{@} \gamma^- & = & v_1{}^- \mathbin{@} \gamma^- \\
& = & (v_1 \mathbin{@} \gamma)^- \\
& = & ([v_1, v_2/f, a]e)^- \\
& = & [v_1{}^-, v_2{}^-/f, a]e^-.
\end{array}$$

Therefore, by induction, $[v_1{}^-, v_2{}^-/f, a]e^- \Downarrow^{\mathsf{e}}_{\mathsf{p}} v'^-$, and there exists $\Sigma' \supseteq \Sigma_2$ consistent with v' such that σ' tracks Σ'. It follows that $(t_1\, t_2)^- = t_1{}^-\, t_2{}^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v'^-$, as required.

For return statements, we have two cases to consider, according to whether the current branch is in the domain of the current memo table. Suppose that $\sigma, l{:}\beta, \texttt{return(}t\texttt{)} \Downarrow^{\mathsf{e}} v, \sigma'$ with Σ consistent with $\texttt{return(}t\texttt{)}$, σ tracking Σ, $\gamma^\circ = \beta$, $\Sigma(l)@\gamma^- = (\texttt{return(}t\texttt{)})^- = \texttt{return(}t^-\texttt{)}$, and $\emptyset;\emptyset \vdash \texttt{return(}t\texttt{)} : \tau$. Note that by Lemma 84, $(\Sigma(l) \mathbin{@} \beta)^- = \Sigma(l) \mathbin{@} \beta^- = \texttt{return(}t^-\texttt{)}$.

For the first case, suppose that $\sigma(l)(\beta) = v$. Since σ tracks Σ and $l \in \mathrm{dom}(\sigma)$, we have $\Sigma(l) = \texttt{mfun}\, f\ \texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e^-\,\texttt{end}$ with $e^- \mathbin{@} \beta^- = \texttt{return(}t^-\texttt{)}$, and $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$. Note that $\sigma' = \sigma$, so taking $\Sigma' = \Sigma$ completes the proof.

For the second case, suppose that $\sigma(l)(\beta)$ is undefined. By induction $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$ and there exists $\Sigma' \supseteq \Sigma$ consistent with v such that σ' tracks Σ'. Let $\theta' = \sigma'(l)$, and note $\theta'(\beta) \uparrow$, by Lemma 83. Let $\theta'' = \theta'[\beta \mapsto v]$, and $\sigma'' = \sigma'[l \leftarrow \theta'']$. Let $\Sigma'' = \Sigma'$; we are to show that Σ'' is consistent with v, and σ'' tracks Σ''. By the choice of Σ'' it is enough to show that $\Sigma'(l) \mathbin{@} \beta^- = \texttt{return(}t^-\texttt{)}$, which we noted above.

■

The soundness theorem (Theorem 86) for MFL states that evaluation of a program (a closed term) with memoization yields the same outcome as evaluation without memoization. The theorem follows from Theorem 85.

Theorem 86 (Soundness)
If $\emptyset, t \Downarrow^{\mathsf{t}} v, \sigma$*, where* $\emptyset;\emptyset \vdash t : \tau$*, then* $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$*.*

Type safety follows from the soundness theorem, since type safety holds for the non-memoized semantics. In particular, if a term or expression had a non-canonical value in the memoized semantics, then the same term or expression would have a non-canonical value in the non-memoized semantics, contradicting safety for the non-memoized semantics.

12.4.5 Performance

We show that memoization slows down an MFL program by a constant factor (expected) with respect to a standard, non-memoizing semantics even when no results are re-used. The result relies on representing a branch as a sequence of integers and using this sequence to key memo tables, which are implemented as hash tables. To represent branches as integer sequences we use the property of MFL that the underlying type η of a bang type, $!\ \eta$, is an indexable type. Since any value of an indexable type has an integer index, we can represent a branch of dependencies as sequence of integers corresponding to the indices of `let!`'ed values, and zero or one for `inl` and `inr`.

Consider a non-memoizing semantics, where the `return` rule always evaluates its body and neither looks up nor updates memo tables (stores). Consider an MFL program and let T denote the time it takes (the number of evaluation steps) to evaluate the program with respect to this non-memoizing semantics. Let T' denote the time it takes to evaluate the same program with respect to the memoizing semantics. In the worst case, no results are re-used, thus the difference between T and T' is due to memo-table lookups and updates done by the memoizing semantics. To bound the time for these, consider a memo table lookup or update with a branch β and let $|\beta|$ be the length of the branch. Since a branch is a sequence of integers, a lookup or update can be performed in expected $O(|\beta|)$ time using nested hash tables to represent memo tables. Now note that the non-memoizing semantics takes $|\beta|$ time to build the branch thus, the cost of a lookup or update can be charged to the evaluations that build the branch β, *i.e.*, evaluations of `let!` and `mcase`. Furthermore, each evaluation of `let!` and `mcase` can be charged by exactly one `return`. Thus, we conclude that $T' = O(T)$ in the expected case.

12.5 Implementation

We describe an implementation of our framework as a Standard ML library. The aspects of the MFL language that relies on the syntactic distinction between resources and variables cannot be enforced statically in Standard ML. Therefore, we use a separate type for resources and employ run-time checks to detect violations of correct usage.

The interface for the library (shown in Figure 12.10) provides types for expressions, resources, bangs, products, sums, memoized functions along with their introduction and elimination forms. All expressions have type `'a expr`, which is a monad with `return` as the inclusion and various forms of "bind" induced by the elimination forms `letBang, letx`, and `mcase`. A resource has type `'a res` and `expose` is its elimination form. Resources are only created by the library, thus no introduction form for resources is available to the user. The introduction and elimination form for bang types are `bang` and `letBang`. The introduction and elimination form for product types are `pair`, and `letx` and `split` respectively. The `letx` is a form of "bind" for the monad `expr`; `split` is the elimination form for the term context. The treatment of sums is similar to product types. The introduction forms are `inl` and `inr`, and the elimination forms are `mcase` and `choose`; `mcase` is a form of bind for the `expr` monad and `choose` is the elimination for the term context.

```
signature MEMO =
sig
  (* Expressions *)
  type 'a expr
  val return:(unit -> 'a) -> 'a expr

  (* Resources *)
  type 'a res
  val expose:'a res -> 'a

  (* Bangs *)
  type 'a bang
  val bang :('a -> int) -> 'a -> 'a bang
  val letBang:('a bang) -> ('a -> 'b expr) -> 'b expr

  (* Products *)
  type ('a,'b) prod
  val pair:'a -> 'b -> ('a,'b) prod
  val letx:('a,'b) prod -> (('a res * 'b res) -> 'c expr) -> 'c expr
  val split:('a,'b) prod -> (('a * 'b) -> 'c) -> 'c

  (* Sums *)
  type ('a,'b) sum
  val inl:'a -> ('a,'b) sum
  val inr:'b -> ('a,'b) sum
  val mcase:('a,'b) sum -> ('a res -> 'c expr) -> ('b res -> 'c expr)-> 'c expr
  val choose:('a,'b) sum -> ('a -> 'c) -> ('b -> 'c) -> 'c

  (* Memoized arrow *)
  type ('a,'b) marrow
  val mfun:('a res -> 'b expr) -> ('a,'b) marrow
  val mfun_rec:(('a, 'b) marrow -> 'a res -> 'b expr) -> ('a,'b) marrow
  val mapply:('a,'b) marrow -> 'a -> 'b
end

signature BOX = sig
  type 'a box
  val init:unit->unit
  val box:'a->'a box
  val unbox:'a box->'a
  val getKey:'a box->int
end
```

Figure 12.10: The signatures for the memo library and boxes.

Memoized functions are introduced by `mfun` and `mfun_rec`; `mfun` takes a function of type `'a res -> 'b expr` and returns the memoized function of type `('a,'b) marrow`; `mfun_rec` is similar to `mfun` but it also takes as a parameter its memoized version. Note that the result type does not contain the "effect" `expr`—we encapsulate memoization effects, which are benign, within the function. The elimination form for the `marrow` is the memoized apply function `mapply`.

Figure 12.11 shows an implementation of the library without the run-time checks for correct usage. To incorporate the run-time checks, one needs a more sophisticated definition of resources in order to detect

```
functor BuildMemo (structure Box:BOX structure Memopad:MEMOPAD):MEMO =
struct
  type 'a expr = int list * (unit -> 'a)
  fun return f = (nil,f)

  type 'a res = 'a
  fun res v = v
  fun expose r = r

  type 'a bang = 'a * ('a -> int)
  fun bang h t = (t,h)
  fun letBang b f =
    let val (v,h) = b
        val (branch,susp) = f v
    in ((h v)::branch, susp) end

  type ('a,'b) prod = 'a * 'b
  fun pair x y = (x,y)
  fun split p f = f p
  fun letx (p as (x1,x2)) f = f (res x1, res x2)

  datatype ('a,'b) sum = INL of 'a | INR of 'b
  fun inl v = INL(v)
  fun inr v = INR(v)
  fun mcase s f g =
    let val (lr,(branch,susp)) = case s of
                                   INL v => (0,f (res v))
                                 | INR v => (1,g (res v))
    in (lr::branch,susp) end
  fun choose s f g = case s of INL v => f v | INR v => g v

  type ('a,'b) marrow = 'a -> 'b
  fun mfun_rec f =
    let val mpad = Memopad.empty ()
        fun mf rf x =
          let val (branch,susp) = f rf (res x)
              val result = case Memopad.extend mpad branch of
                             (NONE,SOME mpad') =>        (* Not found *)
                               let val v = susp ()
                                   val _ = Memopad.add v mpad'
                               in v end
                           | (SOME v,NONE) => v          (* Found *)
          in result end
        fun mf' x = mf mf' x
    in mf' end

  fun mfun f = ...  (* Similar to mfun_rec *)

  fun mapply f v = f v
end
```

Figure 12.11: The implementation of the memoization library.

when a resource is exposed out of its context (*i.e.*, function instance). In addition, the interface must be updated so that the first parameter of `letBang`, `letx`, and `mcase`, occurs in suspended form. This allows us to update the state consisting of certain flags before forcing a term.

The implementation extends the operational semantics of the MFL language (Section 12.4.3) with boxes. The `bang` primitive takes a value and an injective function, called the index function, that maps the value to an integer, called the index. The index of a value is used to key memo tables. The restriction that the indices be unique, enables us to implement memo tables as a nested hash tables, which support update and lookup operations in expected constant time. The primitive `letBang` takes a value `b` of bang type and a body. It applies the body to the underlying value of `b`, and extends the branch with the index of `b`. The function `letx` takes a pair `p` and a body. It binds the parts of the pair to two resources and and applies the body to the resources; as with the operational semantics, `letx` does not extend the branch. The function `mcase` takes value `s` of sum type and a body. It branches on the outer form of `s` and binds its inner value to a resource. It then applies the body to the resource and extends the branch with `0` or `1` depending on the outer form of `s`. The elimination forms of sums and products for the term context, `split` and `choose` are standard.

The `return` primitive finalizes the branch and returns its body as a suspension. The branch is used by `mfun_rec` or `mfun`, to key the memo table; if the result is found in the memo table, then the suspension is disregarded and the result is re-used; otherwise the suspension is forces and the result is stored in the memo table keyed by the branch. The `mfun_rec` primitive takes a recursive function `f` as a parameter and "memoizes" `f` by associating it with a memo pad. A subtle issue is that `f` must calls its memoized version recursively. Therefore `f` must take its memoized version as a parameter. Note also that the memoized function internally converts its parameter to a resource before applying `f` to it.

The interface of the library provides no introduction form for resources. Indeed, all resources are created by the library inside the `letx`, `mcase`, `mfun_rec`, and `mfun`. The function `expose` is the elimination form for resources. If, for example, one would like to apply `letBang` to a resource, then he must first `expose` the resource, which "exposes" the underlying value.

Figure 12.12 show the examples from Section 12.3 written in the SML library. Note that the memoized Fibonacci function `mfib` creates a memo table every time it is called. When `mfib` finishes, this table can be garbage collected (the same applies to `mks`). For Quicksort, we provide a function `mqs` that returns an instance of memoized Quicksort when applied. Each such instance has its own memo table. Note also that `mqs` creates a local instance of the hash-cons function so that each instance of memoized Quicksort has its own memo table for hash-consing.

In the examples, we do not use the sum types provided by the library to represent boxed lists, because we do not need to. In general, one will use the provided sum types instead of their ML counterparts (for example if an `mcase` is needed). The examples in Figure 12.12 can be implemented using the following definition of boxed lists.

```
datatype 'a boxlist' =
         ROLL of (unit, (('a, 'a boxlist' box) prod)) sum
type 'a boxlist = ('a boxlist') box
```

Changing the code in Figure 12.12 to work with this definition of boxed lists requires several straightforward modifications.

12.6 Discussion

Space and Cache Management. Our framework associates a separate memo table with each memoized function. This allows the programmer to control the life-span of memo tables by conventional scoping. In some applications, finer level of control over memo table entries may be desirable. In particular, an application can benefit from specifying a caching scheme for each memo table that determines the size of the memo table and the replacement policy. We discuss how the framework can be extended to support this type of control over memo tables.

The caching scheme should be specified in the form of a parameter to the `mfun` construct. When evaluated, this construct will bind the caching scheme to the memo table and the memo table will be maintained accordingly. Changes to the operational semantics to accommodate this extension is small. The store σ will now map a label to a pair consisting of a memo table and its caching scheme. The handling of the `return` will be changed so that the stores do not merely expand but are updated according to the caching scheme before adding a new entry. The following shows the updated return rule. Here $\mathcal{S}$ denotes a caching scheme and θ denotes a memo table. The `update` function denotes a function that updates the memo table to accommodate a new entry by possibly purging an existing entry. The programmer must ensure that the caching scheme does not violate the integrity of the memo table by tampering with stored values.

$$\frac{\sigma(l) = (\theta, \mathcal{S}) \quad \theta(\beta) = v}{\sigma, l{:}\beta, \texttt{return}\,(t) \Downarrow^{\mathsf{e}} v, \sigma} \quad \text{(Found)}$$

$$\frac{\begin{array}{c}\sigma(l) = (\theta, \mathcal{S}) \quad \theta(\beta)\uparrow \\ \sigma, t \Downarrow^{\mathsf{t}} v, \sigma' \\ \sigma'(l) = (\theta', \mathcal{S}) \quad \theta'' = \texttt{update}(\theta', \mathcal{S}, (\beta, v))\end{array}}{\sigma, l{:}\beta, \texttt{return}\,(t) \Downarrow^{\mathsf{e}} v, \sigma'[l \leftarrow \theta'']} \quad \text{(Not Found)}$$

For example, we can specify that the memo table for the Fibonacci function, shown in Figure 12.1, can contain at most two entries and be managed using the least-recently-used replacement policy. This is sufficient to ensure that the memoized Fibonacci runs in linear time. This extension can also be incorporated into the type system described in Section 12.4. This would require that we associate types with memo stores and also require that we develop a type system for “safe” `update` functions.

Local *vs.* Non-local Dependences. Our techniques only track “local” dependences between the input and the result of a function. Local dependences of a function `f` are those that are created inside the static scope of `f`. A non-local dependence of `f` is created when `f` passes its input to some other function `g`, which examines `f`’s input indirectly. In previous work, Abadi *et. al.* [1] and Heydon *et. al.* [47] showed a program analysis technique for tracking non-local dependences by propagating dependences of a function to its caller. They do not, however, make clear the performance implications of their technique.

Our framework can be extended to track non-local dependences by introducing a memoized application construct for expressions. This extension would, for example, allow for dependences of non-constant length. We chose not to support non-local dependences because it is not clear if its utility exceeds its overhead.

```
structure Examples =
struct
  type 'a box = 'a Box.box

  fun iB v = bang (fn i => i) v
  fun bB b = bang (fn b => Box.key b) b

  (** Boxed lists **)
  datatype 'a blist' = NIL | CONS of ('a * (('a blist') box))

  type 'a blist = ('a blist') box
  (** Hash-cons **)
  fun hCons' (x') =
    letx (expose x') (fn (h',t') =>
    letBang (expose h') (fn h => letBang (expose t') (fn t =>
    return (fn()=> box (CONS(h,t))))))
  val hCons = mfun hCons'

  (** Fibonacci **)
  fun mfib' f (n') =
    letBang (expose n') (fn n =>
    return (fn()=>if n < 2 then n else (mapply f (iB(n-1))) + (mapply f (iB(n-2)))
  fun mfib n = mapply (mfun_rec mfib') n

  (** Knapsack **)
  fun mks' mks (arg) =
    letx (expose arg) (fn (c',l') =>
    letBang (expose c') (fn c =>
    letBang (expose l') (fn l => return (fn () =>
      case (unbox l) of
        NIL => 0
      | CONS((w,v),t) => if (c < w) then mapply mks (pair (iB c) (bB t))
                         else let val v1 = mapply mks (pair (iB c) (bB t))
                                  val v2 = v + mapply mks (pair (iB (c-w)) (bB t))
                              in if (v1 > v2) then v1 else v2 end))))

  val mks x = mfun_rec mks'

  (** Quicksort **)
  fun mqs () =
    let val empty = box NIL
        val hCons = mfun hCons'
        fun fil f l =
          case (unbox l) of
            NIL => empty
          | CONS(h,t) => if (f h) then (mapply hCons (pair (iB h) (bB (fil f t))))
                         else (fil f t)
        fun qs' qs (l') = letBang (expose l') (fn l => return (fn () =>
            case (unbox l) of
              NIL => nil
            | CONS(h,t) => let val ll = fil (fn x=>x<h) t
                               val gg = fil (fn x=>x>=h) t
                               val sll = mapply qs (bB ll)
                               val sgg = mapply qs (bB gg)
                           in sll@(h::sgg) end))
    in mfun_rec qs' end
end
```

Figure 12.12: Examples from Section 12.3 in the SML library.

Chapter 13

Self-Adjusting Functional Programming

This chapter presents a purely functional language, called Self-adjusting functional Language (SLf, read "self"), for writing self-adjusting programs. The language enables the programmer to transform an ordinary (non-self-adjusting) program into a self-adjusting program by making small changes to the program in a methodical fashion. Self-adjusting programs written in SLf have the following properties:

- their performance under an input change can be determined using analytical techniques,
- they yield programs that self-adjust efficiently, even when compared to special-purpose algorithms.

The SLf language combines the AFL and MFL languages and extends them with non-strict dependences. The key mechanisms under the SLf language are the memoized dynamic dependence graphs (MDDGs), and the memoized change-propagation algorithm. As a SLf language executes an MDDG is constructed. The MDDG is then used to adjust the computation to external changes via memoized change propagation. The SLf language allows computations to be tagged with two kinds of dependences: *strict* and *non-strict*. To re-use a memoized computation, it is only required that the strict dependences match. Re-used computations are adjusted to non-strict dependences by running the memoized change-propagation algorithm on the MDDG of the re-used computation. Supporting non-strict dependences requires relatively simple extensions to the memoized DDGs and the memoized change-propagation algorithm.

We present a static semantics and a dynamic semantics for the SLf language. The static semantics is a slight extension of the combination of the static semantics of the MFL and the AFL languages. It is therefore straightforward to prove the type safety of the SLf language based on the type safety of AFL and MFL languages. The dynamic semantics formalizes the construction of memoized DDGs. Based on the dynamic semantics, we present a semantics for memoized change propagation.

The semantics for memoized change propagation combines the semantics of non-memoized change propagation with the semantics of selective memoization. Although we prove both non-memoized change propagation (Section 11.5.2) and selective memoization (Section 12.4.4), we omit the proof for memoized change propagation. Because of the large number of cases to be considered and the technical challenges that arise when combining memoization (a purely functional technique) with change-propagation (which relies on side effects), the proof is quite involved. The proof, however, seems tractable (we are quite close), and therefore we a conjecture that the memoized change-propagation algorithm is correct.

In the SLf language, the programmer has full control over what dependences should be strict and what dependences should be non-strict. The programmer can therefore deem any dependence non-strict. It makes economical sense, however, to deem only certain kinds of dependences non-strict. In particular, when memoizing an expression e, it suffices to deem non-strict the dependences to values that are only passed to other functions—dependences to all other values should be strict. This *strictness principle* yields a methodological way to transform an ordinary (non-self-adjusting) program into a self-adjusting program.

A key property of the SLf language is that it enables writing efficient self-adjusting programs. We show in Chapter 15 that the language yields efficient programs from ordinary programs by considering a number of applications. For these applications, we validate that the self-adjusting programs written in the SLf language match the complexity bounds that we have obtained in Part III by performing an experimental evaluation. Note that the bounds obtained in Part III rely on an imperative computation model (the closure machine model).

The key to the efficiency of the SLf language is memoized change propagation and non-strict dependences. Although memoized change propagation suffices to obtain good complexity bounds in the imperative setting, this does now work in the purely functional setting. This is because the complexity bounds presented in the imperative setting crucially rely on the explicit management of memory allocation. Since purely functional programs do not allow explicit management of memory, memoized change propagation alone does not yield efficient self-adjusting programs. We address this problem by enabling memoization of computations independent of memory locations, which can be deemed as non-strict dependences.

An important property of the SLf language is that the complexity of change propagation for programs written in the language can be determined analytically. The two approaches for such analysis are (1) to transform the program into the imperative setting presented in the first part of this thesis and use trace-stability analysis techniques, (2) to give a cost model for change propagation based on traces. The first approach is relatively straightforward, because the imperative model is more expressive than the purely functional model supported by the SLf language. The second approach has the advantage that the programmer need to remain only in the purely functional setting. The traces used for change propagation, however, are too detailed and somewhat inflexible for the more informal and algorithmic approach that is traditionally preferred for the purposes of complexity analysis.

13.1 The Language

Figure 13.1 shows the abstract syntax for the *Self-Adjusting functional Language SLf*. Meta-variables x, y, z and their variants range over an unspecified set of variables, meta-variables a, b, c and variants range over an unspecified set of resources. Meta variable l and variants range over a unspecified set of locations. Meta variable m ranges over a unspecified set of memo-function identifiers. Variables, resources, locations, memo-function identifiers are mutually disjoint. The syntax of SLf is restricted to "2/3-cps" or "named form" to streamline the presentation of the dynamic semantics.

The types of SLf consists of the types of the AFL and the MFL languages. In particular, the types include the base type `int`, bang types $!\,\tau$, modifiables $\texttt{mod}_\tau$, products $\tau_1 \times \tau_2$ and sums $\tau_1 + \tau_2$, stable functions $\tau_1 \xrightarrow{\mathsf{ms}} \tau_2$, and changeable function $\tau_1 \xrightarrow{\mathsf{mc}} \tau_2$. Extending SLf with recursive or polymorphic types presents no fundamental difficulties but omitted here for the sake of brevity.

As with the MFL language, the underlying type of a bang type $!\,\tau$ is required to be an indexable type.

Types	τ	$::=$	$\texttt{int} \mid \texttt{!}\,\tau \mid \tau\,\texttt{mod} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \xrightarrow{\texttt{s}} \tau_2 \mid \tau_1 \xrightarrow{\texttt{c}} \tau_2$
Values	v	$::=$	$n \mid x \mid a \mid l \mid m \mid \texttt{!}\,v \mid \texttt{(}v_1\texttt{,}v_2\texttt{)} \mid \texttt{inl}_{\tau_1+\tau_2} v \mid \texttt{inr}_{\tau_1+\tau_2} v \mid$ $\texttt{ms\_fun}_m\, f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e_s\ \texttt{end} \mid \texttt{mc\_fun}_m\, f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e_c\ \texttt{end}$
Operators	o	$::=$	$\texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots$
Stable Expr	e_s	$::=$	$\texttt{return}\,\texttt{(}t_s\texttt{)} \mid \texttt{memo}\ (x = v)^*\ \texttt{in}\ t_s\ \texttt{end} \mid \texttt{let}\ a\texttt{:}\tau\ \texttt{be}\ t_s\ \texttt{in}\ e_s\ \texttt{end} \mid$ $\texttt{let !}\ x\texttt{:}\tau\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} \mid \texttt{let}\ a_1\texttt{:}\tau_1 \times a_2\texttt{:}\tau_2\ \texttt{be}\ v\ \texttt{in}\ e_s\ \texttt{end} \mid$ $\texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\ \texttt{(}a_1\texttt{:}\tau_1\texttt{)} \Rightarrow e_s \mid \texttt{inr}\ \texttt{(}a_2\texttt{:}\tau_2\texttt{)} \Rightarrow e_s\ \texttt{end}$
Changeable Expr	e_c	$::=$	$\texttt{return}\,\texttt{(}t_c\texttt{)} \mid \texttt{memo}\ (x = v)^*\ \texttt{in}\ t_c\ \texttt{end} \mid \texttt{let}\ a\texttt{:}\tau\ \texttt{be}\ t_s\ \texttt{in}\ e_s\ \texttt{end} \mid$ $\texttt{let !}\ x\texttt{:}\tau\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} \mid \texttt{let}\ a_1\texttt{:}\tau_1 \times a_2\texttt{:}\tau_2\ \texttt{be}\ v\ \texttt{in}\ e_c\ \texttt{end} \mid$ $\texttt{mcase}\ v\ \texttt{of}\ \texttt{inl}\ \texttt{(}a_1\texttt{:}\tau_1\texttt{)} \Rightarrow e_c \mid \texttt{inr}\ \texttt{(}a_2\texttt{:}\tau_2\texttt{)} \Rightarrow e'_c\ \texttt{end}$
Stable Terms	t_s	$::=$	$v \mid o\,\texttt{(}v_1, \ldots, v_n\texttt{)} \mid \texttt{ms\_app}\,\texttt{(}v_1, v_2\texttt{)} \mid$ $\texttt{ms\_fun}\ f\ \texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e_s\ \texttt{end} \mid \texttt{mc\_fun}\ f\ \texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e_c\ \texttt{end} \mid$ $\texttt{let}\ x\ \texttt{be}\ t_s\ \texttt{in}\ t'_s\ \texttt{end} \mid \texttt{mod}_\tau\ t_c \mid$ $\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\ \texttt{(}x_1\texttt{:}\tau_1\texttt{)} \Rightarrow t_s \mid \texttt{inr}\ \texttt{(}x_2\texttt{:}\tau_2\texttt{)} \Rightarrow t'_s\ \texttt{end}$
Changeable Terms	t_c	$::=$	$\texttt{write}\,\texttt{(}v\texttt{)} \mid \texttt{mc\_app}\,\texttt{(}v_1, v_2\texttt{)} \mid$ $\texttt{let}\ x\ \texttt{be}\ t_s\ \texttt{in}\ t_c\ \texttt{end} \mid \texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ t_c\ \texttt{end} \mid$ $\texttt{case}\ v\ \texttt{of}\ \texttt{inl}\ \texttt{(}x_1\texttt{:}\tau_1\texttt{)} \Rightarrow t_c \mid \texttt{inr}\ \texttt{(}x_2\texttt{:}\tau_2\texttt{)} \Rightarrow t'_c\ \texttt{end}$

Figure 13.1: The abstract syntax of SLf.

An *indexable type* accepts an injective *index* function into integers. The index function is used to determine equality. Any type can be made indexable by supplying an index function based on boxing or tagging. Since this is completely standard and well understood, we do not have a separate category for indexable types.

The terms and expressions of SLf are obtained by combining the term/expression syntax for MFL and AFL and extending the expressions with the `memo` construct. The *stable terms* and *changeable terms* corresponds to the changeable and stable expressions of the AFL language respectively. The *expressions* corresponds to the expressions of the MFL language and are partitioned into *stable expressions* and *changeable expressions* depending on whether they include a changeable or stable term.

Stable and changeable expressions are written as e_s and e_c respectively; and stable and changeable terms are written as t_s and t_c respectively. Terms evaluate independent of their contexts, as in ordinary functional programming, whereas expression are evaluated with respect to a memo table. The value of a stable expression or term is not sensitive to the modifications to the input, whereas the value of a changeable expression may be affected by them.

Stable and Changeable Terms. The stable and changeable terms of SLf are very similar to the stable and changeable expressions of the AFL language. The only difference between the two is that in SLf, all functions are memoized. It is straightforward to extend the language with non-memoized functions by enriching the stable and changeable terms with ordinary (non-memoized) stable and changeable functions.

The stable terms consists of ordinary mechanisms of functional programming, stable and changeable functions, stable function application, the $\mathtt{mod}_\tau\ t_c$ term for creating and initializing modifiables, the `let` construct for sequencing, and the `case` construct for branching on sum types.

Changeable terms are written in destination-passing style with an implicit target. The changeable term $\mathtt{write}(v)$ writes the value v into the target. The term $\mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ t_c\ \mathtt{end}$ binds the contents of the modifiable v to the variable x, then continues evaluation of t_c. A `read` is considered changeable because the contents of the modifiable on which it depends is subject to change. A changeable function itself is a stable term, but its body is a changeable expressions. The application of a changeable function is a changeable term. The sequential `let` construct allows for the inclusion of stable sub-computations in changeable mode. Case expressions with changeable branches are changeable.

Stable and Changeable Expressions. The expressions of the SLf language are similar to the expressions of the MFL language. The differences between the two are 1) the expression of SLf are divided into two depending on whether they culminate in a stable or changeable term, and 2) the expressions contain the `memo` construct.

Expression are evaluated in the context of a memo table and are divided into stable and changeable. Stable and changeable expressions are symmetric with the exception that stable terms are included in stable expressions. The `return` and `memo` constructs provide two forms of inclusion.

The expression primitives consists of `let!, let*, mcase`, `return`, and `memo` constructs. The `let!, let*, mcase`, express *strict data dependences* between the input and the output of the function. In particular the expression `let !x = v in e` expresses that the value of this expression depends on the value of x, and the expression
$\mathtt{mcase}\,v\,\mathtt{of}\,\mathtt{inl}\ (a_1\mathtt{:}\tau_1)\ \Rightarrow\ e_1 \mid \mathtt{inr}\ (a_2\mathtt{:}\tau_2)\ \Rightarrow\ e_2'\,\mathtt{end}$ expresses that the value of this expression depends on the branch taken (the tag of the sum type `v`). We therefore call these primitives *strict*.

The `return` construct provides an includes of a changeable or a stable term inside of an expression. The `memo` expression, $\mathtt{memo}\ x = x^1 = v^1, \ldots, x^n = v^n\ \mathtt{in}\ t\ \mathtt{end}$ consists of the bindings $x^1 = v_1 \ldots x^n = v^n$ and the body `t`. It is a restriction of the language (enforced by the type system) that the values $v^1, \ldots, v^n$ have modifiable type. The `memo` primitive expresses *non-strict data dependences* between the value of the expression and the bindings $x^1 \ldots x^n$. The body t of a memoized expression may depend on all the values $x_1, \ldots, x_n$ bound by its bindings. But since these values are modifiables and since the values under modifiables may change, we call these dependences *non-strict*. In particular, a computation, A, can be reused in place of another computation, B, as long as the strict dependences of A and B are the same—the non-strict dependences are *forced* to match by change propagation.

Strictness Principle. When programming in the AFL language, the programmer has to choose the strict and the non-strict dependences between the input and the output of a function. The programmer expresses the strict dependences via `let!`, `let*` and `mcase` constructs and the non-strict dependences are expressed using the `memo` construct. The principle to determine the strict and the non-strict dependences is the following *strictness principle:* A function must be memoized non-strictly on the values that it passes to other function calls and must be memoized strictly on all other values.

13.2 An Example: Quicksort

As an example, we consider the Quicksort algorithm and describe how to transform standard Quicksort into self-adjusting Quicksort using SLf.

```
datatype α list =
   nil
 | cons of (α * α list)

f:α list -> α list
fun f(l) =
 case l of
   nil => nil
 | cons(h,t) =>
    if test(h) then
      cons(h,f t)
    else
      f(t)
```

```
datatype α modlist =
  NIL
 | CONS of (α * α modlist mod)

f:α modlist -> α modlist
fun f(c) =
 case c of
   NIL => write NIL
 | CONS(h,t) =>
   if test(h) then
     write (CONS(h,mod (read t as ct in f ct end))
   else
     read t as ct in f ct end
```

```
datatype α modlist =
   NIL
 | CONS of (α * α modlist mod)

f:!α modlist -> !α modlist
mfun f(c) =
 mcase c of
   NIL => return (write NIL)
 | CONS(h,t) =>
   let !xh = h in
     memo xt = t in
       if test(xh) then
         write(CONS(xh,mod (read xt as ct in f ct end)))
     else
       read xt as ct in f ct end
     end
   end
```

Figure 13.2: The code for ordinary, adaptive, and self-adjusting filter functions `f`.

Figure 13.2 shows the ordinary, the adaptive, and the self-adjusting code for a filter function `f`. We use this simpler function to describe the main ideas behind the SLf language before we describe the Quicksort. All versions of `f` assume that there is a function `test` defined in scope. The differences between successive (left to right, top to bottom) codes are underlined.

The ordinary version takes a list as an input and walks down the list while copying elements that test positive to the output list. The adaptive version (top right) is obtained by making the input list a modifiable list, where each tail is placed inside of a modifiable, and then changing the code so that modifiables are read and written appropriately. The adaptive version is obtained from the ordinary version by inserting the underlined pieces of code. The transformation of an ordinary programs into an adaptive program is described in more detail in Section 11.2.2.

The self-adjusting version (bottom right) is obtained from the adaptive version by inserting memoization primitives. This transformation involves determining what dependences should be strict and (what dependences should be non-strict). Based on the memoization principle that a function must depend on the values that are passed down to recursive call non-strictly, we decide that the result depends on `t` non-strictly, and on the tag of the list and on the head of the list `h` strictly. The code using the SLf language expresses these dependences using the `mcase, let!` and `memo` constructs respectively. Since the `let!` construct requires that the head item be a bang type, the type of the function `f` is `f:!`α` modlist -> !`α` modlist`.

Figure 13.3 shows the complete code for the standard Quicksort algorithm and its self-adjusting version written in SLf. To avoid linear-time concatenations, `qsort` uses an accumulator to store the sorted tail of the input list. As usual the transformation consists of two steps. The first step transforms the code into the AFL language by placing modifiables in the input and inserting `mod`, `read` and `write` primitives. The second step memoizes functions by determining the strict and non-strict dependences and expressing them appropriately. For transforming the `qsort` function, we realize that the work to be done in the case of the list being empty is trivial and decide not to memoize this computation. When the list is non-empty, however, non-trivial work takes place and therefore this branch is memoized. By inspecting the code, it is easy to see that the only strict dependence is on the head item. The tail `t` and the sorted accumulator `r` are passed down to the recursive calls and are therefore memoized non-strictly via the `memo` construct.

Note that the standard version and the self-adjusting version of the Quicksort algorithm are algorithmically equivalent.

13.3 Static Semantics

The section presents the complete static semantics for the SLf language.

Each typing judgment takes place under three contexes: Δ for resources, Λ for locations, and Γ for ordinary variables. We distinguish two modes, stable and changeable. Stable terms and expressions are typed in the stable mode and changeable terms are typed in the changeable mode.

The judgment $\Delta;\Lambda;\Gamma \Vdash t : \tau$ states that t is a well formed stable term of type τ relative to Δ,Λ and Γ. The judgment $\Delta;\Lambda;\Gamma \Vdash e : \tau$ states that e is a well formed stable expression of type τ relative to Δ,Λ and Γ.

The judgment $\Delta;\Lambda;\Gamma \vDash t : \tau$ states that t is a well formed changeable term of type τ relative to Δ,Λ and Γ. The judgment $\Delta;\Lambda;\Gamma \vDash e : \tau$ states that e is a well formed changeable expression of type τ relative to Δ,Λ and Γ.

The stable and changeable expression are almost identical except for the `return` construct. Figure 13.4 shows the typing rules for values, Figure 13.5 shows the typing rules for terms, and Figure 13.6 shows the typing rules for expressions. The abstract syntax is shown in Figure 13.1.

```
datatype α list =
    nil
  | cons of (α * α list)

fil:(α-> bool)->α list->α list
fun fil test l =
let
  fun f(l) =
    case l of
      nil => nil
    | cons(h,t) =>
      if test(h) then
        cons(h,f t)
      else
        f(t)
in
  f(l)
end

qsort:  α list -> α list
fun qsort(l) =
let
  fun qs(l,r) =
    case l of
      nil =>
        r
    | cons(h,t) =>
    let
      l = fil (fn x => x<h) t
      g = fil (fn x => x>=h) t
      gs = qs(g,r)
    in
      qs(l,cons(h,gs))
    end
in
  qs(l,nil)
end
```

```
datatype α list' =
    NIL
  | CONS of (α * α list' mod)

fil:(!α -> bool) -> !α list -> !α list
fun fil' test l =
let
  fun f(c) =
    mcase c of
      NIL => return (write NIL)
    | CONS(h,t) =>
      let !x = h in
      memo y = t in
        if test(x) then
          write(CONS(x,
                mod(read y as c in f c end)))
        else
          read y as c in f c end
in
  mod (read l as c in f c end)
end

qsort:  !α list -> !α list
fun qsort(l) =
let
  fun qs(c,r) =
    mcase c of
      NIL =>
        return (read r as cr in write cr end)
    | CONS(h,t) =>
      let !x = h in
      memo y = t and z = r in
      let
        l = fil (fn k => k<x) t
        g = fil (fn k => k>=x) t
        g' = mod(read g as c in qs (c,z) end)
      in
        read l as c in qs(c,CONS(x,g')) end
      end end end
in
  mod(read l as c in qs(c,mod(write NIL)) end)
end
```

Figure 13.3: The complete code for ordinary (left) and self-adjusting (right) Quicksort.

$$\frac{}{\Delta;\Lambda;\Gamma \Vdash n : \texttt{int}}\ \textbf{(number)}$$

$$\frac{(\Delta(a)=\tau)}{\Delta;\Lambda;\Gamma \Vdash a : \tau}\ \textbf{(resource)} \qquad \frac{(\Lambda(l)=\tau)}{\Delta;\Lambda;\Gamma \Vdash l : \tau\,\texttt{mod}}\ \textbf{(location)} \qquad \frac{(\Gamma(x)=\tau)}{\Delta;\Lambda;\Gamma \Vdash x : \tau}\ \textbf{(variable)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_1 : \tau_1 \quad \Delta;\Lambda;\Gamma \Vdash v_2 : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{(}v_1\texttt{,}v_2\texttt{)} : \tau_1 \times \tau_2}\ \textbf{(product)} \qquad \frac{\emptyset;\Lambda;\Gamma \Vdash v : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{!}\,v : \texttt{!}\,\tau}\ \textbf{(bang)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1}{\Delta;\Lambda;\Gamma \Vdash \texttt{inl}_{\tau_1+\tau_2} v : \tau_1 + \tau_2}\ \textbf{(sum/inl)} \qquad \frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{inr}_{\tau_1+\tau_2} v : \tau_1 + \tau_2}\ \textbf{(sum/inr)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma, f{:}\tau_1 \xrightarrow{\mathsf{ms}} \tau_2; \Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{ms\_fun}_m\, f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\;\texttt{is}\;e_s\;\texttt{end} : \tau_1 \xrightarrow{\mathsf{ms}} \tau_2}\ \textbf{(stable fun)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma, f{:}\tau_1 \xrightarrow{\mathsf{c}} \tau_2; \Vdash e_c : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{mc\_fun}_m\, f\,\texttt{(}a\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\;\texttt{is}\;e_c\;\texttt{end} : \tau_1 \xrightarrow{\mathsf{c}} \tau_2}\ \textbf{(changeable fun)}$$

Figure 13.4: Typing of values.

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_i : \tau_i \quad (1 \le i \le n) \quad \vdash_o o : (\tau_1,\ldots,\tau_n)\,\tau}{\Delta;\Lambda;\Gamma \Vdash o\,(v_1,\ldots,v_n)\ : \tau}\ \textbf{(primitives)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma,\ f{:}\tau_1 \xrightarrow{\mathsf{s}} \tau_2 \Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \mathtt{ms\_fun}\, f\ (a : \tau_1) : \tau_2\ \mathtt{is}\ e_s\ \mathtt{end} : \tau_1 \xrightarrow{\mathsf{s}} \tau_2}\ \textbf{(stable mfun)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma,\ f{:}\tau_1 \xrightarrow{\mathsf{c}} \tau_2 \vDash e_c : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \mathtt{mc\_fun}\, f\ (a : \tau_1) : \tau_2\ \mathtt{is}\ e_c\ \mathtt{end} : \tau_1 \xrightarrow{\mathsf{c}} \tau_2}\ \textbf{(changeable mfun)}$$

$$\frac{\Lambda;\Gamma \Vdash v_1 : (\tau_1 \xrightarrow{\mathsf{s}} \tau_2) \quad \Lambda;\Gamma \Vdash v_2 : \tau_1}{\Lambda;\Gamma \Vdash \mathtt{s\_app}\,(v_1, v_2)\ : \tau_2}\ \textbf{(stable apply)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Lambda;\Gamma, x : \tau_1 \Vdash t'_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \mathtt{let}\ x\ \mathtt{be}\ t_s\ \mathtt{in}\ t'_s\ \mathtt{end} : \tau_2}\ \textbf{(let)}$$

$$\frac{\Delta;\Lambda;\Gamma \vDash t_c : \tau}{\Delta;\Lambda;\Gamma \Vdash \mathtt{mod}_\tau\ t_c : \tau\ \mathtt{mod}}\ \textbf{(mod)}$$

$$\frac{\begin{array}{rcl} \Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta;\Lambda;\Gamma, x_1{:}\tau_1 & \Vdash & t_s : \tau \\ \Delta;\Lambda;\Gamma, x_2{:}\tau_2 & \Vdash & t'_s : \tau \end{array}}{\Delta;\Lambda;\Gamma \Vdash \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}\ (x_1 : \tau_1)\ \Rightarrow\ t_s \mid \mathtt{inr}\ (x_2 : \tau_2)\ \Rightarrow\ t'_s\ \mathtt{end} : \tau}\ \textbf{(case)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau}{\Delta;\Lambda;\Gamma \vDash \mathtt{write}\,(v)\ : \tau}\ \textbf{(write)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v_1 : (\tau_1 \xrightarrow{\mathsf{c}} \tau_2) \quad \Delta;\Lambda;\Gamma \Vdash v_2 : \tau_1}{\Delta;\Lambda;\Gamma \vDash \mathtt{c\_app}\,(v_1, v_2)\ : \tau_2}\ \textbf{(apply)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Delta;\Lambda;\Gamma, x : \tau_1 \vDash t_c : \tau_2}{\Delta;\Lambda;\Gamma \vDash \mathtt{let}\ x\ \mathtt{be}\ t_s\ \mathtt{in}\ t_c\ \mathtt{end} : \tau_2}\ \textbf{(let)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1\ \mathtt{mod} \quad \Delta;\Lambda;\Gamma, x : \tau_1 \vDash t_c : \tau_2}{\Delta;\Lambda;\Gamma \vDash \mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ t_c\ \mathtt{end} : \tau_2}\ \textbf{(read)}$$

$$\frac{\begin{array}{rcl} \Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta;\Lambda;\Gamma, x_1{:}\tau_1 & \vDash & t_c : \tau \\ \Delta;\Lambda;\Gamma, x_2{:}\tau_2 & \vDash & t'_c : \tau \end{array}}{\Delta;\Lambda;\Gamma \Vdash \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}\ (x_1 : \tau_1)\ \Rightarrow\ t_c \mid \mathtt{inr}\ (x_2 : \tau_2)\ \Rightarrow\ t'_c\ \mathtt{end} : \tau}\ \textbf{(case)}$$

Figure 13.5: Typing of stable (top) and changeable (bottom) terms.

$$\frac{\emptyset;\Lambda;\Gamma \Vdash t_s : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{return}\,(t_s) : \tau}\ \textbf{(return)}$$

$$\frac{\begin{array}{c}\Delta;\Lambda;\Delta \Vdash v_1 : \tau_1 \quad \ldots \quad \Delta;\Lambda;\Delta \Vdash v_n : \tau_n \\ \Delta;\emptyset;\Delta, x_1 : \tau_1,\ldots,x_n : \tau_n \Vdash t_s : \tau\end{array}}{\Delta;\Lambda;\Gamma \Vdash \texttt{memo}\, x_1 = v_1 \ldots x_n = v_n \,\texttt{in}\, t_s \,\texttt{end} : \tau}\ \textbf{(memo)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Delta, a{:}\tau_1;\Lambda;\Gamma \vDash e_c : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\, a\texttt{:}\tau_1 \,\texttt{be}\, t_s \,\texttt{in}\, e_c \,\texttt{end} : \tau_2}\ \textbf{(let)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \,!\,\tau_1 \quad \Delta;\Lambda;\Gamma, x{:}\tau_1 \Vdash e_s : \tau_2}{\Delta;\Lambda;\Gamma \Vdash \texttt{let !}\, x\texttt{:}\tau_1 \,\texttt{be}\, v \,\texttt{in}\, e_s \,\texttt{end} : \tau_2}\ \textbf{(let!)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1 \times \tau_2 \quad \Delta, a_1{:}\tau_1, a_2{:}\tau_2;\Lambda;\Gamma \Vdash e_s : \tau}{\Delta;\Lambda;\Gamma \Vdash \texttt{let}\, a_1{:}\tau_1 \times a_2{:}\tau_2 \,\texttt{be}\, v \,\texttt{in}\, e_s \,\texttt{end} : \tau}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta, a_1{:}\tau_1;\Lambda;\Gamma & \Vdash & e_s : \tau \\ \Delta, a_2{:}\tau_2;\Lambda;\Gamma & \Vdash & e'_s : \tau\end{array}}{\Delta;\Lambda;\Gamma \Vdash \texttt{mcase}\, v \,\texttt{of inl}\,(a_1\texttt{:}\tau_1) \Rightarrow e_s \mid \texttt{inr}\,(a_2\texttt{:}\tau_2) \Rightarrow e'_s \,\texttt{end} : \tau}\ \textbf{(case)}$$

$$\frac{\emptyset;\Lambda;\Gamma \vDash t_c : \tau}{\Delta;\Lambda;\Gamma \vDash \texttt{return}\,(t_c) : \tau}\ \textbf{(return)}$$

$$\frac{\begin{array}{c}\Delta;\Lambda;\Delta \Vdash v_1 : \tau_1 \quad \ldots \quad \Delta;\Lambda;\Delta \Vdash v_n : \tau_n \\ \Delta;\emptyset;\Delta, x_1 : \tau_1,\ldots,x_n : \tau_n \vDash t_c : \tau\end{array}}{\Delta;\Lambda;\Gamma \vDash \texttt{memo}\, x_1 = v_1 \ldots x_n = v_n \,\texttt{in}\, t_c \,\texttt{end} : \tau}\ \textbf{(memo)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash t_s : \tau_1 \quad \Delta, a{:}\tau_1;\Lambda;\Gamma \vDash e_c : \tau_2}{\Delta;\Lambda;\Gamma \vDash \texttt{let}\, a\texttt{:}\tau_1 \,\texttt{be}\, t_s \,\texttt{in}\, e_c \,\texttt{end} : \tau_2}\ \textbf{(let)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \,!\,\tau \quad \Delta;\Lambda;\Gamma, x{:}\tau \vDash e_c : \tau}{\Delta;\Lambda;\Gamma \vDash \texttt{let !}\, x\texttt{:}\tau \,\texttt{be}\, v \,\texttt{in}\, e_c \,\texttt{end} : \tau}\ \textbf{(let!)}$$

$$\frac{\Delta;\Lambda;\Gamma \Vdash v : \tau_1 \times \tau_2 \quad \Delta, a_1{:}\tau_1, a_2{:}\tau_2;\Lambda;\Gamma \vDash e_c : \tau}{\Delta;\Lambda;\Gamma \vDash \texttt{let}\, a_1{:}\tau_1 \times a_2{:}\tau_2 \,\texttt{be}\, v \,\texttt{in}\, e_c \,\texttt{end} : \tau}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \Vdash & v : \tau_1 + \tau_2 \\ \Delta, a_1{:}\tau_1;\Lambda;\Gamma & \vDash & e_c : \tau \\ \Delta, a_2{:}\tau_2;\Lambda;\Gamma & \vDash & e'_c : \tau\end{array}}{\Delta;\Lambda;\Gamma \vDash \texttt{mcase}\, v \,\texttt{of inl}\,(a_1\texttt{:}\tau_1) \Rightarrow e_c \mid \texttt{inr}\,(a_2\texttt{:}\tau_2) \Rightarrow e'_c \,\texttt{end} : \tau}\ \textbf{(case)}$$

Figure 13.6: Typing of stable (top) and changeable (bottom) expressions.

13.4 Dynamic Semantics

The dynamic semantics consists of four separate evaluation judgments corresponding to stable and changeable terms and stable and changeable expressions. All evaluation judgments take place with respect to a state $\sigma = (\alpha, \mu, \chi, \mathtt{T})$ consisting of a location store α, a memoized-function identifier store μ, a set of changed locations χ, and a re-use trace $\mathtt{T}$. The location store is where modifiables are allocated, the memoized-function identifier store dispenses unique identifiers for memoized functions that are used for memo lookups. The set of changed location contains the locations that has been changed since the previous execution. The re-use trace is the trace available for re-use by the memo functions. Re-use trace is provided by change propagation and is empty in the initial evaluation.

Traces

Evaluation of a term or an expression records its activity in a *trace*. Like terms and expressions, traces are divided into stable and changeable. The abstract syntax of traces is given by the following grammar, where $\mathtt{T}$ stands for a trace, $\mathtt{T}_s$ stands for a stable trace and $\mathtt{T}_c$ stands for a changeable trace.

$$\begin{array}{lcl}
\mathtt{T} & ::= & \mathtt{T}_s \mid \mathtt{T}_c \\
\mathtt{T}_s & ::= & \epsilon \mid \langle \mathtt{T}_c \rangle_{l:\tau} \mid \mathtt{T}_s \,;\, \mathtt{T}_s \mid \{\, \mathtt{T}_s \,\}^{m:\beta}_{(v,(l_1,\ldots,l_n))} \\
\mathtt{T}_c & ::= & \mathtt{W}_\tau \mid R^{x.t}_l(\mathtt{T}_c) \mid \mathtt{T}_s \,;\, \mathtt{T}_c \mid \{\, \mathtt{T}_c \,\}^{m:\beta}_{(l_1,\ldots,l_n)}
\end{array}$$

When writing traces, we adopt the convention that "**;**" is right-associative.

This definition of traces extends the trace notion of the AFL language with traces for memoized stable and changeable terms.

A stable trace records the sequence of allocations of modifiables that arise during the evaluation of a stable term or expression. The trace $\langle \mathtt{T}_c \rangle_{l:\tau}$ records the allocation of the modifiable, l, its type, τ, and the trace of the initialization code for l. The trace $\mathtt{T}_s \,;\, \mathtt{T}'_s$ results from evaluation of a `let` expression in stable mode, the first trace resulting from the bound expression, the second from its body. The trace $\{\, \mathtt{T}_s \,\}^{m:\beta}_{(v,(l_1,\ldots,l_n))}$ arises from the evaluation of a stable memoized function application; m is the identifier, β is the branch expressing the input-output dependences, the value v is the result of the evaluation, $l_1 \ldots l_n$ are the local modifiables, and $\mathtt{T}_s$ is the trace of the body of the function.

A changeable trace has one of four forms. A write, $\mathtt{W}_\tau$, records the storage of a value of type τ in the target. A sequence $\mathtt{T}_s \,;\, \mathtt{T}_c$ records the evaluation of a `let` expression in changeable mode, with $\mathtt{T}_s$ corresponding to the bound stable expression, and $\mathtt{T}_c$ corresponding to its body. A read $R^{x.t}_l(\mathtt{T}_c)$ trace specifies the location read, l, the context of use of its value, $x.e$, and the trace, $\mathtt{T}_c$, of the remainder of evaluation with the scope of that read. This records the dependency of the target on the value of the location read. The memoized changeable trace $\{\, \mathtt{T}_c \,\}^{m:\beta}_{(l_1,\ldots,l_n)}$ arises from the evaluation of a changeable memoized function; m is the identifier, β is the branch expressing the input-output dependences, $l_1 \ldots l_n$ are the free modifiables, and $\mathtt{T}_c$ is the trace of the body of the function. Since changeable function write their result to the store, the trace has no result value.

Branches

Expression evaluation takes place in the context of a memo branch. The incremental evaluation constructs (`let!`, `let*`, `mcase`) create a branch, denoted β. A *branch* is a list of *events* corresponding to "choice points" in the evaluation of an expression.

$$\begin{array}{llll} \textit{Event} & \varepsilon & ::= & !v \mid \texttt{inl} \mid \texttt{inr} \\ \textit{Branch} & \beta & ::= & \bullet \mid \varepsilon \cdot \beta \end{array}$$

This notion of branches is identical to the branches of the MFL language. The branch and the identifier m is used by the `memo` construct to lookup the re-use trace for a match. If a match is found, the result is returned and the body of `memo` is skipped. Otherwise, the body is executed and the computation is memoized in the form of a memo trace.

Term evaluation

The term evaluation judgments are similar to that of the AFL language. As with AFL, term evaluation consists of changeable and stable evaluation forms. The judgment $\sigma, t_s \Downarrow^{\mathsf{t}}_{\mathsf{s}} v, \sigma', \mathtt{T}_s$ states that evaluation of the stable term t_s with respect to the state σ yields value v, state σ', and the trace $\mathtt{T}_s$. The judgment $\sigma, l \leftarrow t_c \Downarrow^{\mathsf{t}}_{\mathsf{c}} \sigma', \mathtt{T}_c$ states that evaluation of the changeable term t_c with respect to the state σ writes to destination l and yields the state σ', and the trace $\mathtt{T}_c$.

Figures 13.7 and 13.8 show the evaluation rules for stable and changeable terms. Memoized stable and memoized changeable functions are evaluated into values by generating a new memoized function identifier m. Changeable and stable applications evaluate some expression in the context of an identifier m and a branch β. As in selective memoization, the branch collects the precise dependencies between the input and the output. For stable applications the branch starts out empty. For changeable applications the branch is initialized to the target—since a changeable expressions writes to its target, the target must be identical for result re-use to take place.

Expression Evaluation

The expression evaluation judgments consists of changeable and stable evaluation forms. The evaluation of changeable and stable expressions are very similar. The only difference is that the changeable evaluation takes place with respect to a destination to which the result is written.

Figure 13.9 shows the rules for stable-expression evaluation and Figure 13.10 shows the rules for changeable-expression evaluation. The evaluation $\sigma, m\,{:}\,\beta, e_s \ \Downarrow^{\mathsf{e}}_{\mathsf{s}}\ v, \sigma', \mathtt{T}_s$ states that the evaluation of stable expression e_s in the context of the state σ with memo function identifier m and branch β yields the value v, the state σ', and the trace $\mathtt{T}_s$. The evaluation $\sigma, m\,{:}\,\beta, l \leftarrow e_c \ \Downarrow^{\mathsf{e}}_{\mathsf{c}}\ \sigma', \mathtt{T}_c$ states that the evaluation of changeable expression e_c in the context of the state σ, with memo function identifier m and branch β write to location l and yields the state σ' and the trace $\mathtt{T}_c$.

The evaluation rules for the `let!, let*` and `mcase` expressions are similar to those of the MFL language. The `let!` expression binds the underlying value of the bang value to a variable, extends the branch with that value, and evaluates its body. The `let*` construct binds each component of a pair to a

$$\frac{}{\sigma, v \Downarrow^{\mathsf{t}}_{\mathsf{s}} v, \sigma, \varepsilon}\ \textbf{(value)}$$

$$\frac{}{\sigma, o\,(v_1, \ldots, v_n)\ \Downarrow^{\mathsf{t}}_{\mathsf{s}}\ \mathtt{app}\,(o, (v_1, \ldots, v_n)), \sigma, \varepsilon}\ \textbf{(primitive)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ \sigma' = (\alpha, \mu \cup \{m\}, \chi, \mathtt{T}),\ m \notin \mathrm{dom}(\mu)\end{array}}{\sigma, \mathtt{ms\_fun}\, f\ (a\mathtt{:}\tau_1)\mathtt{:}\tau_2\, \mathtt{is}\, e_s\, \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{s}}\ \mathtt{ms\_fun}_m\, f\,(a\mathtt{:}\tau_1)\mathtt{:}\tau_2\, \mathtt{is}\, e_s\, \mathtt{end}, \sigma', \varepsilon}\ \textbf{(stable mfun)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ \sigma' = (\alpha, \mu \cup \{m\}, \chi, \mathtt{T}),\ m \notin \mathrm{dom}(\mu)\end{array}}{\sigma, \mathtt{mc\_fun}\, f\ (a\mathtt{:}\tau_1)\mathtt{:}\tau_2\, \mathtt{is}\, e_c\, \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{s}}\ \mathtt{mc\_fun}_m\, f\,(a\mathtt{:}\tau_1)\mathtt{:}\tau_2\, \mathtt{is}\, e_c\, \mathtt{end}, \sigma, \varepsilon}\ \textbf{(changeable mfun)}$$

$$\frac{\begin{array}{c}(v_1 = \mathtt{ms\_fun}\, f\ (a\mathtt{:}\tau_1)\mathtt{:}\tau_2\, \mathtt{is}\, e_s\, \mathtt{end}) \\ \sigma, m\mathtt{:}\varepsilon, [v_1/f, v_2/a]\ e_s\ \ \Downarrow^{\mathsf{e}}_{\mathsf{s}}\ \ v, \sigma', \mathtt{T}_s\end{array}}{\sigma, \mathtt{ms\_app}\,(v_1, v_2)\ \Downarrow^{\mathsf{t}}_{\mathsf{s}}\ v, \sigma' \mathtt{T}_s}\ \textbf{(stable apply)}$$

$$\frac{\begin{array}{lcl}\sigma, t_s & \Downarrow^{\mathsf{t}}_{\mathsf{s}} & v_1, \sigma', \mathtt{T}_s \\ \sigma', [v_1/x]\, t'_s & \Downarrow^{\mathsf{t}}_{\mathsf{s}} & v_2, \sigma'', \mathtt{T}'_s\end{array}}{\sigma, \mathtt{let}\ x\ \mathtt{be}\ t_s\ \mathtt{in}\ t'_s\ \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{s}}\ v_2, \sigma'', (\mathtt{T}_s\ ;\ \mathtt{T}'_s)}\ \textbf{(let)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ \alpha' = \alpha[l \mapsto \Box],\ l \notin \mathrm{dom}(\alpha) \\ (\alpha', \mu, \chi, \mathtt{T}), l \leftarrow t_c\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma', \mathtt{T}_c\end{array}}{\sigma, \mathtt{mod}_\tau\ t_c\ \Downarrow^{\mathsf{t}}_{\mathsf{s}}\ l, \sigma', \langle \mathtt{T}_c \rangle_{l:\tau}}\ \textbf{(mod)}$$

$$\frac{\sigma, [v/x_1]t_s\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ v', \sigma', \mathtt{T}_s}{\sigma, \mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (x_1\mathtt{:}\tau_1)\ \Rightarrow\ t_s \mid \mathtt{inr}\ (x_2\mathtt{:}\tau_2)\ \Rightarrow\ t'_s\ \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ v', \sigma', \mathtt{T}_s}\ \textbf{(case/inl)}$$

$$\frac{\sigma, [v/x_2]t'_s\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ v', \sigma', \mathtt{T}_s}{\sigma, \mathtt{case}\ \mathtt{inr}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (x_1\mathtt{:}\tau_1)\ \Rightarrow\ t_s \mid \mathtt{inr}\ (x_2\mathtt{:}\tau_2)\ \Rightarrow\ t'_s\ \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ v', \sigma', \mathtt{T}_s}\ \textbf{(case/inr)}$$

Figure 13.7: Evaluation of stable terms.

resource and evaluates its body. The `mcase` performs pattern matching on sum types. It binds the underlying value of the sum to a resource, extends the branch with the tag of the sum, and evaluates the appropriate branch. The `return` expression evaluates its body and returns—no memo look ups are performed. The `let` expression provides for sequencing by evaluating a stable term and binding its value to a resource. The evaluation of these expressions are symmetric for changeable and stable expressions.

The key construct in the SLf language is the `memo` construct. Evaluating a memo expression in the form `memo` $x_1 = l_1 \ldots x_n = l_n$ `in` t `end`, involves first checking the memo table for a memo look up with the current branch. If a memo miss take place, then the locations $l_1, \ldots, l_n$ are copied into fresh locations

$$\frac{\begin{array}{c}(\alpha,\mu,\chi,\mathtt{T})=\sigma\\ \sigma'=(\alpha[l\leftarrow v],\mu,\chi,\mathtt{T})\end{array}}{\sigma,l\leftarrow\mathtt{write}\,(v)\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',\mathtt{W}_\tau}\ \textbf{(write)}$$

$$\frac{\begin{array}{c}(v_1=\mathtt{mc\_fun}\,f\ (a\,\mathtt{:}\,\tau_2)\,\mathtt{:}\,\tau\ \mathtt{is}\ e_c\ \mathtt{end})\\ \sigma,m\,\mathtt{:}\,!\ l,l\leftarrow[v_1/f,v_2/a]\ e_c\ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}}\ \ \sigma',\mathtt{T}\end{array}}{\sigma,l\leftarrow\mathtt{mc\_app}\,(v_1,v_2)\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',\mathtt{T}}\ \textbf{(memo apply)}$$

$$\frac{\begin{array}{lll}\sigma,t_s & \Downarrow^{\mathsf{t}}_{\mathsf{s}} & v_1,\sigma',\mathtt{T}_s\\ \sigma',l\leftarrow[v_1/x]t_c & \Downarrow^{\mathsf{t}}_{\mathsf{c}} & \sigma'',\mathtt{T}_c\end{array}}{\sigma,l\leftarrow\mathtt{let}\ x\ \mathtt{be}\ t_s\ \mathtt{in}\ t_c\ \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma'',(\mathtt{T}_s\ ;\ \mathtt{T}_c)}\ \textbf{(let)}$$

$$\frac{\sigma,l'\leftarrow[\sigma(l)/x]\ t_c\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,l'\leftarrow\mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ t_c\ \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',R_l^{x.t_c}(\mathtt{T}_c)}\ \textbf{(read)}$$

$$\frac{\sigma,l\leftarrow[v/x_1]t_c\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,l\leftarrow\mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2}v\ \mathtt{of}\ \mathtt{inl}\ (x_1\,\mathtt{:}\,\tau_1)\ \Rightarrow\ t_c\ |\ \mathtt{inr}\ (x_2\,\mathtt{:}\,\tau_2)\ \Rightarrow\ t'_c\ \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',\mathtt{T}_c}\ \textbf{(case/inl)}$$

$$\frac{\sigma,l\leftarrow[v/x_2]t'_c\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',\mathtt{T}_c}{\sigma,l\leftarrow\mathtt{case}\ \mathtt{inr}_{\tau_1+\tau_2}v\ \mathtt{of}\ \mathtt{inl}\ (x_1\,\mathtt{:}\,\tau_1)\ \Rightarrow\ t_c\ |\ \mathtt{inr}\ (x_2\,\mathtt{:}\,\tau_2)\ \Rightarrow\ t'_c\ \mathtt{end}\ \Downarrow^{\mathsf{t}}_{\mathsf{c}}\ \sigma',\mathtt{T}_c}\ \textbf{(case/inr)}$$

Figure 13.8: Evaluation of changeable terms.

$l'_1,\ldots,l'_n$, called *local modifiables*, and the body of the memo `t` is evaluated after substituting the fresh location in place of $x_1,\ldots,x_n$. The trace obtained by the evaluation of t is then extended with the trace representing the copy operations and the result is returned.

When a match is found in the memo, the values of free modifiables $l_1\ldots l_n$ are copied to the local modifiables $l'_1\ldots l'_n$ of the re-used trace and a change propagation is performed to update the re-used trace. The trace returned by change propagation forms the result trace together with the trace of the copies. Since a result is found in the memo, the body of the `memo` is skipped.

The key property of the evaluation of `memo` construct is that memo look ups do not take into account the values being bound by the memo construct. Consequently, the construct allows memoizing computations that have free variables. The re-used trace may therefore belong to an evaluation of the body of `memo` with a different set of values. The technique described here ensures correctness by copying the free variables (which are known to be modifiables) to modifiables that are local to the memoized computation. When a computation is re-used these local modifiables are updated with the new values of the free variables and a change propagation is performed. The change propagation ensures that the re-used computation is adjusted according to the values of the free variables.

The memo look ups are performed using the `find` relation; Figure 13.11 shows the definition of `find`. The `find` relation seeks for a memoized result in the re-use trace whose identifier and branch matches m (current memo table) and β (current branch). If a result is not found, then `find` returns an empty trace. If a result is found, then `find` returns the trace found and the uninspected tail of the re-use trace.

Change Propagation

Based on the dynamic semantics of SLf, we present an operational semantics for the change-propagation algorithm. The change-propagation algorithm extends the memoized change-propagation algorithm (Section 5.4) by providing support for non-strict dependences.

Given a trace, a state ς, and a set of changed locations χ, the algorithm scans through the trace as it seeks for reads of changed locations. When such a read is found, the body of the read is re-evaluated to obtain a revised trace. Crucial point is that the re-evaluation of a read re-uses the trace of that read. Since re-evaluation can change the value of the target of the re-evaluated read, the target is added to the set of changed locations.

Figure 13.12 shows the rules for change propagation. The change propagation algorithm is given by these two judgments:

1. *Stable propagation*: $\varsigma, \chi, \mathtt{T}_s \overset{\mathsf{s}}{\curvearrowright} \mathtt{T}'_s, \chi', \varsigma'$

2. *Changeable propagation*: $\varsigma, \chi, l \leftarrow \mathtt{T}_c \overset{\mathsf{c}}{\curvearrowright} \mathtt{T}'_c, \chi', \varsigma'$

These judgments define the change-propagation for a stable trace, $\mathtt{T}_s$ (respectively, changeable trace, $\mathtt{T}_c$), with respect to a a set of changed locations χ, and state $\varsigma = (\alpha, \mu)$ consisting of a location store α, and function identifier store μ. For changeable propagation a target location, l, is maintained as in the changeable evaluation mode of SLf.

Given a trace, change propagation mimics the evaluation rule of SLf that originally generated that trace. To stress this correspondence, each rule is marked with the name of the evaluation rule to which it corresponds. For example, the propagation rule for the trace $\mathtt{T}_s \mathbin{;} \mathtt{T}'_s$ mimics the `let` rule of the stable mode that gives rise to this trace.

13.5 Type Safety

The type system of the SLf language is a simple extension of the product of the AFL and MFL languages. The type safety of SLf can therefore be easily proven based on the type safety of the AFL (Section 11.4) and the MFL languages (Section 12.4.4).

13.6 Correctness

It is possible to give a correctness theorem for change propagation in the SLf language. The proof would rely on the correctness of the AFL, and the MFL languages. Our attempts to proving this theorem, however, shows that the proof is tractable but technically challenging. The challenges are due to the large number of cases, and due to the interaction between memoization and side effects (writes).

13.7 Performance

One of the key properties of the SLf language is that the performance of change propagation for programs written in SLf can be determined analytically. There are two approaches to this.

The first approach is to use the algorithmic techniques presented in the first part of this thesis. The idea is to transform the SLf program into the imperative model discussed in Part I, and analyze its performance using trace-stability techniques (Part II). The transformation of a program from the SLf language into the imperative model is relatively straightforward, especially because the imperative model is more expressive than the SLf language.

The second approach relies on the trace model described in this chapter. If the traces of a program satisfies certain *monotonicity* properties, then performance under change propagation can be analyzed by comparing the sets of evaluated `memo` expressions. Let P be a self-adjusting program written in the SLf language. We say that two traces T and T' of P are monotone, if T and T' of P satisfy the following properties. For the definition, we say that a trace M_1 is *contained* in another trace M_2 if M_1 is part of M_2.

1. **All evaluations of a `memo` construct is uniquely identified by its branch:** For any (stable or changeable) memo trace $\{\_\}^{m\,:\,\beta}_{(\_)}$ in $T(T')$, there is no other trace $\{\_\}^{m'\,:\,\beta'}_{(\_)}$ such that $m = m'$ and $\beta = \beta'$.
2. **Evaluation order and containment relation of two memo constructs are preserved:** Suppose that $M_1 = \{\_\}^{m_1\,:\,\beta_1}_{(\_)}$ and $M_2 = \{\_\}^{m_2\,:\,\beta_2}_{(\_)}$ are two memo traces in T. Suppose also that there exists memo traces $M_1' = \{\_\}^{m_1'\,:\,\beta_1'}_{(\_)}$ and $M_2' = \{\_\}^{m_2'\,:\,\beta_2'}_{(\_)}$ in T' such that $m_1 = m_1'$, $m_2 = m_2'$, and $\beta_1 = \beta_1'$, $\beta_2 = \beta_2'$. The following properties are satisfied
   (a) If M_1 comes before M_2 in T, then M_1' comes before M_2' in T'.
   (b) If M_1 is contained in M_2, then M_1' is contained in M_2'.

We state the following performance theorem that shows that the time for change propagation can be measured by comparing the trace of the program with inputs before and after the change. For the theorem, we defined the *set of memo traces* of a trace T as the set of all memo traces in T where each memo trace is assigned a weight corresponds to the evaluation time (number of derivations) of the `memo` construct that creates the memo trace.

Theorem 87
Let P be a self-adjusting program written in the SLf language and suppose that all functions are memoized such that the terms within the `return` expressions take no more than constant time. Consider evaluating P on some input I and changing I to I' and performing a change propagation. Let T and T' be the traces of P with I and I' and suppose that T and T' are respectively monotone. Let M and M' be the set of memo traces of T and T' and define the distance d between T and T' as $d = \sum_{m \in M \setminus M'} w(m) + \sum_{m' \in M' \setminus M} w(m')$, where $w(m)$ is the weight of a memo trace. The time for change propagation is no more than $O(d \log d)$.

We do not present the full proof for this theorem here, because it is a restatement of the trace-stability theorem proven in Chapter 8. As with the trace stability theorem, the key idea is behind the proof is to show that when the traces are monotone, all memo traces that are common in T and T' will be re-used and therefore will contribute no cost to change propagation.

$$\frac{\sigma, t_s \ \Downarrow^{\mathsf{t}}_{\mathsf{s}} \ v, \sigma', \mathtt{T}_s}{\sigma, m\,{:}\,\beta, \mathtt{return}\,(t_s) \quad \Downarrow^{\mathsf{e}}_{\mathsf{s}} \quad v, \sigma', \mathtt{T}_s} \ \textbf{(return)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ m : \beta, \mathtt{T} \overset{find}{\rightsquigarrow} \epsilon, \_ \\ \alpha' = \alpha[l'_1 \leftarrow \alpha[l_1]] \ldots, [l'_n \leftarrow \alpha[l_n]],\ l'_i \notin \mathrm{dom}(\alpha),\ l'_i \neq l'_j \\ (\alpha', \mu, \chi, \mathtt{T}), [l'_1/x_1, \ldots, l'_n/x_n]\ t_s \ \Downarrow^{\mathsf{t}}_{\mathsf{s}} \ v, \sigma', \mathtt{T}_s \\ \mathtt{T}'_s = \langle R^{x.\mathtt{write}(x)}_{l_1} \mathtt{W}_{\tau_1} \rangle_{l'_1:\tau_1} \,;\, \ldots \,;\, \langle R^{x.\mathtt{write}(x)}_{l_n} \mathtt{W}_{\tau_n} \rangle_{l'_n:\tau_n}\end{array}}{\sigma, m\,{:}\,\beta, \mathtt{memo}\ x_1 = l_1 \ldots x_n = l_n\ \mathtt{in}\ t_s\ \mathtt{end} \ \Downarrow^{\mathsf{e}}_{\mathsf{s}} \ v, \sigma', \left(\mathtt{T}'_s \,;\, \{\, \mathtt{T}_s \,\}^{m:\beta}_{(v,(l'_1,\ldots,l'_n))}\right)} \ \textbf{(memo/not found)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ m : \beta, \mathtt{T} \overset{find}{\rightsquigarrow} \{\, \mathtt{T}_s \,\}^{m:\beta}_{(v,(l'_1,\ldots l'_n))}, \mathtt{T}' \\ \alpha' = \alpha[l'_1 \leftarrow \alpha[l_1]] \ldots [l'_n \leftarrow \alpha[l_n]] \\ (\alpha', \mu), \chi \cup \{l'_1, \ldots, l'_n\}, \{\, \mathtt{T}_s \,\}^{m:\beta}_{(v,(l'_1,\ldots l'_n))} \ \overset{\mathsf{s}}{\curvearrowright} \ \mathtt{T}'_s, \chi', (\alpha'', \mu') \\ \mathtt{T}''_s = \langle R^{x.\mathtt{write}(x)}_{l_1} \mathtt{W}_{\tau_1} \rangle_{l'_1:\tau_1} \,;\, \ldots \,;\, \langle R^{x.\mathtt{write}(x)}_{l_n} \mathtt{W}_{\tau_n} \rangle_{l'_n:\tau_n}\end{array}}{\sigma, m\,{:}\,\beta, \mathtt{memo}\ x_1 = l_1 \ldots x_n = l_n\ \mathtt{in}\ t_s\ \mathtt{end} \ \Downarrow^{\mathsf{e}}_{\mathsf{s}} \ v, (\alpha'', \mu', \chi', \mathtt{T}'), (\mathtt{T}''_s \,;\, \mathtt{T}'_s)} \ \textbf{(memo/found)}$$

$$\frac{\begin{array}{lcl}\sigma, t_s & \Downarrow^{\mathsf{t}}_{\mathsf{s}} & v, \sigma', \mathtt{T}_s \\ \sigma', m\,{:}\,\beta, [v/a]e_s & \Downarrow^{\mathsf{e}}_{\mathsf{s}} & v', \sigma'', \mathtt{T}'_s\end{array}}{\sigma, m\,{:}\,\beta, \mathtt{let}\,a : \tau\,\mathtt{be}\,t_s\,\mathtt{in}\,e_s\,\mathtt{end} \ \Downarrow^{\mathsf{e}}_{\mathsf{s}} \ v', \sigma'', \mathtt{T}_s \,;\, \mathtt{T}'_s} \ \textbf{(let)}$$

$$\frac{\sigma, m\,{:}\,!\ v \cdot \beta, [v/x]e_s \quad \Downarrow^{\mathsf{e}}_{\mathsf{s}} \quad v', \sigma', \mathtt{T}_s}{\sigma, m\,{:}\,\beta, \mathtt{let}\ !\ x : \tau\,\mathtt{be}\ !\ v\,\mathtt{in}\,e_s\,\mathtt{end} \ \Downarrow^{\mathsf{e}}_{\mathsf{s}} \ v', \sigma', \mathtt{T}_s} \ \textbf{(let!)}$$

$$\frac{\sigma, m\,{:}\,\beta, [v_1/a_1, v_2/a_2]e_s \quad \Downarrow^{\mathsf{e}}_{\mathsf{s}} \quad v, \sigma', \mathtt{T}_s}{\sigma, m\,{:}\,\beta, \mathtt{let}\,a_1 \times a_2\,\mathtt{be}\,v_1 \times v_2\,\mathtt{in}\,e_s\,\mathtt{end} \ \Downarrow^{\mathsf{e}}_{\mathsf{s}} \ v, \sigma', \mathtt{T}_s} \ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma, m\,{:}\,\mathtt{inl} \cdot \beta, [v/a_1]e_s \quad \Downarrow^{\mathsf{e}}_{\mathsf{s}} \quad v', \sigma', \mathtt{T}_s}{\sigma, m\,{:}\,\beta, \mathtt{mcase}\ \mathtt{inl}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (a_1 : \tau_1)\ \Rightarrow\ e_s \mid \mathtt{inr}\ (a_2 : \tau_2)\ \Rightarrow\ e'_s\ \mathtt{end} \ \Downarrow^{\mathsf{e}}_{\mathsf{s}} \ v', \sigma', \mathtt{T}_s} \ \textbf{(case/inl)}$$

$$\frac{\sigma, m\,{:}\,\mathtt{inr} \cdot \beta, [v/a_2]e_s \quad \Downarrow^{\mathsf{e}}_{\mathsf{s}} \quad v', \sigma', \mathtt{T}_s}{\sigma, m\,{:}\,\beta, \mathtt{mcase}\ \mathtt{inr}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (a_1 : \tau_1)\ \Rightarrow\ e_s \mid \mathtt{inr}\ (a_2 : \tau_2)\ \Rightarrow\ e'_s\ \mathtt{end} \ \Downarrow^{\mathsf{e}}_{\mathsf{s}} \ v', \sigma', \mathtt{T}_s} \ \textbf{(case/inr)}$$

Figure 13.9: Evaluation of stable expressions.

$$\frac{\sigma, l \leftarrow t_c \ \Downarrow^{\mathsf{t}}_{\mathsf{c}} \ \sigma', \mathtt{T}_c}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{return}\,(t_c) \ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}} \ \ \sigma', \mathtt{T}_c}\ \textbf{(return)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ m : \beta, \mathtt{T} \overset{find}{\rightsquigarrow} \epsilon, \epsilon \\ \alpha' = \alpha[l_1' \leftarrow \alpha[l_1]] \ldots [l_n' \leftarrow \alpha[l_n]],\ l_i' \notin \mathrm{dom}(\alpha),\ l_i' \neq l_j' \\ (\alpha', \mu, \chi, \mathtt{T}), l \leftarrow [l_1'/x_1, \ldots l_n'/x_n]\, t_c \ \Downarrow^{\mathsf{t}}_{\mathsf{c}} \ \sigma', \mathtt{T}_c \\ \mathtt{T}_s = \langle R^{x.\mathtt{write}(x)}_{l_1} \mathtt{W}_{\tau_1} \rangle_{l_1':\tau_1} ; \ldots ; \langle R^{x.\mathtt{write}(x)}_{l_n} \mathtt{W}_{\tau_n} \rangle_{l_n':\tau_n}\end{array}}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{memo}\, x_1 = l_1 \ldots x_n = l_n\ \mathtt{in}\, t_c\ \mathtt{end} \ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}} \ \ \sigma', \left(\mathtt{T}_s\,;\{\,\mathtt{T}_c\,\}^{m:\beta}_{((l_1',\ldots,l_n'))}\right)}\ \textbf{(memo/not found)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}) = \sigma \\ m : \beta, \mathtt{T} \overset{find}{\rightsquigarrow} \{\,\mathtt{T}_c\,\}^{m:\beta}_{((l_1',\ldots l_n'))}, \mathtt{T}' \\ \alpha' = \alpha[l_1' \leftarrow \alpha[l_1]] \ldots [l_n' \leftarrow \alpha[l_n]] \\ (\alpha', \mu), \chi \cup \{l_1', \ldots, l_n'\}, l \leftarrow \{\,\mathtt{T}_c\,\}^{m:\beta}_{((l_1',\ldots l_n'))} \ \overset{\mathsf{c}}{\curvearrowright} \ \mathtt{T}_c', \chi', (\alpha'', \mu') \\ \mathtt{T}_s = \langle R^{x.\mathtt{write}(x)}_{l_1} \mathtt{W}_{\tau_1} \rangle_{l_1':\tau_1} ; \ldots ; \langle R^{x.\mathtt{write}(x)}_{l_n} \mathtt{W}_{\tau_n} \rangle_{l_n':\tau_n}\end{array}}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{memo}\, x_1 = l_1 \ldots x_n = l_n\ \mathtt{in}\, t_c\ \mathtt{end} \ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}} \ \ (\alpha'', \mu', \chi', \mathtt{T}'), (\mathtt{T}_s\,; \mathtt{T}_c')}\ \textbf{(memo/found)}$$

$$\frac{\begin{array}{lll}\sigma, t_s & \Downarrow^{\mathsf{t}}_{\mathsf{s}} & v, \sigma', \mathtt{T}_s \\ \sigma', m\!:\!\beta, l \leftarrow [v/a]e_c & \Downarrow^{\mathsf{e}}_{\mathsf{c}} & \sigma'', \mathtt{T}_c\end{array}}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{let}\, a : \tau\, \mathtt{be}\, t_s\, \mathtt{in}\, e_c\, \mathtt{end} \ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}} \ \ \sigma'', \mathtt{T}_s\,;\,\mathtt{T}_c}\ \textbf{(let)}$$

$$\frac{\sigma, m\!:\,!\,v \cdot \beta, l \leftarrow [v/x]e_c \quad \Downarrow^{\mathsf{e}}_{\mathsf{c}} \quad \sigma', \mathtt{T}_c}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{let}\ !\,x : \tau\, \mathtt{be}\ !\,v\, \mathtt{in}\, e_c\, \mathtt{end} \ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}} \ \ \sigma', \mathtt{T}_c}\ \textbf{(let!)}$$

$$\frac{\sigma, m\!:\!\beta, l \leftarrow [v_1/a_1, v_2/a_2]e_c \quad \Downarrow^{\mathsf{e}}_{\mathsf{c}} \quad \sigma', \mathtt{T}_c}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{let}\, a_1 \!\times\! a_2\, \mathtt{be}\, v_1 \times v_2\, \mathtt{in}\, e_c\, \mathtt{end} \ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}} \ \ v, \sigma', \mathtt{T}_c}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma, m\!:\!\mathtt{inl} \cdot \beta, l \leftarrow [v/a_1]e_c \quad \Downarrow^{\mathsf{e}}_{\mathsf{c}} \quad \sigma', \mathtt{T}_c}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{mcase}\, \mathtt{inl}_{\tau_1+\tau_2} v\, \mathtt{of}\, \mathtt{inl}\ (a_1\!:\!\tau_1)\ \Rightarrow\ e_c \mid \mathtt{inr}\ (a_2\!:\!\tau_2)\ \Rightarrow\ e_c'\, \mathtt{end} \ \ \Downarrow^{\mathsf{e}}_{\mathsf{c}} \ \ \sigma', \mathtt{T}_c}\ \textbf{(case/inl)}$$

$$\frac{\sigma, m\!:\!\mathtt{inr} \cdot \beta, l \leftarrow [v/a_2]e_c \quad \Downarrow^{\mathsf{e}}_{\mathsf{c}} \quad \sigma', \mathtt{T}_c}{\sigma, m\!:\!\beta, l \leftarrow \mathtt{mcase}\, \mathtt{inr}_{\tau_1+\tau_2} v\, \mathtt{of}\, \mathtt{inl}\ (a_1\!:\!\tau_1)\ \Rightarrow\ e_c \mid \mathtt{inr}\ (a_2\!:\!\tau_2)\ \Rightarrow\ e_c'\, \mathtt{end} \ \Downarrow^{\mathsf{t}}_{\mathsf{s}} \ v', \sigma', \mathtt{T}_c}\ \textbf{(case/inr)}$$

Figure 13.10: Evaluation of changeable expressions.

$$\frac{}{m : \beta, \epsilon \stackrel{\mathit{find}}{\rightsquigarrow} \epsilon, \epsilon} \qquad \frac{m : \beta, \mathtt{T}_c \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}{m : \beta, \langle \mathtt{T}_c \rangle_{l:\tau} \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}$$

$$\frac{m : \beta, \mathtt{T}_s \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \quad \mathtt{T}_1 \neq \epsilon}{m : \beta, \mathtt{T}_s \,;\, \mathtt{T}'_s \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \,;\, \mathtt{T}'_s} \qquad \frac{\begin{array}{c} m : \beta, \mathtt{T}_s \stackrel{\mathit{find}}{\rightsquigarrow} \epsilon, \epsilon \\ m : \beta, \mathtt{T}'_s \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \end{array}}{m : \beta, \mathtt{T}_s \,;\, \mathtt{T}'_s \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}$$

$$\frac{m = m' \wedge \beta = \beta'}{m : \beta, \{\, \mathtt{T}_s \,\}^{m' : \beta'}_{(v,(l_1,\ldots,l_n))} \stackrel{\mathit{find}}{\rightsquigarrow} \{\, \mathtt{T}_s \,\}^{m' : \beta'}_{(v,(l_1,\ldots,l_n))}, \epsilon} \qquad \frac{\begin{array}{c} m \neq m' \vee \beta \neq \beta' \\ m : \beta, \mathtt{T}_s \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \end{array}}{m : \beta, \{\, \mathtt{T}_s \,\}^{m' : \beta'}_{(v,(l_1,\ldots,l_n))} \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}$$

$$\frac{}{m : \beta, \mathtt{W}_\tau \stackrel{\mathit{find}}{\rightsquigarrow} \epsilon, \epsilon} \qquad \frac{m : \beta, \mathtt{T}_c \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}{m : \beta, R^{x.t}_l(\mathtt{T}_c) \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}$$

$$\frac{m : \beta, \mathtt{T}_s \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \quad \mathtt{T}_1 \neq \epsilon}{m : \beta, \mathtt{T}_s \,;\, \mathtt{T}_c \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \,;\, \mathtt{T}_c} \qquad \frac{\begin{array}{c} m : \beta, \mathtt{T}_s \stackrel{\mathit{find}}{\rightsquigarrow} \epsilon, \epsilon \\ m : \beta, \mathtt{T}_c \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \end{array}}{m : \beta, \mathtt{T}_s \,;\, \mathtt{T}_c \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}$$

$$\frac{m = m' \wedge \beta = \beta'}{m : \beta, \{\, \mathtt{T}_c \,\}^{m' : \beta'}_{((l_1,\ldots,l_n))} \stackrel{\mathit{find}}{\rightsquigarrow} \{\, \mathtt{T}_c \,\}^{m' : \beta'}_{((l_1,\ldots,l_n))}, \epsilon} \qquad \frac{\begin{array}{c} m \neq m' \vee \beta \neq \beta' \\ m : \beta, \mathtt{T}_c \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2 \end{array}}{m : \beta, \{\, \mathtt{T}_c \,\}^{m' : \beta'}_{((l_1,\ldots,l_n))} \stackrel{\mathit{find}}{\rightsquigarrow} \mathtt{T}_1, \mathtt{T}_2}$$

Figure 13.11: The rules for memo look up changeable (top) and stable (bottom) traces.

$$\varsigma, \chi, \varepsilon \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} \varepsilon, \chi, \varsigma$$

$$\frac{\varsigma, \chi, l \leftarrow \mathtt{T}_c \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} \mathtt{T}'_c, \chi', \varsigma'}{\varsigma, \chi, \langle \mathtt{T}_c \rangle_{l:\tau} \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} \langle \mathtt{T}'_c \rangle_{l:\tau}, \chi', \varsigma'} \ \textbf{(mod)}$$

$$\frac{\begin{array}{c}\varsigma, \chi, \mathtt{T}_s \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} \mathtt{T}''_s, \chi', \varsigma' \\ \varsigma', \chi', \mathtt{T}'_s \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} \mathtt{T}'''_s, \chi'', \varsigma''\end{array}}{\varsigma, \chi, (\mathtt{T}_s \,;\, \mathtt{T}'_s) \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} (\mathtt{T}''_s \,;\, \mathtt{T}'''_s), \chi'', \varsigma''} \ \textbf{(let)}$$

$$\frac{\varsigma, \chi, \mathtt{T}_s \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} \mathtt{T}'_s, \chi', \varsigma'}{\varsigma, \chi, \{\, \mathtt{T}_s \,\}^{m\,:\,\beta}_{(v,(l_1,\ldots,l_n))} \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} \{\, \mathtt{T}'_s \,\}^{m\,:\,\beta}_{(v,(l_1,\ldots l_n))}, \chi', \varsigma'} \ \textbf{(memo)}$$

$$\varsigma, \chi, l \leftarrow \mathtt{W}_\tau \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} \mathtt{W}_\tau, \chi, \varsigma \ \textbf{(write)}$$

$$\frac{\begin{array}{c}(l \notin \chi) \\ \varsigma, \chi, l' \leftarrow \mathtt{T}_c \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} \mathtt{T}'_c, \chi', \varsigma'\end{array}}{\varsigma, \chi, l' \leftarrow R^{x.t_c}_l(\mathtt{T}_c) \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} R^{x.t_c}_l(\mathtt{T}'_c), \chi', \varsigma'} \ \textbf{(read, no change)}$$

$$\frac{\begin{array}{c}(l \in \chi) \\ (\alpha, \mu) = \varsigma \\ (\alpha, \mu, \chi, \mathtt{T}_c), l' \leftarrow [\alpha(l)/x]\, t_c \Downarrow^{\mathsf{t}}_{\mathsf{c}} (\alpha', \mu', \chi', \_), \mathtt{T}'_c \\ \varsigma' = (\alpha', \mu') \\ \chi'' = \chi' \cup \{l'\}\end{array}}{\varsigma, \chi, l' \leftarrow R^{x.t_c}_l(\mathtt{T}_c) \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} R^{x.t_c}_l(\mathtt{T}'_c), \chi'', \varsigma'} \ \textbf{(read, change)}$$

$$\frac{\begin{array}{c}\varsigma, \chi, \mathtt{T}_s \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} \mathtt{T}'_s, \chi', \varsigma' \\ \varsigma', \chi', l \leftarrow \mathtt{T}_c \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} \mathtt{T}'_c, \chi'', \varsigma''\end{array}}{\varsigma, \chi, l \leftarrow (\mathtt{T}_s \,;\, \mathtt{T}_c) \overset{\overset{\mathsf{s}}{\curvearrowright}}{\rightarrow} (\mathtt{T}'_s \,;\, \mathtt{T}'_c), \chi'', \varsigma''} \ \textbf{(let)}$$

$$\frac{\varsigma, \chi, l \leftarrow \mathtt{T}_c \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} \mathtt{T}'_c, \chi', \varsigma'}{\varsigma, \chi, l \leftarrow \{\, \mathtt{T}_c \,\}^{m\,:\,\beta}_{((l_1,\ldots,l_n))} \overset{\overset{\mathsf{c}}{\curvearrowright}}{\rightarrow} \{\, \mathtt{T}'_c \,\}^{m\,:\,\beta}_{((l_1,\ldots l_n))}, \chi', \varsigma'} \ \textbf{(memo)}$$

Figure 13.12: Change propagation for stable (top) and changeable (bottom) traces.

Chapter 14

Imperative Self-Adjusting Programming

This section presents an imperative language, called SLi, for self-adjusting computation. The language extends the SLf language with modifiable references that can be written any number of times, instead of just once. The key idea behind supporting multiple writes is to keep a version log of all writes to a modifiable. The idea of keeping track of all versions of a reference is used by Driscoll, Sarnak, Sleator, and Tarjan to make data structures persistent [31].

We present the static and dynamic semantics for the language. Based on the dynamic semantics, we give an operational semantics of change propagation with multiple writes. We do not, however, present an implementation of the language, or study the complexity properties of programs written in the language—these are left for future work.

14.1 The Language

The abstract syntax of the SLi language is given in Figure 14.1. Meta-variables x, y, z and their variants range over an unspecified set of variables, Meta-variables a, b, c and variants range over an unspecified set of resources. Meta variable l and variants range over a unspecified set of locations. Meta variable m ranges over a unspecified set of memo-function identifiers. Variables, resources, locations, memo-function identifiers are mutually disjoint. The syntax is restricted to "2/3-cps" or "named form" to streamline the presentation of the dynamic semantics.

The types include `int`, `unit`, sums $\tau_1 + \tau_2$, and products $\tau_1 \times \tau_2$, bang $!\,\tau$ types, the function types, $\tau_1 \rightarrow \tau_2$, and memoized function types $\tau_1 \stackrel{\mathsf{m}}{\rightarrow} \tau_2$.

As with the SLf language (Chapter 13) the underlying type of a bang type $!\,\tau$ is required to be an indexable type. An *indexable type* accepts an injective *index* function into integers. Any type can be made indexable by supplying an index function based on boxing or tagging. Since boxing is completely standard and well understood, we do not have a separate category for indexable types.

The abstract syntax is structured into *terms* and *expression*. Terms evaluate independent of their contexts, whereas expression evaluate with respect to a memo table. Since multiple writes to modifiables are permitted, terms and expressions need not be further partitioned into stable and changeable terms.

Terms. Familiar mechanism of functional programming are embedded in the language in the form terms.

Types	τ	$::=$	$\texttt{int} \mid \texttt{unit} \mid \tau\,\texttt{mod} \mid\, !\,\tau \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 \xrightarrow{\texttt{m}} \tau_2$
Values	v	$::=$	$\star \mid n \mid x \mid a \mid l \mid m \mid\, !\,v \mid \texttt{(}v_1\texttt{,}v_2\texttt{)} \mid \texttt{inl}_{\tau_1+\tau_2} v \mid \texttt{inr}_{\tau_1+\tau_2} v \mid$ $\texttt{m\_fun}_m\, f\,\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,t_c\,\texttt{end}$
Operators	o	$::=$	$+ \mid - \mid = \mid < \mid \ldots$
Expressions	e	$::=$	$\texttt{return}\,\texttt{(}t\texttt{)} \mid \texttt{let}\,!\,x\texttt{:}\tau\,\texttt{be}\,v\,\texttt{in}\,e\,\texttt{end} \mid \texttt{let}\,a_1\texttt{:}\tau_1\times a_2\texttt{:}\tau_2\,\texttt{be}\,v\,\texttt{in}\,e\,\texttt{end} \mid$ $\texttt{mcase}\,v\,\texttt{of}\,\texttt{inl}\,\texttt{(}a_1\texttt{:}\tau_1\texttt{)} \Rightarrow e \mid \texttt{inr}\,\texttt{(}a_2\texttt{:}\tau_2\texttt{)} \Rightarrow e\,\texttt{end}$
Terms	t	$::=$	$v \mid o\,\texttt{(}v_1,\ldots,v_n\texttt{)} \mid \texttt{mfun}\,f(a:\tau_1):\tau_2\,\texttt{is}\,e\,\texttt{end} \mid v_1@v_2 \mid$ $\texttt{let}\,x\,\texttt{be}\,t_1\,\texttt{in}\,t_2\,\texttt{end} \mid \texttt{mod}_\tau\,t \mid \texttt{read}\,v\,\texttt{as}\,x\,\texttt{in}\,t\,\texttt{end} \mid \texttt{write}\,\texttt{(}v_1,v_2\texttt{)} \mid$ $\texttt{case}\,v\,\texttt{of}\,\texttt{inl}\,\texttt{(}x_1\texttt{:}\tau_1\texttt{)} \Rightarrow t_1 \mid \texttt{inr}\,\texttt{(}x_2\texttt{:}\tau_2\texttt{)} \Rightarrow t_2\,\texttt{end}$

Figure 14.1: The abstract syntax of SLi.

In addition, terms include constructs for creating, reading, and writing modifiables, and memo functions. The stable term $\texttt{mod}_\tau\ t$ allocates a new modifiable reference whose value is determined by the term t. The term $\texttt{write}\,\texttt{(}l,v\texttt{)}$ writes the value v into the modifiable l. The term $\texttt{read}\ l\ \texttt{as}\ x\ \texttt{in}\ t\ \texttt{end}$ binds the contents of the modifiable l to the variable x, then continues evaluation of t. Memo functions are functions whose bodies are expressions.

Expressions. Expressions are evaluated in the context of a memo table. Terms are included in expressions via a `return`. As with the SLf language, the constructs `let*`, `let!`, `let?`, `mcase` express dependences between the input and the result of a memoized function. The `return` computes the result based on the the dependences expressed by these constructs. The ability to write modifiables multiple times eliminates the need for differentiating between strict and non-strict dependences. Non-strict dependences can be "simulated" by separating the allocation and the writing of modifiables. For this reason, the language does not support non-strict dependences.

14.1.1 Static Semantics

The static semantics consists of typing judgments for terms and expressions. Each typing judgment takes place under three contexts: Δ for resources, Λ for locations, and Γ for ordinary variables. The judgment $\Delta;\Lambda;\Gamma \vdash t : \tau$ states that t is a well formed term of type τ relative to Δ,Λ and Γ. The judgment $\Delta;\Lambda;\Gamma \vdash e : \tau$ states that e is a well formed expression of type τ relative to Δ,Λ and Γ.

Figure 14.2 shows the typing rules for values, Figure 14.3 shows the typing rules for terms and and expressions. The term typing rules are very similar to those of the MFL language extended with ordinary references. Like ordinary references, modifiables can be written many times. The key difference between ordinary references and modifiable references is that the read of a modifiable reference provides a body that uses the underlying value of that modifiable. Also, the return type of `read` is set to `unit` to prevent a modifiable value from being used without being read. By providing a body, the `read` enables re-computation of all values that depend on a modifiable, by re-evaluating the code that depends on it. The typing rules for expressions are very similar to those of the MFL language. The only difference is the addition of the context

Λ for modifiables. As in the MFL language, the body of `return` must use no resources. The `letx`, and `mcase` constructs bind resources to their subject and the `let!` construct binds the underlying value of a bang type to an ordinary variable.

$$\frac{}{\Delta;\Lambda;\Gamma\vdash n:\texttt{int}}\ \textbf{(integer)}\qquad \frac{}{\Delta;\Lambda;\Gamma\vdash \star:\texttt{unit}}\ \textbf{(unit)}$$

$$\frac{(\Delta(a)=\tau)}{\Delta;\Lambda;\Gamma\vdash a:\tau}\ \textbf{(resource)}\qquad \frac{(\Lambda(l)=\tau)}{\Delta;\Lambda;\Gamma\vdash l:\tau\ \texttt{mod}}\ \textbf{(location)}\qquad \frac{(\Gamma(x)=\tau)}{\Delta;\Lambda;\Gamma\vdash x:\tau}\ \textbf{(variable)}$$

$$\frac{\emptyset;\Lambda;\Gamma\vdash t:\tau}{\Delta;\Lambda;\Gamma\vdash \texttt{!}\,t:\texttt{!}\,\tau}\ \textbf{(bang)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash v_1:\tau_1\quad \Delta;\Lambda;\Gamma\vdash v_2:\tau_2}{\Delta;\Lambda;\Gamma\vdash \texttt{(}v_1\texttt{,}v_2\texttt{)}:\tau_1\times\tau_2}\ \textbf{(product)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash v:\tau_1}{\Delta;\Lambda;\Gamma\vdash \texttt{inl}_{\tau_1+\tau_2}v:\tau_1+\tau_2}\ \textbf{(sum/l)}\qquad \frac{\Delta;\Lambda;\Gamma\vdash v:\tau_2}{\Delta;\Lambda;\Gamma\vdash \texttt{inr}_{\tau_1+\tau_2}v:\tau_1+\tau_2}\ \textbf{(sum/r)}$$

$$\frac{\Delta,a{:}\tau_1;\Lambda;\Gamma,f{:}\tau_1\xrightarrow{\mathsf{m}}\tau_2;\vdash e:\tau_2}{\Delta;\Lambda;\Gamma\vdash \texttt{m\_fun}_m\,f\,\texttt{(}a\texttt{:}\tau_1\texttt{):}\tau_2\,\texttt{is}\,e\,\texttt{end}:\tau_1\xrightarrow{\mathsf{m}}\tau_2}\ \textbf{(mfun)}$$

Figure 14.2: Typing judgments for values.

14.1.2 Dynamic Semantics

The dynamic semantics relies on keeping track of read and write operations in the form of a trace and using the trace to propagate changes. The key ingredient of the semantics is a mechanism for maintaining a version log of all writes so that dependences between all values can be tracked. For a given modifiable l, we define each value written into l as a *version* of l. The dynamic semantics maintains all versions of the modifiables along with a total ordering on them according. The total order is created by tagging each version with a time stamp that represents the time at which the write takes place. Writing to a modifiable adds a new version for that modifiable to the store but does not destroy the previous version. The read of a modifiable accesses the most recent version. By keeping track of versions of modifiables, we create a *persistent* data structure that represent the executions. This idea is inspired by the work of Driscoll, Sarnak, Sleator, and Tarjan on persistent data structures [31].

The dynamic semantics consists of evaluation judgments for terms and expressions. All evaluation takes place with respect to a state $\sigma = (\alpha, \mu, \chi, \mathtt{T}, \Upsilon^r)$ consisting of a location store α, a memoized-function identifier store μ, a set of changed locations χ, a re-use trace $\mathtt{T}$, and a *time store* Υ and the current time stamp r. The store α keeps the versions of modifiables and the memoized-function identifier store dispenses unique identifiers for memoized functions. The set of changed location χ contains locations that has been changed since the previous evaluation. The re-use trace $\mathtt{T}$ is the trace of available results to be re-used by the memo functions. Re-use trace is provided by change propagation and is empty in the initial evaluation. The time store is a set of time stamps.

Time Stamps. Time stamps are used to tag versions with the time at which they are created. Each time stamp is represented by a real number. All live time-stamps are kept in a time store, Υ. The semantics relies on three operations, `next`, `new`, and `delete` on time stamps. The `next` operation takes a time store Υ and a time stamp r, and returns the earliest time stamp in Υ that is greater than r. The `new` operation takes a time store Υ and a time stamp r, and returns a new time store Υ' and time stamps r'. The new time stamps r' comes in between r and the first time stamp after r and the new store Υ' is the store Υ extended with r'. The `delete` operations take a time store Υ and a two time interval and returns a new time store consisting of the time intervals outside the interval. More precisely,

$$
\begin{array}{rcl}
\texttt{next}(\Upsilon, r) & = & r' \text{ s.t. } r' \in \Upsilon \wedge (\forall r'' \in \Upsilon.\, r'' < r \vee r'' > r' \vee r'' = r \vee r'' = r') \\
\texttt{new}(\Upsilon, r) & = & (r', \Upsilon') \text{ s.t. } r' = (r + r'')/2 \text{ where } r'' = \texttt{next}(r, \Upsilon) \wedge \Upsilon' = \Upsilon \cup \{r'\} \\
\texttt{delete}(\Upsilon, r_1, r_2) & = & \{r \mid r \in \Upsilon \wedge (r < r_1 \vee r > r_2)\}
\end{array}
$$

Note that `new` creates the new time stamp by averaging. In practice, it will not be efficient to represent time-stamps with real numbers. Instead the virtual clock data structure described in Section 6.1.1 or an order-maintenance data structure [30] can be used to implement time stamps efficiently.

Traces. Evaluation of a term or expression records certain information in the form of a trace. The trace is then used during change propagation to update the evaluation according to a change. A trace is formally defined as follows.

$$
\texttt{T} \ ::= \ \epsilon \mid \texttt{W}_{\tau}^{r}(l) \mid \prec^{r_1} R_l^{x.t}\, \texttt{T}\, ^{r_2}\succ \mid \{\, ^{r_1}\, \texttt{T}\, ^{r_2}\, \}_{m\,:\,\beta,v} \mid \texttt{T}\, ;\, \texttt{T}'
$$

When writing traces, we adopt the convention that "`;`" is right-associative.

The empty trace arises as a result of evaluation of terms whose values do no depend on modifiables. A write, $\texttt{W}_{\tau}^{r}(l)$, records the write of a value of type τ in the modifiable `l`, r is the time-stamp of the written value. A read trace $\prec^{r_1} R_l^{x.t}\, \texttt{T}\, ^{r_2}\succ$ specifies the location read l, the context of use of its value $x.e$, the trace `T` of the body of the read, and a time interval consisting of the time stamps r_1 and r_2 that denotes current time stamp at the beginning and at the end of that read. The trace $\texttt{T}\, ;\, \texttt{T}'$ results from evaluation of a `let` expression, the first trace resulting from the bound expression, the second from the body. The trace $\{\, ^{r_1}\, \texttt{T}\, ^{r_2}\, \}_{m\,:\,\beta,v}$ arises from the evaluation of a memoized expressions; m is the identifier, β is the branch expressing the input-output dependences, v is the result of the evaluation, `T` is the trace of the memoized expressions, and (r_1, r_2) is the time interval in which the expression evaluates.

Term evaluation. Figure 14.4 shows the the evaluation rules for terms. The rule $\sigma, t, \Downarrow^{\mathsf{t}} v, \sigma', \texttt{T}_s$ states that evaluation of the term t with respect to the state σ yields value v, state σ', and the trace `T`. The evaluation rules for values, primitives, `case`, and `let` are standard.

The evaluation of a memoized function allocates a fresh identifier for that function. The identifier is used when performing memo look ups. Applying a memo function evaluates the body of the function in the context of an identifier m and a branch β. As in selective memoization, the branch collects the precise dependencies between the input and the output. The branch is initialized to empty (ε) and is extended during the evaluation of the body.

The most interesting judgments belong to `mod`, `read`, and `write` constructs. The evaluation of `mod v` allocates a fresh modifiable l in the store and a new time stamps r_2 and extends the store with a new version of l time stamped with r_2. The evaluation of `write (l,v)` creates a new time stamp r_2 and extends the store by adding a new version for modifiable l and time stamps the version with r_2. The evaluation of `read l as x in t end` substitutes for the current version of l for x in t and evaluates t. The read returns the unit value. The trace of the read records the location being read, the body, and the variable bound. The trace also remembers the time interval in which the read executes. This is critical in maintaining the total ordering of time stamps during change propagation.

Expression Evaluation. Expression evaluation takes place in the context of memo function identifier m, a branch, and a re-use trace. The incremental exploration constructs (`let!`, `let*`, `mcase`) create a branch, denoted β. A *branch* is a list of *events* corresponding to "choice points" in the evaluation of an expression.

$$\begin{array}{lcll} \textit{Event} & \varepsilon & ::= & \texttt{!}v \mid \texttt{inl} \mid \texttt{inr} \\ \textit{Branch} & \beta & ::= & \bullet \mid \varepsilon \cdot \beta \end{array}$$

The branch and the identifier m is used by the `return` construct to lookup the re-use trace for a match. If a match is found, the result is returned and the body of `return` is skipped. The construction and the use of branches is very similar to that of the MFL language. The re-use of results is similar to the SLf language, except that there are no non-strict dependences.

Figure 14.5 shows the judgments for expression evaluation. The judgment $\sigma, m\,\mathbf{:}\,\beta, e \Downarrow^{\mathsf{e}} v, \sigma', \mathtt{T}$ states that the evaluation of the expression e with respect to state σ, branch β, and memo identifier m yields the value v, the state σ', and the trace `T`. The evaluation of `let!` expression binds the underlying value of the banged term to an ordinary variable, extend the branch with that value, and evaluates the body. The evaluation of the `mcase` expressions extends the branch with the branch taken and evaluates the body of that branch.

The evaluation of the `return t` first checks the memo trace T for a match using the memo lookup relation *find*. The judgments for the find relation is shown in Figure 14.6. If the result is not found, then the term t is evaluated and the result is returned. The trace of the expression consists of the trace of the body, the branch, the result, and the time interval in which the body is executed, If a result is found, then the result and the trace for that result will be re-used. To re-use the trace found, we first delete all the versions created between the current time stamp and the start of the re-used trace and add all the locations written within the deleted interval to the changed set χ. We then perform a change propagation on the re-used trace and return the revised trace.

The memo lookup (Figure 14.6) seeks for a memoized result in the re-use trace whose identifier and branch matches m and β'. If a result is not found, then the look up returns an empty trace (ε). If a result is found, then it returns the trace found and the uninspected tail of the re-used trace. The rules for memo looks ups are an adaptation of those for the SLf language to the particular notion of traces considered in this chapter.

14.2 Change Propagation

Given a trace T, a state ς, and a set of changed locations χ, the algorithm scans through T as it seeks for reads of changed locations. When such a read is found, the body of the read is re-evaluated to obtain a revised trace. When re-evaluating a read, the time is rolled back to the start of the read's interval and the trace of the read is passed to the evaluation for the purposes of memoization.

Figure 14.7 shows the rules for change propagation. The judgment $\varsigma, \chi, \mathtt{T} \stackrel{\mathsf{s}}{\curvearrowright} \mathtt{T}', \chi', \varsigma'$ states that change propagation on trace $\mathtt{T}$ with respect to the changed set χ and the state ς yields the revised trace $\mathtt{T}'$, the changed set χ' and the state ς. The state $\varsigma = (\alpha, \mu, \Upsilon)$ consists of a location store α, and memo-function identifier store μ, and a version store Υ.

Given a trace, change propagation mimics the evaluation rule that originally generated that trace. To stress this correspondence, each rule is marked with the name of the evaluation rule to which it corresponds. For example, the propagation rule for the trace $\mathtt{T}_s$; $\mathtt{T}'_s$ mimics the `let` rule of the stable mode that gives rise to this trace. The most interesting judgments are those for the `read` and `write`. If the source location l being read is not in the changed set χ, then that read is skipped over and the trace inside the read is revised. If the source location l is changed, then the body of the read is re-evaluated after substituting the version of l at time r_1 for x. To maintain the ordering of the version numbers, the re-evaluation is started at time r_1. The evaluation advances the time to r_3. After the evaluation, all versions, and time stamps between $r_3$and $r_2$are removed. The judgment for the `write` removes the location being written from the changed set. This is because all following reads of the locations will access this version and therefore need to be re-evaluated.

$$\frac{\Delta;\Lambda;\Gamma \vdash v_i : \tau_i \quad (1 \le i \le n) \quad \vdash_o o : (\tau_1,\ldots,\tau_n)\,\tau}{\Delta;\Lambda;\Gamma \vdash o\,(v_1,\ldots,v_n)\;:\tau} \ \textbf{(primitive)}$$

$$\frac{\Delta, a{:}\tau_1;\Lambda;\Gamma,\ f{:}\tau_1 \xrightarrow{\mathtt{m}} \tau_2 \vdash e : \tau_2}{\Delta;\Lambda;\Gamma \vdash \mathtt{mfun}\ f(a:\tau_1):\tau_2\ \mathtt{is}\ e\ \mathtt{end} : \tau_1 \xrightarrow{\mathtt{m}} \tau_2} \ \textbf{(memo fun)}$$

$$\frac{\Delta;\Lambda;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\mathtt{m}} \tau_2 \quad \Delta;\Lambda;\Gamma \vdash v_2 : \tau_1}{\Delta;\Lambda;\Gamma \vdash v_1 @ v_2 : \tau_2} \ \textbf{(memo apply)}$$

$$\frac{\Delta;\Lambda;\Gamma \vdash t_1 : \tau_1 \quad \Lambda;\Gamma, x:\tau_1 \vdash t_2 : \tau_2}{\Delta;\Lambda;\Gamma \vdash \mathtt{let}\ x\ \mathtt{be}\ t_1\ \mathtt{in}\ t_2\ \mathtt{end} : \tau_2} \ \textbf{(let)}$$

$$\frac{\Delta;\Lambda;\Gamma \vdash t : \tau}{\Delta;\Lambda;\Gamma \vdash \mathtt{mod}_\tau\ t : \tau\ \mathtt{mod}} \ \textbf{(mod)}$$

$$\frac{\Delta;\Lambda;\Gamma \vdash v : \tau\ \mathtt{mod} \quad \Delta;\Lambda;\Gamma, x:\tau \vdash t : \mathtt{unit}}{\Delta;\Lambda;\Gamma \vdash \mathtt{read}\ v\ \mathtt{as}\ x\ \mathtt{in}\ t\ \mathtt{end} : \mathtt{unit}} \ \textbf{(read)}$$

$$\frac{\Delta;\Lambda;\Gamma \vdash v_1 : \tau\ \mathtt{mod} \quad \Delta;\Lambda;\Gamma \vdash v_2 : \tau}{\Delta;\Lambda;\Gamma \vdash \mathtt{write}\,(v_1, v_2)\ : \mathtt{unit}} \ \textbf{(write)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \vdash & v : \tau_1 + \tau_2 \\ \Delta;\Lambda;\Gamma, x_1{:}\tau_1 & \vdash & t_1 : \tau \\ \Delta;\Lambda;\Gamma, x_2{:}\tau_2 & \vdash & t_2 : \tau\end{array}}{\Delta;\Lambda;\Gamma \vdash \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}\ (x_1{:}\tau_1)\ \Rightarrow\ t_1 \mid \mathtt{inr}\ (x_2{:}\tau_2)\ \Rightarrow\ t_2\ \mathtt{end} : \tau} \ \textbf{(case)}$$

$$\frac{\emptyset;\Lambda;\Gamma \ \vdash\ t : \tau}{\Delta;\Lambda;\Gamma \vdash \mathtt{return}\,(t)\ : \tau} \ \textbf{(return)}$$

$$\frac{\Delta;\Lambda;\Gamma \vdash v : \ !\,\tau_1 \quad \Delta;\Lambda;\Gamma, x{:}\tau_2 \vdash e : \tau_2}{\Delta;\Lambda;\Gamma \vdash \mathtt{let}\ !\,x{:}\tau_1\ \mathtt{be}\ v\ \mathtt{in}\ e\ \mathtt{end} : \tau_2} \ \textbf{(let!)}$$

$$\frac{\Delta;\Lambda;\Gamma \vdash v : \tau_1 \times \tau_2 \quad \Delta, a_1{:}\tau_1, a_2{:}\tau_2;\Lambda;\Gamma \vdash e : \tau}{\Delta;\Lambda;\Gamma \vdash \mathtt{let}\ a_1{:}\tau_1{\times}a_2{:}\tau_2\ \mathtt{be}\ v\ \mathtt{in}\ e\ \mathtt{end} : \tau} \ \textbf{(let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\Delta;\Lambda;\Gamma & \vdash & v : \tau_1 + \tau_2 \\ \Delta, a_1{:}\tau_1;\Lambda;\Gamma & \vdash & e_1 : \tau \\ \Delta, a_2{:}\tau_2;\Lambda;\Gamma & \vdash & e_2 : \tau\end{array}}{\Delta;\Lambda;\Gamma \vdash \mathtt{mcase}\ v\ \mathtt{of}\ \mathtt{inl}\ (a_1{:}\tau_1)\ \Rightarrow\ e_1 \mid \mathtt{inr}\ (a_2{:}\tau_2)\ \Rightarrow\ e_2\ \mathtt{end} : \tau} \ \textbf{(mcase)}$$

Figure 14.3: Typing of terms (top) and expressions (bottom).

$$\sigma, v \Downarrow^{\mathsf{t}} v, \sigma, \varepsilon \ \textbf{(value)}$$

$$\sigma, o\,(v_1, \ldots, v_n)\ \Downarrow^{\mathsf{t}} \mathtt{app}\,(o, (v_1, \ldots, v_n))\,, \sigma, \varepsilon \ \textbf{(primitive)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}, \Upsilon^r) = \sigma \\ \sigma' = (\alpha, \mu \cup \{m\}, \chi, \mathtt{T}, \Upsilon^r),\ m \notin \mathrm{dom}(\mu)\end{array}}{\sigma, \mathtt{mfun}\ f(a : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end} \Downarrow^{\mathsf{t}} \mathtt{ms\_fun}_m\ f\,(a{:}\tau_1)\,{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end}, \sigma', \varepsilon} \ \textbf{(memo fun)}$$

$$\frac{\begin{array}{c}(v_1 = \mathtt{m\_fun}_m\ f\,(a{:}\tau_1)\,{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end}) \\ \sigma, m{:}\varepsilon, [v_1/f, v_2/a]\ e \Downarrow^{\mathsf{e}} v, \sigma', \mathtt{T}\end{array}}{\sigma, v_1 @ v_2 \Downarrow^{\mathsf{t}} v, \sigma', \mathtt{T}} \ \textbf{(memo apply)}$$

$$\frac{\begin{array}{lll}\sigma, t_1 & \Downarrow^{\mathsf{t}} & v_1, \sigma', \mathtt{T} \\ \sigma', [v_1/x]\, t_2 & \Downarrow^{\mathsf{t}} & v_2, \sigma'', \mathtt{T}'\end{array}}{\sigma, \mathtt{let}\ x\ \mathtt{be}\ t_1\ \mathtt{in}\ t_2\ \mathtt{end} \Downarrow^{\mathsf{t}} v_2, \sigma'', (\mathtt{T}\ ;\ \mathtt{T}')} \ \textbf{(let)}$$

$$\frac{\sigma, t_1 \Downarrow^{\mathsf{t}} v', \sigma', \mathtt{T}_s}{\sigma, \mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (x_1{:}\tau_1)\ \Rightarrow\ t_1 \mid \mathtt{inr}\ (x_2{:}\tau_2)\ \Rightarrow\ t_2\ \mathtt{end} \Downarrow^{\mathsf{t}} v', \sigma', \mathtt{T}_s} \ \textbf{(case)}$$

$$\frac{\sigma, t_2 \Downarrow^{\mathsf{t}} v', \sigma', \mathtt{T}}{\sigma, \mathtt{case}\ \mathtt{inr}_{\tau_1+\tau_2} v\ \mathtt{of}\ \mathtt{inl}\ (x_1{:}\tau_1)\ \Rightarrow\ t_1 \mid \mathtt{inr}\ (x_2{:}\tau_2)\ \Rightarrow\ t_2\ \mathtt{end} \Downarrow^{\mathsf{t}} v', \sigma', \mathtt{T}_s} \ \textbf{(case)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}, {\Upsilon_1}^{r_1}) = \sigma \\ (\Upsilon_2, r_2) = \mathtt{new}(\Upsilon_1, r_1) \\ \alpha' = \alpha[(l, r_2) \mapsto v],\ l \notin \mathrm{locs}(\alpha) \\ \sigma' = (\alpha', \mu, \chi, \mathtt{T}, {\Upsilon_2}^{r_2}))\end{array}}{\sigma, \mathtt{mod}_\tau\ v \Downarrow^{\mathsf{t}} l, \sigma', \epsilon} \ \textbf{(mod)}$$

$$\frac{\begin{array}{c}(\alpha, \mu, \chi, \mathtt{T}, {\Upsilon_1}^{r_1}) = \sigma \\ (\Upsilon_2, r_2) = \mathtt{new}(\Upsilon_1, r_1) \\ \alpha' = \alpha[(l, r_2) \mapsto v] \\ \sigma' = (\alpha', \mu, \chi, \mathtt{T}, {\Upsilon_2}^{r_2})\end{array}}{\sigma, \mathtt{write}\,(l, v)\ \Downarrow^{\mathsf{t}} v, \sigma', \mathtt{W}_\tau} \ \textbf{(write)}$$

$$\frac{\begin{array}{c}(\_, \_, \_, \_, {\Upsilon_1}^{r_1})) = \sigma \\ \sigma, [\sigma(l, r_1)/x]\ t \Downarrow^{\mathsf{t}} \sigma', \mathtt{T} \\ (\_, \_, \_, \_, {\Upsilon_2}^{r_2}) = \sigma' \\ r_3 = \mathtt{next}(\Upsilon_2, r_1)\end{array}}{\sigma, \mathtt{read}\ l\ \mathtt{as}\ x\ \mathtt{in}\ t\ \mathtt{end} \Downarrow^{\mathsf{t}} \star, \sigma', \prec^{r_3} R_l^{x.t}\ \mathtt{T}\ ^{r_2}\!\succ} \ \textbf{(read)}$$

Figure 14.4: Evaluation of terms.

$$\frac{\begin{array}{c}(\_,\_,\_,\mathtt{T},{\Upsilon_1}^{r_1}) = \sigma \\ m:\beta,\mathtt{T} \overset{find}{\leadsto} \epsilon,\epsilon \\ \sigma,t \Downarrow^{\mathsf{t}} v,\sigma',\mathtt{T}' \\ (\_,\_,\_,\_,{\Upsilon_2}^{r_2}) = \sigma'\end{array}}{\sigma,m\mathbin{:}\beta,\mathtt{return}\,(t) \Downarrow^{\mathsf{e}} v,\sigma',\{\,^{r_1}\,\mathtt{T}'\,^{r_2}\,\}_{m:\beta,v}}\ \textbf{(return, not found)}$$

$$\frac{\begin{array}{c}(\alpha,\_,\chi,\mathtt{T},{\Upsilon_1}^{r_1}) = \sigma \\ m\mathbin{:}\beta,\mathtt{T} \overset{find}{\leadsto} \{\,^{r_2}\,\mathtt{T}_1\,^{r_3}\,\}_{m:\beta,v},\mathtt{T}_2 \\ \Upsilon_2 = \mathtt{delete}(\Upsilon_1,\mathtt{next}(\Upsilon_1,r_1),r_2) \\ \alpha' = \alpha \setminus \{(l,r) \mid l,r) \in \mathrm{dom}(\alpha) \wedge r_1 < r \leq r_2\} \\ \chi' = \chi \cup \{l \mid (l,r) \in \mathrm{dom}(\alpha) \wedge r_1 < r \leq r_2\} \\ (\alpha',\mu,{\Upsilon_1}^{r_2}),\chi',\mathtt{T}_1 \overset{\curvearrowright}{\rightarrow} \mathtt{T}_1',\chi'',(\alpha'',\mu',{\Upsilon_2}^{r_4}) \\ \sigma' = (\alpha'',\mu',\chi'',\mathtt{T}_2,{\Upsilon_2}^{r_4})\end{array}}{\sigma,m\mathbin{:}\beta,\mathtt{return}\,(t) \Downarrow^{\mathsf{e}} v,\sigma',\{\,^{r_2}\,\mathtt{T}_1'\,^{r_4}\,\}_{m:\beta,v}}\ \textbf{(return, found)}$$

$$\frac{\sigma,m\mathbin{:}\,!\,v\cdot\beta,[v/x]e \Downarrow^{\mathsf{e}} v',\sigma',\mathtt{T}}{\sigma,m\mathbin{:}\beta,\mathtt{let}\ !\,x:\tau\,\mathtt{be}\ !\,v\,\mathtt{in}\,e\,\mathtt{end} \Downarrow^{\mathsf{e}} v',\sigma',\mathtt{T}}\ \textbf{(let!)}$$

$$\frac{\sigma,m\mathbin{:}\beta,[v_1/a_1,v_2/a_2]e \quad \Downarrow^{\mathsf{e}} \quad v,\sigma',\mathtt{T}}{\sigma,m\mathbin{:}\beta,\mathtt{let}\,a_1{\times}a_2\,\mathtt{be}\,v_1\times v_2\,\mathtt{in}\,e\,\mathtt{end} \Downarrow^{\mathsf{e}} v,\sigma',\mathtt{T}}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\sigma,m\mathbin{:}\mathtt{inl}\cdot\beta,[v/a_1]e \quad \Downarrow^{\mathsf{e}} \quad v',\sigma',\mathtt{T}}{\sigma,m\mathbin{:}\beta,\mathtt{mcase}\,\mathtt{inl}_{\tau_1+\tau_2}v\,\mathtt{of}\,\mathtt{inl}\ (a_1\mathbin{:}\tau_1)\ \Rightarrow\ e \mid \mathtt{inr}\ (a_2\mathbin{:}\tau_2)\ \Rightarrow\ e'\,\mathtt{end} \Downarrow^{\mathsf{e}} v',\sigma',\mathtt{T}}\ \textbf{(case)}$$

$$\frac{\sigma,m\mathbin{:}\mathtt{inr}\cdot\beta[v/a_2]e \quad \Downarrow^{\mathsf{e}} \quad v',\sigma',\mathtt{T}}{\sigma,m\mathbin{:}\beta,\mathtt{mcase}\,\mathtt{inr}_{\tau_1+\tau_2}v\,\mathtt{of}\,\mathtt{inl}\ (a_1\mathbin{:}\tau_1)\ \Rightarrow\ e \mid \mathtt{inr}\ (a_2\mathbin{:}\tau_2)\ \Rightarrow\ e'\,\mathtt{end} \Downarrow^{\mathsf{e}} v',\sigma',\mathtt{T}}\ \textbf{(case)}$$

Figure 14.5: Evaluation of expressions.

$$\frac{}{m : \beta, \epsilon \overset{find}{\rightsquigarrow} \epsilon, \epsilon} \qquad \frac{}{m : \beta, \mathtt{W}_{\tau}^{r}(l) \overset{find}{\rightsquigarrow} \epsilon, \epsilon} \qquad \frac{m : \beta, \mathtt{T} \overset{find}{\rightsquigarrow} \mathtt{T}', \mathtt{T}''}{m : \beta, \prec^{r_1} R_l^{x.t}\, \mathtt{T}\, ^{r_2}\succ \overset{find}{\rightsquigarrow} \mathtt{T}', \mathtt{T}''}$$

$$\frac{m = m' \wedge \beta = \beta'}{m : \beta, \{\, ^{r_1}\, \mathtt{T}\, ^{r_2}\, \}_{m' : \beta', v} \overset{find}{\rightsquigarrow} \{\, ^{r_1}\, \mathtt{T}\, ^{r_2}\, \}_{m' : \beta', v}, \epsilon} \qquad \frac{\begin{array}{c} m \neq m' \vee \beta \neq \beta' \\ m : \beta, \mathtt{T} \overset{find}{\rightsquigarrow} \mathtt{T}', \mathtt{T}'' \end{array}}{m : \beta, \{\, ^{r_1}\, \mathtt{T}\, ^{r_2}\, \}_{m' : \beta', v} \overset{find}{\rightsquigarrow} \mathtt{T}', \mathtt{T}''}$$

$$\frac{m : \beta, \mathtt{T}_1 \overset{find}{\rightsquigarrow} \mathtt{T}_1', \mathtt{T}_1'' \quad \mathtt{T}_1' \neq \epsilon}{m : \beta, \mathtt{T}_1 \,;\, \mathtt{T}_2 \overset{find}{\rightsquigarrow} \mathtt{T}_1', \mathtt{T}_1'' \,;\, \mathtt{T}_2} \qquad \frac{\begin{array}{c} m : \beta, \mathtt{T}_1 \overset{find}{\rightsquigarrow} \epsilon, \epsilon \\ m : \beta, \mathtt{T}_2 \overset{find}{\rightsquigarrow} \mathtt{T}_2', \mathtt{T}_2'' \end{array}}{m : \beta, \mathtt{T}_1 \,;\, \mathtt{T}_2 \overset{find}{\rightsquigarrow} \mathtt{T}_2', \mathtt{T}_2''}$$

Figure 14.6: The rules for memo look up.

$$\frac{}{\varsigma, \chi, \epsilon \stackrel{\curvearrowright}{\rightarrow} \epsilon, \chi, \varsigma}\ \textbf{(empty)}$$

$$\frac{\chi' = \chi \setminus \{l\}}{\varsigma, \chi, \mathtt{W}^{r}_{\tau}(l) \stackrel{\curvearrowright}{\rightarrow} \mathtt{W}^{r}_{\tau}(l), \chi', \varsigma}\ \textbf{(write)}$$

$$\frac{\begin{array}{c}(l \notin \chi)\\ \varsigma, \chi, \mathtt{T} \stackrel{\curvearrowright}{\rightarrow} \mathtt{T}', \chi', \varsigma'\end{array}}{\varsigma, \chi, \prec^{r_1} R^{x.t}_{l}\ \mathtt{T}\ {}^{r_2}\succ \stackrel{\curvearrowright}{\rightarrow} \prec^{r_1} R^{x.t}_{l}\ \mathtt{T}'\ {}^{r_2}\succ, \chi', \varsigma'}\ \textbf{(read, no change)}$$

$$\frac{\begin{array}{c}(l \in \chi)\\ (\alpha, \mu, \Upsilon_1{}^{\text{-}}) = \varsigma\\ (\alpha, \mu, \chi, \mathtt{T}, \Upsilon_1{}^{r_1}), [\alpha(l, r_1)/x]\ t \Downarrow^{\mathsf{t}} \star, (\alpha', \mu', \chi', \_, \Upsilon_2{}^{r_3}), \mathtt{T}'\\ \Upsilon_3 = \mathtt{delete}(\Upsilon_2, \mathtt{next}(\Upsilon_2, r_3), r_2)\\ \alpha'' = \alpha' \setminus \{(l, r) \mid l, r) \in \mathrm{dom}(\alpha) \wedge r_3 < r \leq r_2\}\\ \chi'' = \chi' \cup \{l \mid (l, r) \in \mathrm{dom}(\alpha) \wedge r_3 < r \leq r_2\}\end{array}}{\varsigma, \chi, \prec^{r_1} R^{x.t}_{l}\ \mathtt{T}\ {}^{r_2}\succ \stackrel{\curvearrowright}{\rightarrow} \prec^{r_1} R^{x.t}_{l}\ \mathtt{T}'\ {}^{r_3}\succ, \chi'', (\alpha'', \mu, \Upsilon_3{}^{r_3})}\ \textbf{(read, change)}$$

$$\frac{\begin{array}{c}\varsigma, \chi, \mathtt{T}_1 \stackrel{\curvearrowright}{\rightarrow} \mathtt{T}'_1, \chi', \varsigma'\\ \varsigma', \chi', \mathtt{T}_2 \stackrel{\curvearrowright}{\rightarrow} \mathtt{T}'_2, \chi'', \varsigma''\end{array}}{\varsigma, \chi, (\mathtt{T}_1\ ;\ \mathtt{T}_2) \stackrel{\curvearrowright}{\rightarrow} (\mathtt{T}'_1\ ;\ \mathtt{T}'_2), \chi'', \varsigma''}\ \textbf{(let)}$$

$$\frac{\varsigma, \chi, \mathtt{T} \stackrel{\curvearrowright}{\rightarrow} \mathtt{T}', \chi', \varsigma'}{\varsigma, \chi, \{\ {}^{r_1}\ \mathtt{T}\ {}^{r_2}\ \}_{m\,:\,\beta, v} \stackrel{\curvearrowright}{\rightarrow} \{\ {}^{r_1}\ \mathtt{T}'\ {}^{r_2}\ \}_{m\,:\,\beta, v}, \chi', \varsigma'}\ \textbf{(memo)}$$

Figure 14.7: Change propagation.

Part V

Implementation and Experiments

Introduction

This part studies the practical effectiveness of self-adjusting computation in two contexts. First we present a general-purpose ML library for writing self-adjusting programs. Using the library, we give self-adjusting versions of our applications and present an experimental evaluation. Second, we describe and evaluate an implementation of self-adjusting tree contraction in the C++ language.

In Chapter 15, we describe the general-purpose ML library and the implementation of our applications. The library enables the programmer to make any purely functional program self-adjusting by making small, methodological changes to ordinary (non-self-adjusting) code. The applications that we consider include combining values in a list, quick sort, merge sort, and Graham's Scan and quick hull algorithms for planar convex hulls. These applications are chosen to include a number of computing paradigms such as random-sampling, two forms of divide-and-conquer, and incremental result construction. Using our applications, we present an experimental evaluation of the library. The experiments show that self-adjusting programs written with our library has an overhead between four and ten compared to ordinary programs, and the overhead of change propagation is less than six. Under small changes to the input the self-adjusting programs can be arbitrarily faster than recomputing from scratch. For the input sizes we consider, our approach yields up to three orders of magnitude faster update times than recomputing from scratch.

Chapter 16 describes a library for kinetic data structures based on our general-purpose library. As applications, we consider kinetic convex hulls. The library makes it nearly trivial to obtain kinetic data structures from self-adjusting programs. The resulting kinetic data structures are composable and adjust to a broad range of external changes.

Chapter 17 describes a solution to the dynamic-trees problem that is obtained by applying self-adjusting computation techniques to the tree-contraction algorithm. We implement self-adjusting tree contraction using a version of our general-purpose library that is specialized for tree contraction. Both the library and the self-adjusting tree contraction are implemented in the C++ language. We present an experimental evaluation of library by considering a broad range of applications, including path queries, subtree queries, non-local search queries *etc.* We also present a direct comparison to the fastest previously proposed data structure for dynamic trees, known as Link-Cut Trees [91, 92]. The experimental results show that our implementation not only compares well to existing data structures but also is faster for certain applications. In addition, self-adjusting tree contraction directly supports batch, *i.e.*, multiple simultaneous, changes to the input. Our experiments show that processing batch changes can be asymptotically a logarithmic factor faster than processing them one by one as would be required by existing data structures.

Chapter 15

A General-Purpose Library

This chapter present a general purpose library for self-adjusting computation. The library is written in the ML language and enables transforming any ordinary purely functional program into a self-adjusting program methodically by using a small number of annotations. We study the effectiveness of the library by considering a number of applications and presenting an experimental evaluation.

15.1 The Library

We present an implementation of the SLf language as an ML library. The key difference between the library and the SLf language is that the library does not support memoization based on branches. Rather, the programmer is expected to supply the strict and the non-strict dependences explicitly. We decided not to implement support for branches, because it is difficult to ensure safe use of memoization primitives in the context of an ML library.

Figure 15.1 shows the interface to the library. The `BOXED_VALUE` module provides operations for boxing and un-boxing values, the `COMBINATORS` module provides combinators for modifiables and for memoization. The `META_OPERATIONS` library provides operations for performing external changes and for running change propagation.

To support constant-time equality tests, we use tag equality based on boxes. The `BOXED_VALUE` module supplies functions for operating on boxed values. The `new` function creates a boxed value by associating a unique integer *index* with a given value. The `eq` function compares two boxed values by comparing their indices. The `valueOf` and `indexOf` functions return the value and the index of a boxed value, respectively. The boxed value module may be extended with functions for creating boxed values for a type (*e.g.*, `fromInt`, `fromOption`). Such type-specific functions must be consistent with the equality of the underlying types. For example, the function `fromInt` may assign the index i to integer i.

The `COMBINATORS` module defines modifiable references (modifiables for short) and changeable computations. Every execution of a changeable computation of type α `cc` starts with the creation of a fresh modifiable of type α `modref`. The modifiable is written at the end of the computation. For the duration of the execution, the reference never becomes explicit. Instead, it is carried "behind the scenes" in a way that is strongly reminiscent of a monadic computation. Any non-trivial changeable computation reads one

```
signature BOXED_VALUE =
sig
  type index=int
  type 'a t

  val init:  unit -> unit
  val new:  'a -> 'a t
  val eq:  'a box* 'a t -> bool
  val valueOf:  'a t -> 'a
  val indexOf:  'a t -> index
  val fromInt:  int -> int t
  val fromOption:  'a t option -> 'a t option t
end

signature COMBINATORS =
sig
  type 'a modref
  type 'a cc

  val modref :  'a cc -> 'a modref
  val new :  'a -> 'a modref
  val write :  'a Box.t -> 'a Box.t cc
  val write' :  ('a * 'a -> bool) -> 'a -> 'a cc
  val read :  'b modref * ('b -> 'a cc) -> 'a cc

  val mkLift :  ('a *'a -> bool) ->
                (int list * 'a) -> ('a modref -> 'b) -> 'b
  val mkLiftCC : (('a * 'a -> bool) * ('b * 'b -> bool)) ->
                 (int list * 'a) -> ('a modref -> 'b cc) -> 'b cc
end

signature META_OPERATIONS =
sig
  type 'a modref

  val init:  unit -> unit
  val change:  'a Box.t modref -> 'a Box.t -> unit
  val change':  ('a * 'a -> bool) -> 'a modref -> 'a -> unit
  val deref :  'a modref -> 'a
  val propagate :  unit -> unit
end
```

Figure 15.1: Signatures for boxed values, combinators, and the meta operations.

or more other modifiables and performs calculations based on the values read.

Values of type α `cc` representing modifiable computations are constructed using `write`, `read`, and `mkLiftCC`. The `modref` function executes a given computation on a freshly generated modifiable before returning that modifiable as its result. The `write` function creates a trivial computation which merely writes the given value into the underlying modifiable. The `read` combinator, which we will often render as $\Longrightarrow$ in infix notation, takes an existing modifiable reference together with a receiver for the value read. The result of the `read` combinator is a computation that encompasses the process of reading from the modifiable, a calculation that is based on the resulting value, and a continuation represented by another changeable computation. Calculation and continuation are given by body and result of the receiver.

The key to efficient change propagation is the operation of skipping ahead (in constant time) to the earliest read of a modifiable containing a value that differs from the previous run. Fast-forwarding to such a read operation within computation c may skip the creation of the reference that c eventually writes to. As a consequence, the final `write` operation can result in another change to the value of a modifiable. Just like the earlier change that caused it, the new change gets recorded and propagated. To avoid unnecessary propagation, old and new values are compared for equality at the time of `write`.[1]

Functions `mkLift` and `mkLiftCC` handle memoization in the presence of non-strict dependencies. With ordinary memoization, a memo table lookup for a function call will fail whenever there is no precise match for the entire argument list. The idea behind non-strict dependencies is to distinguish between strict and non-strict arguments and base memoization on strict arguments only. By storing *computations* (as opposed to mere return values) in the memo table, a successful lookup can then be adjusted to any changes in non-strict arguments using the change-propagation machinery. Since change propagation relies on `read` operations on modifiables, the memoized function has to access its non-strict arguments via such modifiables. The memo table, indexed by just the strict part of the original argument list, remembers the modifiables set aside for non-strict arguments as well as the memoized computation.

Given the strict part of the argument list, a *lift operation* maps a function of type α `modref` $\rightarrow \beta$ to a function of type $\alpha \rightarrow \beta$ where α is the type of the non-strict argument.[2] Our `mkLift` and `mkLiftCC` combinators create lift operations for ordinary and changeable computations from appropriately chosen equality predicates for the types involved. The strict part of the argument list is represented by an index list, assuming a $1-1$ mapping between values and indices, *e.g.*, the one provided by the `BOXED_VALUE` interface. Not shown here, our library also contains `mkLift2`, `mkLiftCC2`, `mkLift3`, `mkLiftCC3` and so on to support more than one non-strict argument.

When its memo table lookup fails, a lifted function creates fresh modifiables containing its non-strict arguments, executes its body, and stores both the computation and the modifiables into the memo table. Computations are memoized by recording their return value and a representation of their dynamic dependence graph (DDG) using time stamps [3]. A successful memo lookup finds modifiables and a computation. The lifted function then writes its current non-strict arguments into their respective modifiables, and lets change propagation adjust the computation to the resulting changes.

It is the responsibility of the programmer to ensure that all applications of lift functions specify the strict and non-strict variables accurately. For correctness, it is required that all free variables of a memoized expression are specified as a strict or a non-strict variable. The classification of a variable as strict or non-strict does not affect correctness but just performance.

The `META_OPERATIONS` module supplies functions for performing external changes and for change propagation. The `change` and `change'` functions are similar to `write` and `write'` of the `COMBINATOR`

[1]Function `write'` is like `write` but uses an explicit equality predicate instead of insisting on boxed values.

[2]In the actual library, arguments appear in a different order to make idiomatic usage of the interface by actual code more convenient.

module. They change the underlying value of the modifiable to a new value—these are all implemented as side effects. The `change` function is a specialization of `change'` for boxed values. The `propagate` function runs the change-propagation algorithm. Change propagation updates a computation based on the changes issued since the last execution or the last change propagation. The meta operations can only be used at the top level. Using these operations inside a self-adjusting program is not safe in general.

15.2 Applications

We consider algorithms for combining values in a list, sorting, and convex hulls, and show how to implement them using our library. For sorting, we show how to implement quick sort and the merge sort algorithms. For convex hulls, we show how to implement the Graham's Scan and the quick hull algorithms.

An important property of our approach is that it requires little change to the ordinary (non-self-adjusting) code. All the examples considered here can be obtained from their ordinary versions in a few hours. The transformation of an ordinary program to a self-adjusting program follows a straightforward methodology that is explained in Chapter 13.

Although any purely functional program can be transformed into a self-adjusting program, not all programs perform efficiently under changes to their inputs. In particular, if a program is highly input-sensitive then it will not yield an efficient self-adjusting program. For example, a program that sums the elements in list by running the list from head to tail while computing partial sums is highly input sensitive—changing the first element in the list can change all the partial sums. Many of the applications that we consider in this thesis rely on randomization to reduce their input sensitivity.

In our implementation we rely on *random hash functions* [97] to generate random numbers. All the code that we show assumes that there is an ML structure, called `Hash`, with signature `sig new: unit -> int -> int end`. The structure supplies random binary hash functions from integers to the range $\{0, 1\}$. The motivation for using random hash function instead of pseudo random number generators is two fold. First, random hash functions reduce the number of random bits required by a computation. Second they enable matching the randomness between two independent computations. For all the applications that we consider here, the random hash functions can be replaced by a call to a random number generator without significantly affecting their performance.

15.2.1 Modifiable Lists

All the algorithms that we consider in this chapter operate on *modifiable lists*. A modifiable list is similar to a list but each tail element is stored inside of a modifiable. Modifiable lists enable the user to change their contents by changing the values stored in the tail elements using the `change` and `change'` operations provided by the library. Figure 15.2 shows the signature for modifiable lists, and an implementation as an ML structure. The modifiable list library provides the functions `filter` and `lengthLessThan`.

The `lengthLessThan` function takes a list and a number and returns `true` if the length of the list is less than the number, and returns `false` otherwise. The `filter` function takes a test function `f` and a list `l` and returns a list consisting of the elements that satisfy the test function. The `lengthLessThan` function is not memoized. The `filter` function is memoized strictly on the keys in the input list, and non-strictly on the tails. Therefore, if `filter` is passed a list `l` whose contents (the keys) is identical to that of a previously seen list `l'`, the function will return the same output as before, even if the tail modifiables in `l` and `l'` are not be the same.

15.2.2 Combining Values in a Lists

A fundamental primitive for computing with lists is a `combine` primitive that takes a binary operation and a list, and combines the values in the list by applying the binary operation. By choosing the binary operation to be applied, `combine` can be used to perform various operations on lists. For example, `combine` can be used to compute the minimum or maximum element or the longest increasing sequence in a list.

Figure 15.3 shows the code for a self-adjusting `combine` function. The underlying algorithm is very similar to the algorithm described in Section 9.1 but is slightly more general, because it does not require an identity element. The `combine` subroutine only assumes that the given binary operation is associative—commutativity is not required.

A straightforward way to apply a binary operation on a list is to walk down the list while computing the partial results for the prefix visited so far. Although this technique is simple, it is highly sensitive to changes. In particular, changing the element at the head of the list can change all the partial results and therefore the whole computation may have to be redone. The `combine` code presented here uses a randomized approach instead. The computation is performed by halving the original list into smaller and smaller lists until the list consists of a single element. To halve a list, the program chooses a randomly selected subset of the list and deletes the chosen elements from the list. Deleting an element involves incorporating its value to the closest surviving element to the left of the element. This technique ensures that insertions/deletions to the list can be handled in logarithmic time in the size of the input list.

15.2.3 Sorting

This section consider the quick sort and the merge sort algorithms, and shows how to make them self-adjusting. Figure 15.4 shows the code for the self-adjusting quick sort algorithm. The code is very similar to the ordinary quick sort. To avoid linear time concatenations the algorithm uses an accumulator for storing the sorted tail. The code relies on the `filter` function provided by the modifiable list library. The underlying algorithm is identical to the algorithm presented in Section 9.3.

The only difference between this code and the ordinary quick sort is that this code passes two additional arguments to the main function (`qsort`). These arguments consist of the `prev` and `next` keys that define the interval that the function call will fill in the output list. The arguments ensure that the program is *monotone* for insertions and deletions to the input. Monotonicity ensures that the resulting algorithm self-adjusts in expected $O(\log n)$ time to an insertion/deletion (at a uniformly randomly selected position in the list). Omitting the additional arguments yields an expected $O(\log^2 n)$-time algorithm.

Figure 15.5 shows the code for the self-adjusting merge sort algorithm. The code is obtained from an ordinary version of randomized merge sort by applying the transformation methodology. The underlying algorithm is identical to the algorithm presented in Section 9.2. The difference between the randomized and non-randomized merge sort algorithms is the way the input list is split. The standard algorithm splits the list by putting the elements at odd positions into one list and the elements at even positions in another. This technique is highly sensitive to input changes, because an insertions or deletion from the list can shift a large number of the old and even positions. To avoid this, the randomized merge-sort algorithm selects a random bit (flips a coin) for each element and places the elements that draw the same bit (see the same face) into the same sublist. Randomized splitting suffices to ensure expected $O(\log n)$ time updates under insertions/deletions into/from the input.

15.2.4 Convex Hulls

The *convex hull* of a set of points is the closest polygon that encloses these points. A fundamental problem of computational geometry is to find the convex hull of a set of points. The so-called dynamic convex hull problem requires computing the convex hull of a set of points under insertions and deletion to the input. Both of these problems have been studied extensively in the fields of computational geometry and algorithms [42, 73, 75, 69, 28, 20].

This section describes the self-adjusting versions of the Graham Scan and the quick hull algorithms for computing convex hulls. Both algorithms rely on an ML module that supplies some geometric primitives. The signature for this module is show below. It is straightforward to provide an implementation for this interface.

```
signature POINT =
sig
  type t
  val toLeft :  t * t -> bool
  val toRight :  t * t -> bool
  val leftTurn :  t * t * t -> bool
  val distToLine:  t * t -> t -> real
  val fromCoordinates :  real*real -> t
  val toCoordinates :  t -> real*real
end
```

The Graham's Scan algorithm is sorting based. It first sorts the set of points from right to left, and then computes convex hull by incrementally adding each point to the current hull one-by-one in sorted order. Figure 15.6 shows the self-adjusting version of this algorithm. The code is very similar to the ordinary Graham Scan algorithm and is obtained by applying the standard transformation. Note that the program uses the self-adjusting quick sort algorithm to sort the input points. The underlying algorithm for the graham-scan code is identical to the algorithm presented in Section 9.4. The implementation handles insertions/deletions into/from the input lists at uniformly random positions in expected $O(\log n)$ time.

Figure 15.7 shows the code for the self-adjusting quick-hull algorithm. The code is very similar to the ordinary quick-hull algorithm and is obtained by applying the standard transformation. The ordinary quick hull algorithm is known to work well in practice, but in the worst case it can take $\Theta(n^2)$ time. Similarly, the self-adjusting quick-hull algorithm can take $\Theta(n^2)$ time for a single insertion/deletion. Our experiments show, however, that it too can perform well in practice.

The quick-hull algorithm starts by finding the leftmost and the rightmost points in the input, and calls `split` with the line defined by these points. The `split` function takes a line defined by two points `p1` and `p2`, and filters out the points below the line `(p1,p2)`. The function then finds the point, `max`, farthest away from the line and performs two recursive calls with the lines `(p1,max)` and `(max,p2)` respectively.

The algorithm relies on the `ListCombine` structure for finding the leftmost and the rightmost points, as well as the point farthest away from a line.

```
signature MOD_LIST =
sig
  datatype 'a modcell = NIL | CONS of ('a * 'a modcell Comb.modref)
  type 'a t = 'a modcell Comb.modref

  val write:  'a Box.t modcell -> 'a Box.t modcell Comb.cc
  val eq:  'a Box.t modcell * 'a Box.t modcell -> bool
  val lengthLessThan :  int -> ('a t) -> bool Comb.modref
  val filter:  ('a Box.t -> bool) -> 'a Box.t t -> ('a Box.t t)
end

structure ModList:MOD_LIST =
struct
  structure C = Comb
  datatype 'a modcell = NIL | CONS of ('a * 'a modcell C.modref)
  type 'a t = 'a modcell C.modref

  infix ==> val op ==> = C.read

  fun write c = C.write' eq c
  fun eq (a,b) =
    case (a,b) of
      (NIL,NIL) => true
    | (CONS(ha,ta), CONS(hb,tb)) => Box.eq(ha,hb) andalso (ta=tb)
    | _ => false

  fun lengthLessThan n l =
    let val write = C.write' (fn (a,b) => (a=b))
        fun f(n, l) =
          if (n<1) then write false
          else l ==> (fn c =>
                        case c of
                          NIL => write true
                        | CONS(h,t) => f(n-1,t))
    in C.modref (f (n,l)) end

  fun filter f l =
    let val lift = C.mkLift eq
        fun filterM c =
          case c of
            NIL => (write NIL)
          | CONS(h,t) =>
              t ==> (fn ct => lift ([Box.indexOf h],ct) (fn t =>
                if (f h) then write (CONS(h,C.modref (t ==> filterM)))
                else t ==> (fn ct => filterM ct)))
    in C.modref (l ==> filterM) end
end
```

Figure 15.2: The signature for modifiable lists and an implementation.

```
structure ListCombine =
struct
  exception EmptyList
  structure ML = ModList
  structure C = Comb
  infix ==> val op ==> = C.read

  (*combine:('a Box.t*'a Box.t->'a Box.t)->('a Box.t ML.t->'a Box.t C.modref *)
  fun combine binOp l =
  let
    fun halfList l =
      let
        val hash = Hash.new ()
        fun pairEqual ((b1,c1),(b2,c2)) = Box.eq(b1,b2) andalso ML.eq(c1,c2)
        val writePair = C.write' pairEqual
        val lift = C.mkLiftCC (ML.eq,ML.eq)

        fun half c =
          let fun sumRun(v,c) =
                case c of
                  ML.NIL => writePair (v,c)
                | ML.CONS(h,t) => t ==> (fn ct =>
                  if (hash(Box.indexOf h) = 0) then writePair (binOp(v,h),ct)
                  else sumRun(binOp(v,h),ct)
          in case c of
               ML.NIL => ML.write ML.NIL
             | ML.CONS(h,t) => t ==> (fn ct =>
               lift ([Box.indexOf h],ct) (fn t => t ==> (fn ct =>
                 let val p = C.modref (sumRun (h,ct))
                 in p ==> (fn (v,ct') => ML.write (ML.CONS(v, C.modref (half ct'))))
                 end)))
          end

      in C.modref (l ==> (fn c => half c)) end

    fun comb l =
      ML.lengthLessThan 2 l ==> (fn b =>
        if b then l ==> (fn c =>
          case c of
            ML.NIL => raise EmptyList
          | ML.CONS(h,_) => C.write h)
        else
          comb (halfList l))
      in C.modref (comb l) end
end
```

Figure 15.3: Self-adjusting list combine.

```
structure QuickSort =
struct
  structure ML = ModList
  structure C = Comb
  infix ==> val op ==> = C.read

  fun qsort (compare:'a*'a->order) (l:'a Box.t ML.t):'a Box.t ML.t =
    let
      val lift = C.mkLiftCC2 (ML.eq,ML.eq,ML.eq)
      fun qsortM (l,cr,(prev,next)) = l ==> (fn c =>
        case c of
          ML.NIL => ML.write cr
        | ML.CONS(h,t) => t ==> (fn ct =>
            lift ([Box.indexOf h,
                   Box.indexOf prev,
                   Box.indexOf next],ct,cr) (fn (t,rest) =>
              let
                fun fl k = (compare(Box.valueOf k,Box.valueOf h))=LESS
                fun fg k = (compare(Box.valueOf k,Box.valueOf h))=GREATER
                val les = ML.filter fl t
                val grt = ML.filter fg t
                val bh = Box.fromOption (SOME h)
                val mid = C.modref (rest ==> (fn cr => qsortM (grt,cr,(bh, next))))
              in
                qSortM (les,ML.CONS(h,mid),(prev,bh))
              end)))
    in
      qsortM (l, ML.NIL, (Box.fromOption NONE, Box.fromOption NONE))
    end
end
```

Figure 15.4: Self-adjusting quick sort.

```
structure MergeSort =
struct
  structure ML = ModList
  structure C = Comb
  infix ==> val op ==> = C.read

  fun split(l:'a Box.t ML.t):'a Box.t ML.t*'a Box.t ML.t =
    let val hash = Hash.new ()
        val even = ML.filter (fn h => hash(Box.indexOf h)=0) l
        val odd = ML.filter (fn h => hash(Box.indexOf h)=1) l
    in (even,odd) end

  fun merge (compare:'a*'a -> bool) (a:'a Box.t ML.t, b:'a Box.t ML.t):'a ML.t =
    let
      val (lift1,lift2) = (C.mkLift2 (ML.eq,ML.eq), C.mkLift2 (ML.eq,ML.eq))
      fun mergeM (ca,cb) =
        case (ca,cb) of
          (ML.NIL,_) => ML.write cb
        | (_,ML.NIL) => ML.write ca
        | (ML.CONS(ha,ta),ML.CONS(hb,tb)) =>
            case compare(Box.valueOf ha,Box.valueOf hb) of
              LESS => ta ==> (fn cta =>
                      lift1 ([Box.indexOf ha],cta,cb) (fn (ta,lb) =>
                let val r = C.modref (ta ==> (fn cta =>
                                      lb ==> (fn cb =>
                                      mergeM (cta,cb))))
                in ML.write (ML.CONS(ha,r)) end))
            | _ => tb ==> (fn ctb =>
                  lift2 ([Box.indexOf hb],ctb,ca) (tb,la) =>
                let val r = ML.modlist (tb ==> (fn ctb =>
                                        la ==> (fn ca =>
                                        mergeM (ca,ctb))))
                in ML.write (ML.CONS(hb,r)) end))
    in a ==> (fn ca => b ==> (fn cb => mergeM (ca,cb))) end

  fun msort (compare:'a*'a->bool) (l:'a Box.t ML.modlist):'a Box.t ML.modlist =
    let
      fun msortM (l) =
        C.modref ((ML.lengthLessThan 2 l) ==> (fn b =>
          if b then l ==> ML.write
          else
            let val (even,odd) = split (l)
            in (merge compare (msortM even, msortM odd))
            end))
    in (msortM l) end
end
```

Figure 15.5: Self-adjusting merge sort.

```
structure GrahamScan =
struct
  structure C = Comb
  structure ML = ModList
  structure P = Point
  infix ==> val op ==> = C.read

  fun halfHull (test:P.t*P.t -> bool) (l:P.t Box.t ML.t):P.t Box.t ML.t =
    let
      fun trim (p,c) =
        let
          val (lift1,lift2) = (C.mkLift ML.eq,C.mkLift ML.eq)

          fun trimM c =
            case c of
              ML.NIL => ML.write c
            | ML.CONS(pa,lb) => lb ==> (fn cb =>
                case cb of
                  ML.NIL => lift1 ([Box.indexOf pa],cb)
                                  (fn lb => ML.write (ML.CONS(pa,lb)))
                | ML.CONS(pb, tb) =>
                  lift2 ([Box.indexOf pa, Box.indexOf pb],cb) (fn lb =>
                    if test (Box.valueOf pb,Box.valueOf pa,Box.valueOf p) then
                      ML.write (ML.CONS(pa,lb))
                    else
                      lb ==> (fn cb => trimM cb)))
        in trimM c end

      fun scan (l:P.t Box.t ML.t):P.t Box.t ML.t =
        let val lift = C.mkLiftCC2 (ML.eq,ML.eq,ML.eq)
            fun scanM (c,ch) =
              case c of
                ML.NIL => (ML.write ch)
              | ML.CONS(h,t) =>
                t ==> (fn ct => lift ([Box.indexOf h],ct,ch) (fn (t,hull) =>
                  let val ch' = ML.CONS(h, C.modref (hull==>(fn ch=>trim (h,ch))))
                  in t ==> (fn ct => scanM (ct,ch')) end))
        in C.modref (l ==> (fn c => scanM (c, ML.NIL))) end
    in (scan l) end

  fun grahamScan (l:P.t Box.t ML.t):  P.t Box.t ML.t =
    let fun toLeft (a,b) = if P.toLeft (a,b) then LESS else GREATER)
        val ll = sort toLeft l
    in halfHull P.leftTurn ll end
end
```

Figure 15.6: Self-adjusting Graham Scan.

```
structure QuickHull =
struct
  structure C = Comb
  structure ML = ModList
  structure LC = ListCombine
  structure P = Point
  infix ==> val op ==> = C.read

  fun split (rp1, rp2, ps, hull) =
    let
      val lift = C.mkLiftCC2 (ML.eq,ML.eq,ML.eq)

      fun splitM (p1, p2, ps, hull) =
        let val (v1,v2) = (Box.valueOf p1, Box.valueOf p2)
            fun select p = P.distToLine (v1,v2) (Box.valueOf p) > 0.0
            val l = ML.filter select ps
        in l ==> (fn cl =>
          case cl of
            ML.NIL => ML.write (ML.CONS(p1,hull))
          | ML.CONS(h,t) => hull ==> (fn chull =>
            lift ([Box.indexOf p1,Box.indexOf p2],cl,chull) (fn (l,hull) =>
              let
                fun select (a,b) =
                  if (P.distToLine (v1, v2) (Box.valueOf a) >
                      P.distToLine (v1, v2) (Box.valueOf b))
                  then a
                  else b
                val rmax = LC.combine select l
                val rest = C.modref (rmax ==> (fn max => splitM (max,p2,l,hull)))
              in rmax ==> (fn max => splitM (p1,max,l,rest)) end)))
        end
    in rp1 ==> (fn p1 => rp2 ==> (fn p2 => splitM (p1,p2,ps,hull))) end

  fun qhull l =
    C.modref ((ML.lengthLessThan 2 l) ==> (fn b =>
      if b then ML.write ML.NIL
      else
        let
          fun select f (a,b) =
            if f (Box.valueOf a, Box.valueOf b) then a
            else b

          val min = LC.combine (select P.toLeft) l
          val max = LC.combine (select P.toRight) l
          val h = C.modref (max ==> (fn m => ML.write (ML.CONS(m,C.new ML.NIL))))
        in split (min, max, l, h) end))
end
```

Figure 15.7: Self-adjusting Quick Hull.

15.3 Implementation and Experiments

We describe an implementation of our library and present some experimental results. The experiments confirm the complexity bounds proven in Chapter 9 and show that

- the overhead of self-adjusting programs is no more than about 10,
- the overhead for change propagation is less than 6, and
- the speed up (time for change propagation divided by the time for rerunning from scratch) for insertions/deletions is as much as three orders of magnitude. Note that the speed up for all our applications is asymptotically linear—the numbers we report here are determined by the input sizes considered.

We note that our implementation performs no optimizations specific to our library. Our experiments with hand-coded optimizations show that there is a lot of room for improvement. For example, we are able to reduce the overhead of quick sort (for integer sorting) from 10 to less than 4 using two optimizations based on tail-call identification and inlining of library functions.

15.3.1 Implementation

The complete code for the library is given in Appendix A. The implementation relies on memoized DDGs and the memoized change propagation algorithms and is based on the data structures described in Chapter 6. One key difference is that the implementation deletes dependence edges (reads) lazily. Instead of discarding deleted reads immediately, the implementation marks them. When a marked read is encountered, it is discarded. The lazy deletion eliminates the need to keep back pointers from the time stamps to the reads. One problem with this approach is that marked reads can accumulate over time. In the applications that we consider, this problem does not arise. In general, however, deleted reads must be discarded immediately.

15.3.2 Experiments

For each application, we measure the following three quantities:

Time for initial run: This experiment measures the total time it takes to run a self-adjusting program on a given input. To determine the overhead of our techniques, we divide this time by the time for running the non-self-adjusting version of the same program.

Time for incremental insertions: This experiment measures the time it takes to construct the output by performing incremental (*a.k.a.*, online) insertions. Given an input list l, we construct an empty modifiable list m and run our application on m. We then insert the next value from l at the end of m, and perform a chance propagation. We continue this insert-and-propagate cycle until all the elements in l are inserted into m.

Average time for a deletion/insertion: This experiment measures the average time it takes to perform an insertion/deletion. We start by running a self-adjusting application on a given input list. We then delete the first element in the input list and perform a change propagation. Next, we insert the element back into the list and perform a change propagation. We perform this delete-propagate-insert-propagate operation for each element. Note that after each delete-propagate-insert-propagate operation, the input list is the same as the original input. We compute the average time for an insertion/deletion as the ratio of the total time to the number of insertions and deletions ($2n$ for an input of size n).

15.3.3 Generation of inputs

Given an input size n, we generate the input instances randomly. For integer sorting applications, we generate a random permutation of a list containing the values from 1 to n. For convex hulls, we uniformly chose points from a square of size $10n \times 10n$. As a random number generator, we use the `Random.rand` function supplied by the SML library.

15.3.4 Experimental Results

We ran our our experiments on a desktop computer with a Pentium 4 processor clocked at 2.2 GHz and with one gigabytes of memory; the machine runs the Red-Hat Linux operating system. All experiments were performed using the SML/NJ version 110.0.7. This is an older SML/NJ compiler, but we have found its garbage collector faster than the most recent version (version 110.53).

For each application, we ran our benchmarks for measuring the time for initial run, time for incremental insertions, and average time for a deletion/insertion. To account for the time for garbage collection, we measured the time for major and minor garbage collections, and the time spent solely by the application separately. Figure 15.8 shows the timings for the quick sort algorithm; Figure 15.9 shows the timings for the merge sort algorithm; Figure 15.10 shows the timings for the Graham's Scan algorithm; Figure 15.11 shows the timings for the quick hull algorithms.

As can be seen from the figures, the time for garbage collection is excessively high. The total time for major and minor collections often constitute 75% of the total time. When the total live data of an application is reasonably high, the SML/NJ garbage collector is known to perform poorly [19, 86]. Other research groups experimenting with the compiler have encountered similar problems in different application domains. We have also experimented with the most recent version of the compiler, version 110.53, and found that the applications spend even more time garbage collecting.

We are able to run the experiments for measuring the average time for insertion/deletion for sizes that are half or less as much as we could for initial run (the only exception is the quick hull algorithm). For larger sizes, the system runs out of memory. We expect that this is because of a known garbage-collection leak in the SML/NJ version 110.0.7 compiler. In long-running applications, this slow leak eventually causes the system to run out of physical memory.

These results show that the garbage collector for the SML/NJ language is not appropriate for our setting. For the two reasons described above, we only consider the application time when evaluating our experiments. From our experiments we deduce the following conclusions.

The overhead of the library: The overhead of the library is defined as the ratio of the time for an initial run of a self-adjusting program to the time for an initial run of the ordinary, non-self-adjusting version of the program. We have found that this overhead is between 5 and 10 for our applications.

We note that our implementation performs no optimizations specific to our library. To assess the potential for improvement we implemented a hand-optimized version of the quick sort algorithm. The optimizations reduce the overhead from 10 to about 4. We used two optimizations. The first optimization allows tail calls to share their end time stamps. This reduces the total number of time stamps created during an execution. The second optimization inlines the `lift` functions and eliminates the modifiables created for lifted values. These optimization can be performed by a compiler or static analyzer. We plan to study optimization techniques in more depth in the future.

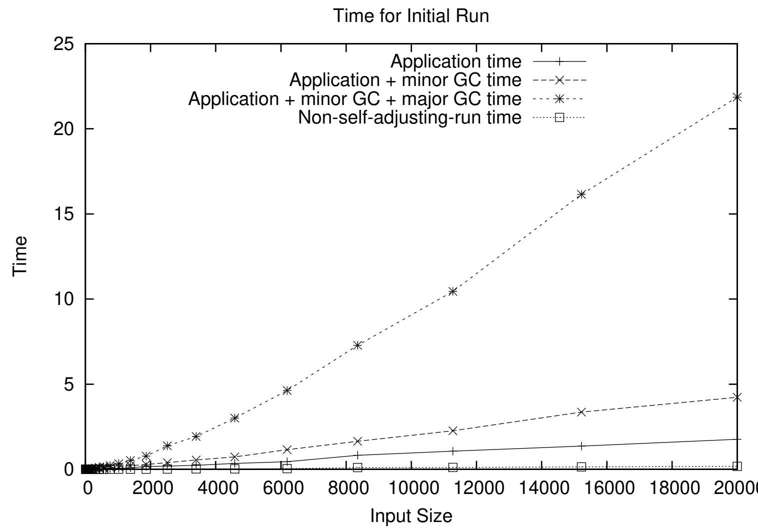
The overhead of change propagation: We measure the overhead of change propagation as the ratio of the time for the incremental run to the time for initial run. For an input with n keys, the incremental run performs n change propagations to yield the same output as the initial run. For all our applications, this ratio is between 4 and 6.

The asymptotic performance: The experiments confirm the theoretical bounds that we have obtained in Chapter 9. For the quick-sort and merge-sort applications, the average time for an insertion/deletion is bounded by $O(\log n)$ as the comparison to the $0.0003 \log n$ shows. Since the Graham Scan application uses merge sort as a substep, and since otherwise the algorithm is linear time, the time for average insertions/deletions is marginally more than the same quantity for merge sort (this can be seen comparing the timings to the $0.0003 \log n$ curve). Note that, we have no asymptotic bounds for the quick hull application. Quick hull can require $\Theta(n^2)$-time for a single insertion or deletion.

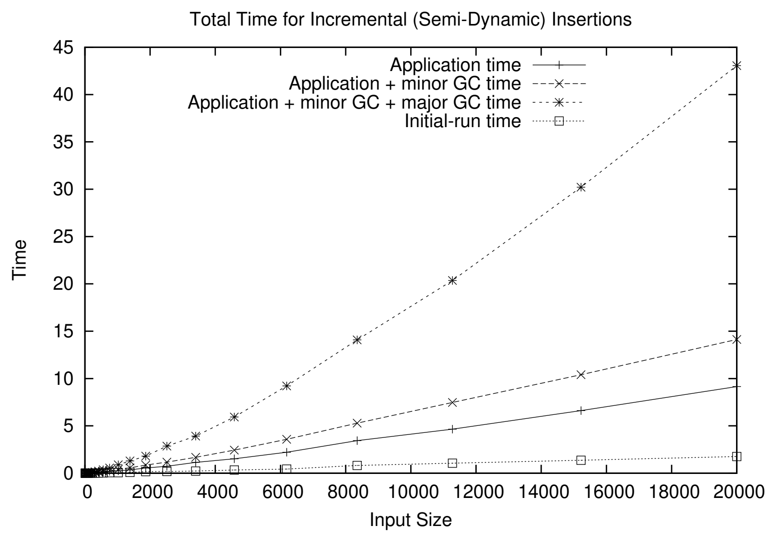
Speed up: For an insertion/deletion our techniques yield up to three orders of magnitude speed up over rerunning from scratch. For example, with merge sort and Graham Scan algorithms, average time for insertions/deletions is up to 250 times faster than rerunning from scratch. For the quick sort algorithm the average speed up can be as much as a factor of 150. For the quick hull algorithms the average speed up can be as much as a factor of 400. Note that these are average numbers. In general certain updates are much faster than others. For example, in quick sort, insertions at the end are up to three orders of magnitude faster than rerunning from scratch. Note also that the speed up is asymptotically large (linear for all our applications). The values we report here are determined by the input size.

Consider the experiments with our integer sorting algorithms Figures 15.8 and 15.9. The time for the initial run, and for incremental insertions with merge sort is nearly twice as large as those with the quick sort algorithm. For insertion/deletions, however, the merge sort algorithm is about 50% faster than quick sort. This is consistent with our trace-stability results. As shown in Chapter 9, the merge sort algorithm requires expected $O(\log n)$ time for an arbitrary insertion/deletion, whereas the quick sort algorithm can require $\Theta(n)$ time.

Of the two convex hull algorithms, the Graham Scan algorithm is the only theoretically efficient algorithm. As shown in Section 9.4, the algorithm requires $O(\log n)$ average time for an insertion/deletion. The quick hull algorithm can take $\Theta(n^2)$ time. Our experiments (Figures 15.10 and 15.11) show that the quick hull algorithm is significantly faster than Graham Scan for an insertion/deletion. Indeed, like the quick sort algorithm, the quick hull algorithm is known to perform well in practice. The quick hull algorithm, however, is highly input sensitive—the performance can vary significantly depending on the input.



.eps



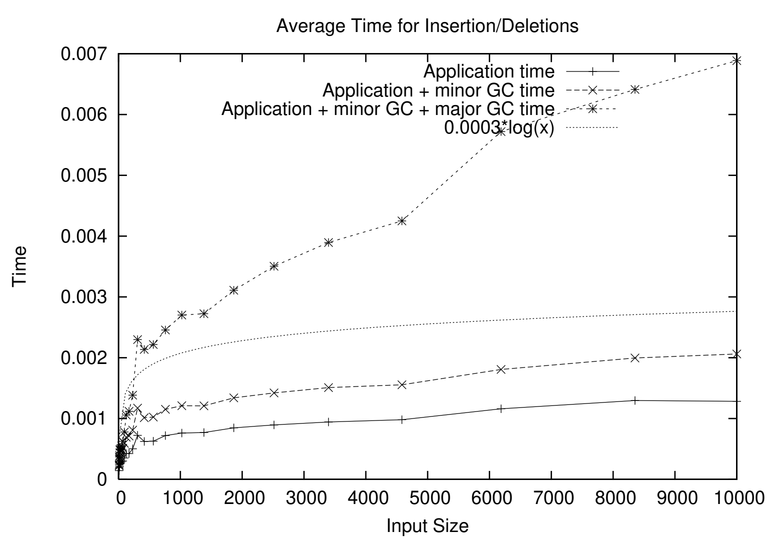


Figure 15.8: Timings for quick sort.

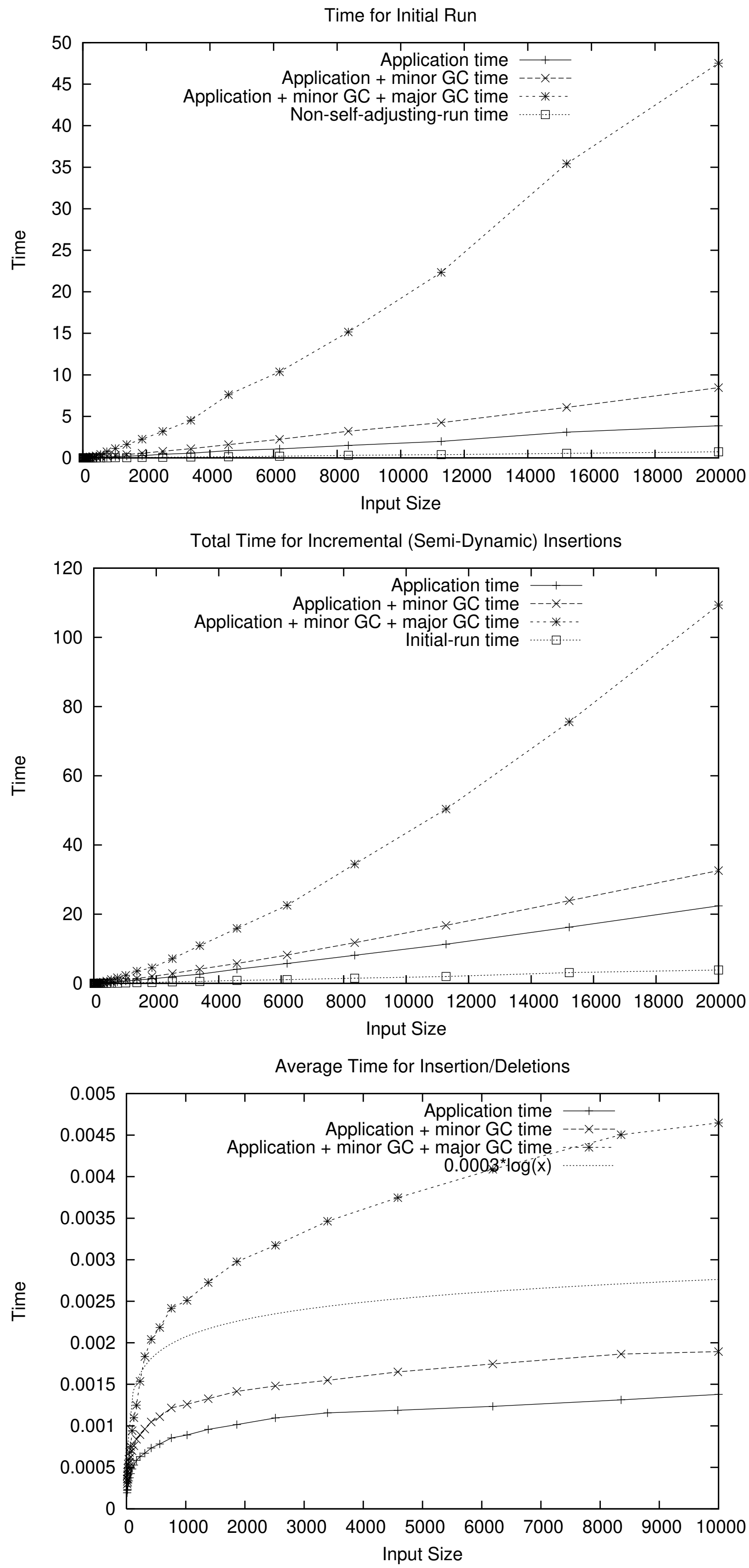


Figure 15.9: Timings for merge sort.

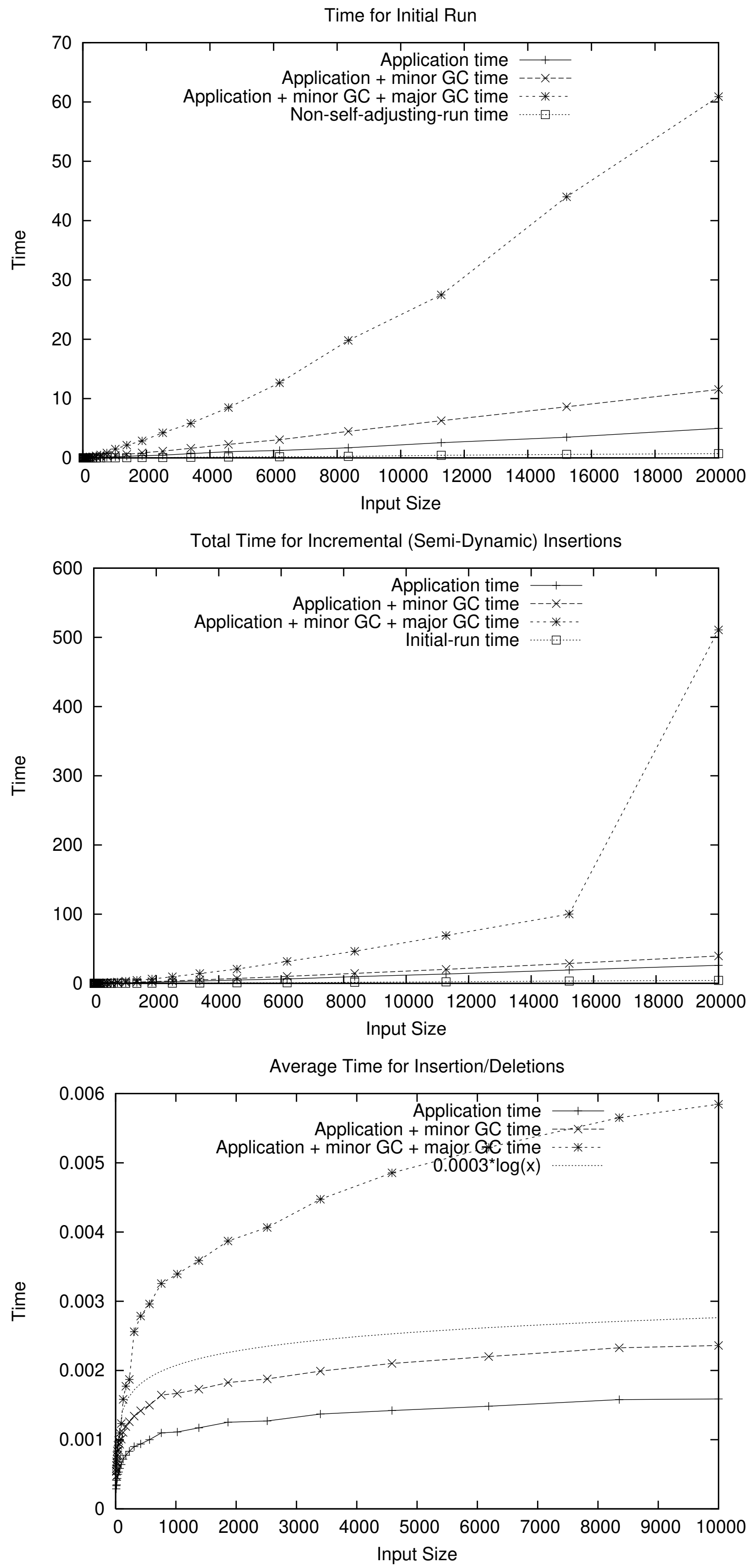


Figure 15.10: Timings for Graham's Scan.

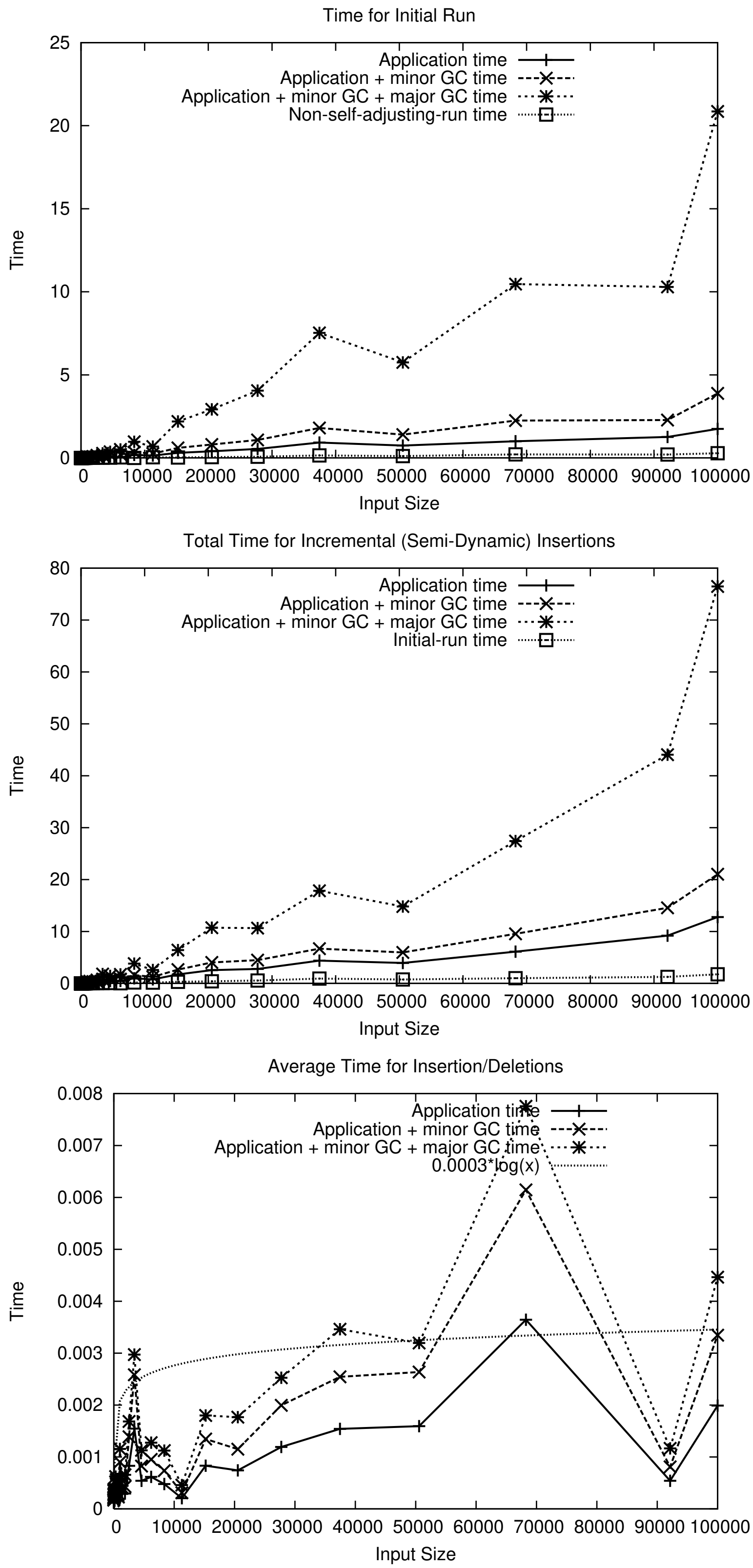


Figure 15.11: Timings for quick hull.

Chapter 16

Kinetic Data Structures

Kinetic data structures [43, 13, 14, 7, 55, 25, 54] compute properties of continuously moving objects. In this chapter, we describe a library for writing kinetic programs based on self-adjusting computation. The library enables the programmer to transform an ordinary program into a kinetic program by applying a methodical transformation. Our approach addresses two key limitations of previous work on kinetic data structures. First our kinetic programs adjust to any change, including insertions/deletions and flight plan updates. Second our kinetic programs are composable. These properies are critical to the development of kinetic data structures. To address the limitations of previous on kinetic data structures, Guibas suggests developing the interaction between dynamic and kinetic algorithms [43]. We know of no prior work that addresses these limitation.

16.1 Background

To compute a property of moving objects, kinetic data structures take advantage of the continuity of motion. The idea is to obtain a "proof of correctness" for the property by running an ordinary program P that computes the property on the set of object assuming that they are stationary, and then "simulate" this proof over time. The proof consists of comparisons between objects. Suppose that for each comparison we compute the *failure time*, *i.e.*, the earliest time at which the comparison fails. Consider the earliest failure time t. Since none of the comparisons change value until time t, the property remains unchanged until that time. We can thus advance the time to t without making any combinatorial changes to the property or the proof. At time t, we compute a new proof from the old by changing the value of the failing comparison and running an *update algorithm*. To simulate the proof we repeat this advance-and-update until the earliest failure time is infinity. To determine the failure times of comparisons, we exploit the continuity of motion. In particular, if we know that the motion parameters of the objects (velocity, direction, acceleration *etc.*), then we can solve for the failure time of a comparison.

A kinetic data structure consists of an *event handler* that drives the simulation, and an *update algorithm* that, given a failing comparison and the old proof, generates a new proof. The proof consists of certificates. A *certificate* is a comparison associated with a failure time. The event handler maintains the certificates in a priority queue and performs the advance-and-update steps. At each step, the event handler removes the certificate with the earliest failure time from the queue, changes the outcome of the certificate, and updates the proof by running the update algorithm. Since the event handler only interacts with the certificates and the update algorithm, it is generic for all kinetic data structures.

The key component of a kinetic data structure is the update algorithm. The update algorithm is a dynamic algorithm that updates an existing proof according to a certificate change. In previous work, the update algorithms and the representation of the proofs are devised on a per problem basis. This approach has two key limitations.

One limitation of kinetic data structures is that they are often designed to support certificate failures but not *discrete changes* such as insertions/deletions, or *flight-plan* changes that change the motion parameters of the object. Another limitation of kinetic data structures is that they are not composable. Guibas points out [43] that supporting discrete changes and composition is critical for the development of kinetic data structures. For example, we can obtain a kinetic data structure for a number of extent problems by combining a kinetic convex hull data structure and a kinetic tournament data structure [43]. To be composable, kinetic data structure must support discrete changes, and must be able to detect discrete changes to their state. To address these limitations, Guibas suggests that the interaction between kinetic and dynamic algorithms needs to be better developed [43]. Note, however, that dynamic data structures themselves are not composable. We know of no prior work that addresses these limitations.

16.2 Our Work

We propose to obtain kinetic programs from self-adjusting programs automatically. The idea is to use a self-adjusting program to compute the property of interest and then use change propagation to update the property when a certificate changes. More concretely, instead of calling an *ad hoc* update algorithm after changing the outcome of a comparison, the event handler will call change propagation. Our approach addresses the limitations of existing work on kinetic data structures. Kinetic self-adjusting programs are composable and they support arbitrary changes to their state.

To determine the effectiveness of kinetic programs, we implemented a library for transforming self-adjusting programs into kinetic programs and applied the library to kinetic convex hulls. The library supplies an event handler, and a set of geometric tests, called kinetic tests. A *kinetic test* creates a certificate by computing the failure time of the test.

The library enables transforming an ordinary (non-kinetic) program into a kinetic program in two steps. First, the programmer makes the program self-adjusting by applying the standard ordinary-to-self-adjusting transformation. Second, the programmer replaces ordinary geometric tests with the kinetic tests. Since the second step is trivial—it can be done automatically or by simply linking the application with the library that supplies kinetic tests, self-adjusting programs are automatically kinetic. For example, transforming the self-adjusting quick-hull program in Figure 15.7 requires replacing the line `structure P = Point` with `structure P = KineticPoint` and replacing the distance comparison with a kinetic geometric test. The `KineticPoint` supplies kinetic tests for points.

Using our library, we have obtained kinetic versions of our convex hull algorithms. At the time of writing, we do not have a careful experimental evaluation but have some videos that we used for testing. Figure 16.1 shows some snapshot from a test that we ran with our program. The experiment starts out with five object that move continuously within some bounding box. When an object hits the box, it bounces off based on known rules of physics. These bounce events change the flight-plan of the object by changing its direction. At every certificate failure, in addition to running change propagation, the event handler also

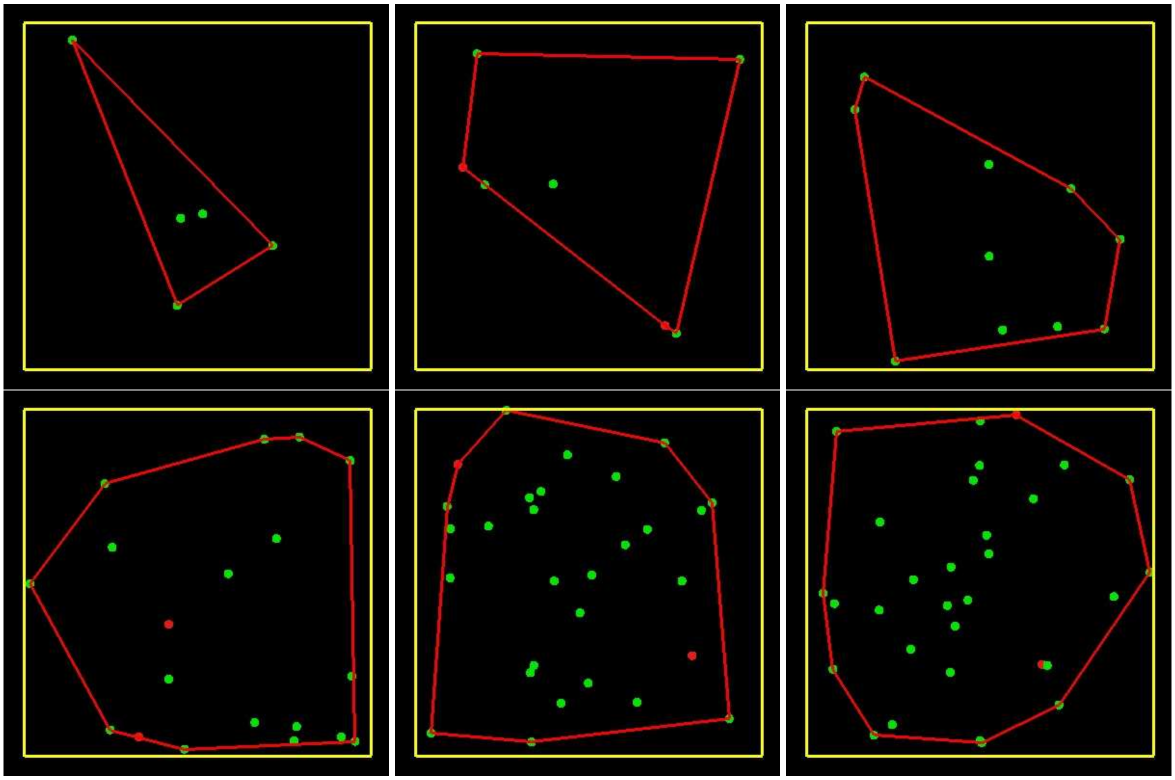


Figure 16.1: Snapshots from a kinetic quickhull simulation.

inserts a number of new moving points and deletes a number of existing points into the input. The number of points to be inserted and deleted are determined randomly. For this example, the number of inserted points are expected to be four times as much as the number deleted points.

Chapter 17

Dynamic Trees

In Chapter 10, we show that applying change propagation on static tree-contraction algorithm of Miller and Reif [62] yields a solution to the dynamic trees-problem of Sleator and Tarjan [91]. In this chapter, we present an implementation of this solution and present an experimental evaluation. The contributions are twofold. First, by applying self-adjusting computation to a difficult problem, we present experimental evidence that self-adjusting computation performs well even when compared to direct implementations of problem-specific solutions. As a second contribution, we present a library for dynamic-trees that support a general interface. The code for the library and the applications is publicly available at the Aladdin Center web site at `www.aladdin.cs.cmu.edu`.

Dynamic-trees data structures are fundamental structures that arise as a substep in many important algorithms such as dynamic graph algorithms [49, 32, 46, 38], max-flow algorithms [91, 92, 39], computational geometry algorithms [40]. Several dynamic-tree data structures have been proposed including Link-Cut Trees [91, 92], Euler-Tour Trees [46, 96], Topology Trees [38], and Top Trees [10, 11, 95]. We implement change propagation on tree-contraction as a solution to the dynamic trees problem and present an experimental evaluation of the approach. For the evaluation we consider a broad range of applications including path queries, subtree queries, least-common-ancestor queries, maintenance of centers and medians of trees, nearest-marked-vertex queries, semi-dynamic minimum spanning trees, and the max-flow algorithm of Sleator and Tarjan [91].

The approach that we take in this chapter is different than the previous. In the previous chapter, we applied self-adjusting computation to a number of problems using a general-purpose, purely functional library written in the ML language. In this chapter, we consider an imperative language, C++, and implement a version of our general-purpose library that is specialized for tree-contraction. The main difference between the specialized library and the general-purpose library is the handling of closures, time-stamps, and memoization. Since C++ does not support closures, we create our own closures using standard data structural techniques. Since tree-contraction is data-parallel, change-propagation does not require time-stamps, and therefore no time-stamps are needed. Since tree-contraction uses loops and does not need general power of recursion, a general memoization technique is not required.

17.1 Overview

Our approach is to map a forest of possibly unbalanced trees, called *primitive trees*, to a set of balanced trees, called *RC-Trees* (Rake-and-Compress Trees) by applying tree-contraction technique of Miller and Reif [62] on each primitive tree, and use change propagation to update the RC-Trees when edges are inserted/deleted dynamically into/from the primitive trees. To process applications-specific queries we annotate RC-Trees with application-specific data and use standard techniques to compute queries. To process application-specific data changes, we recompute the annotations of the RC-Trees that are affected by the change. Beyond the focus on change propagation for handling edge insertions/deletions, our approach have the following distinguishing properties:

1. We separate *structural* operations (links and cuts) from *application-specific* queries and data changes. The structure (*i.e.*, shape) of the data structure depends only on the structure of the underlying tree and is changed only by link and cut operations, applications-specific queries and data changes merely traverse the RC-Trees and change the annotations (tags).

   Previous data structures such as Link-Cut Trees [91, 92] and Top-Trees [11, 10] do allow the structure to be modified by application queries via the *evert* and the *expose* operations. For example, in Link-Cut trees, one node of the underlying tree is designated as root, and all path queries must be with respect to this root. When performing a path query that does not involve the root, one end point of the path query must be made root by using the evert operation. In top trees, the user operates on the data structure using a set of operations provided by the interface and is expected to access only the root node of the data structure. The user ensures that the root note contains the desired information by using the *expose* operation. The expose operation often restructures the top tree. For example a path query that is different than the previous path query requires an expose operation that restructures the top tree.

2. The data structure directly supports batch changes.

3. We present an experimental evaluation by considering a large number of applications including

   - path queries as supported by Link-Cut Trees [91, 92],
   - subtree queries as supported by Euler Tour Trees [46, 96],
   - non-local search: centers, medians, and least-common-ancestors as supported by Top Trees [11, 10],
   - finding the distance to a set of marked vertices from any source,
   - semi-dynamic (incremental) minimum spanning trees, and
   - max-flow algorithm of Sleator and Tarjan [91].

Our experiments confirm the theoretical time bounds presented in Chapter 10 and show that the approach can yield efficient dynamic algorithms even when compared to direct implementations. In particular, the experiments show that, when compared to an implementation of Link-Cut Trees [91, 92], RC-Trees are up to a factor of tree faster for path queries, and up to a factor of five slower for structural operations. This trade-off can make one data structure preferable to the other depending on the application. For example, an incremental minimum-spanning tree algorithm can perform many more queries than insertions and deletions, whereas a max-flow algorithm is likely to perform more structural operations. Note that RC-Trees directly support a broad range of queries, whereas Link-Cut Trees are specialized for path queries only.

A key property of self-adjusting computation is that it directly supports batch processing of changes without requiring any change to the implementation. In the context of dynamic-trees, our experiments show a large performance gap (asymptotically a logarithmic factor) between batch and one-by-one processing of changes as would be required in previously proposed data structures. *Batch changes* arise in the following two settings.

1. **Multiple, simultaneous changes to the input:** For example, the user of a dynamic minimum-spanning tree algorithm may choose to delete many edges at once. In this case, it would be more efficient to process all changes by a single pass on the data structure than to break them up into single changes.

2. **Propagation of a single change to multiple changes:** For example, the dynamic connectivity and the Minimum-Spanning Tree algorithms of Holm *et. al.* [49], maintain a hierarchy of dynamic trees. When a single edge is deleted from the input graph, a number of edges in one level of the hierarchy can be inserted into the next level. Since the dynamic-trees data structure for the next level is never queried until all insertions are performed, the insertions can be processed as a batch. In general, supporting batch changes is essential if dynamic data structures are to be composed, because a single change can propagate to multiple changes.

17.2 Rake-and-Compress Trees

This section describes a solution to the dynamic-trees problem based on change propagation on a static tree-contraction algorithm. The section summarizes the previous work [5] and gives more detail on the processing of application-specific queries.

We use Miller and Reif's tree-contraction algorithm [62] to map arbitrary, bounded-degree trees to a balanced trees. We call the trees being mapped, the *primitive trees*, and the balanced trees that they are mapped to, the *RC-Trees* (Rake-and-Compress Trees). RC-Trees represents a recursive clustering of primitive trees. Every node of an RC-Tree represents a cluster (subtree) of the corresponding primitive tree.

To apply RC-Trees to a particular problem, we annotate RC-Trees with information specific to that problem, and use tree traversals on the RC-Trees to compute properties of the primitive trees. Tree traversals on RC-Trees alone suffice to answer dynamic queries on static primitive trees. To handle dynamic changes to primitive trees, *i.e.*, edge insertions and deletions, we use change propagation. Given a set of changes to a primitive tree, the change-propagation algorithm updates the RC-Tree for that primitive tree by rebuilding the parts of the RC-Tree affected by the change.

17.2.1 Tree Contraction and RC-Trees

Given some tree T, the tree-contraction algorithm of Miller and Reif [62] contracts T to a single vertex by applying rake and compress operations over a number rounds. *Rake* operations delete all leaves in the tree and *compress* operations delete an independent set of vertices that lie on *chains*, *i.e.*, paths consisting of

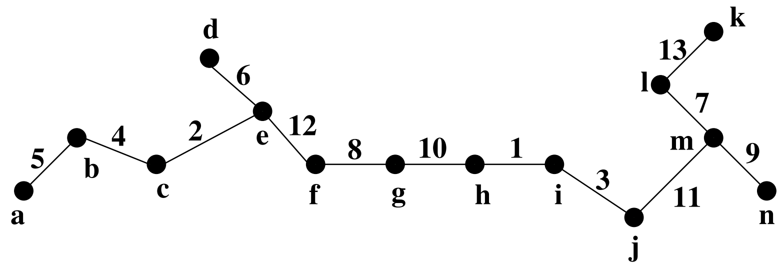


Figure 17.1: A weighted tree.

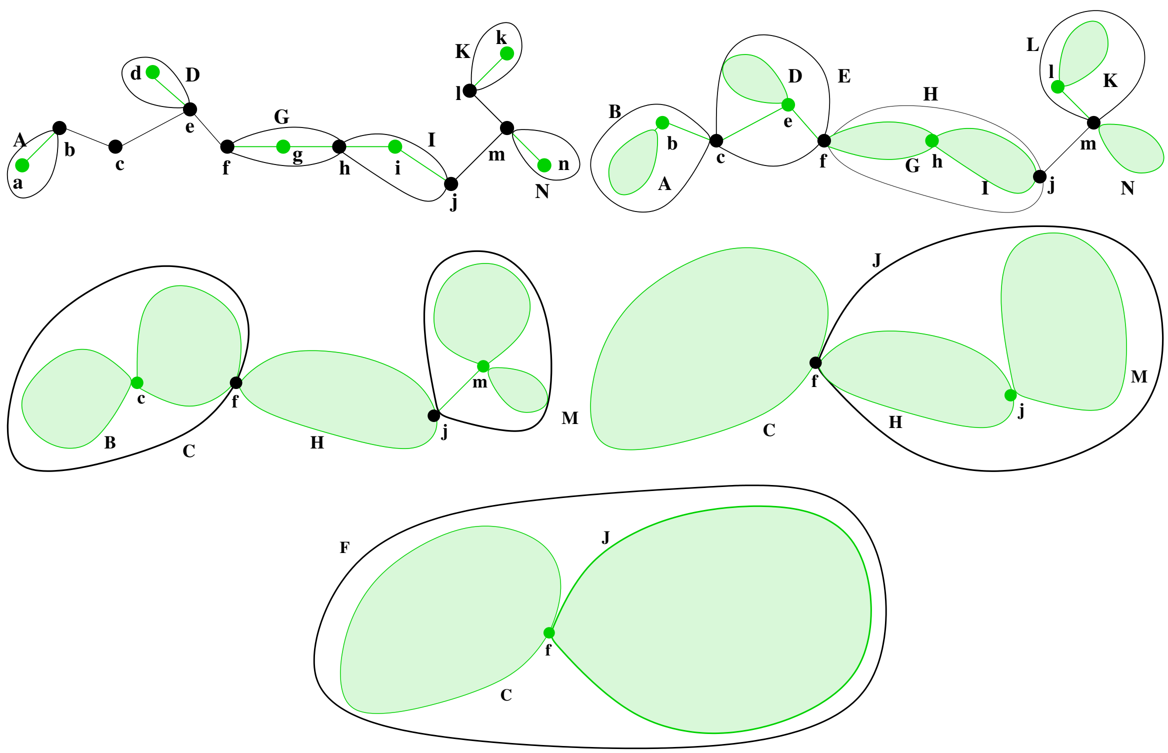


Figure 17.2: An example tree-contraction.

degree two vertices. When the tree is contracted into a single vertex, a special *finalize* operation is performed to finish tree contraction. When using the random-mate technique, tree contraction takes expected linear time, and requires logarithmic number of rounds (in the size of the input) in expectation [62].

The original tree-contraction algorithm of Miller and Reif was described for directed trees. In this paper, we work with undirected trees and use the generalization tree contraction for undirected trees [5]. For applications, we assume that the edges and vertices of trees can be assigned *weights* (for edges) and *labels* (for vertices) respectively. Weights and labels can be of any type.

As an example consider the weighted tree shown in Figure 17.1. Figure 17.2 shows the complete contraction of the tree (the rounds increase from top to bottom). Since tree contraction does not depend on the

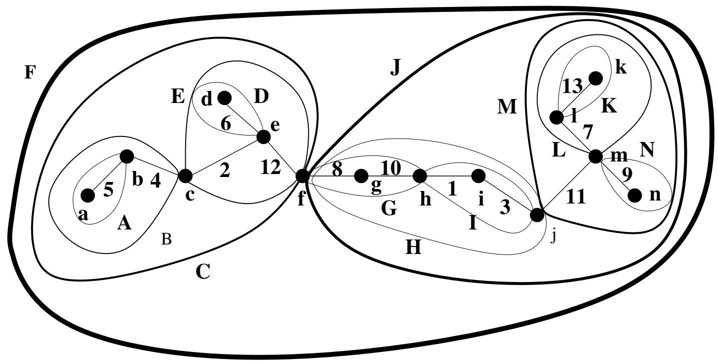


Figure 17.3: A clustering.

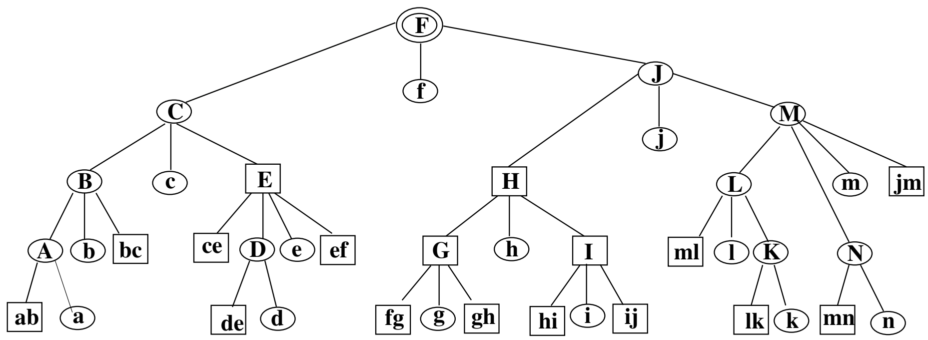


Figure 17.4: An RC-Tree.

weights they are omitted from Figure 17.2. Each round of a contraction deletes a set of vertices by using the rake and compress operations.

— A *rake* operation deletes a leaf and the edge adjacent to it, and stores the contribution of the deleted vertex and the edge in its neighbor. The *contribution* of a raked vertex is computed by calling the `rake_data` operation that is specified by the application. The `rake_data` operation computes the contribution from 1) the label of the deleted vertex, 2) contributions stored at that vertex, and 3) the weight of the deleted edge.

— A *compress* operation removes a degree two vertex, say v, and the edges adjacent to it, (u, v) and (v, w), and inserts an edge (u, w), and stores the contribution of v, (u, v), and (v, w) on u and w. The *contribution* of a compressed vertex is computed by calling the `compress_data` operation that is specified by the application. The `compress_data` operation computes the contributions from 1) the label of v, 2) the contributions stored at v, and 3) the weights of (u, v) and (v, w).

For example, in Figure 17.2, the first round rakes the vertices a, d, n, and k and compresses g, and i. At the end of the contraction, when the tree is reduced to a single vertex, a special *finalize* operation is performed to compute a value from the contributions stored at that vertex by calling the `finalize_data` operation that is specified by the application.

Tree contraction can be viewed as recursively clustering a tree into a single cluster. Initially the vertices and the edges form the *base clusters*. The rake, compress, and finalize operations form larger clusters by a number of smaller clusters and a base cluster consisting of a single vertex. In Figure 17.2, all clusters (except for the base clusters) are shown with petals. Each cluster is labeled with the capitalized label of the vertex joining into that cluster. For example, raking vertex a creates the cluster A consisting of a and b, and the edge (a, b); compressing the vertex g creates the cluster G, consisting of the vertex g and the edges (f, g) and (g, h). In the second round, raking the vertex b creates the cluster B that contains the cluster A and the edge (b, c). In the last round, finalizing f creates the cluster F that contains the clusters C and J. Figure 17.3 shows the *clustering* consisting of all the clusters created by the contraction shown in Figure 17.2.

We define a *cluster* as a subtree of the primitive tree induced by a set of vertices. For a cluster C, we say that vertex v of C is a *boundary vertex* if v is adjacent to a vertex that does not belong to C. The *boundary* of a cluster consists of the set of boundary vertices of that cluster. The *degree* of a cluster is the number vertices in its boundary. For example, in Figure 17.3, the cluster A has the boundary $\{b\}$, and therefore has degree one; the boundary of the cluster G is $\{f, g\}$ and therefore G has degree two. In tree contraction, all clusters except for the final cluster has degree one or degree two. We will therefore distinguish between *unary, binary, and final* clusters. It is a property of the tree contraction that

1. the rake operations yield unary clusters,
2. the compress operations yield binary clusters, and
3. the finalize operation yields the final cluster which has degree zero.

The contraction of a primitive tree can be represented by a tree, called *RC-Tree*, consisting of clusters. Figure 17.4 shows the RC-Tree for the example contraction shown in Figures 17.2 and 17.3. When drawing RC-Trees, we draw the unary clusters with circles, the binary clusters with squares, and the final cluster with two concentric circles. Since tree contraction takes expected logarithmic number of rounds, the RC-Tree of a primitive tree is probabilistically balanced. It is in this sense, that tree contraction maps unbalanced trees to balanced trees.

Throughout this paper, we use the term "node" with RC-Trees and the term "vertex" with the underlying primitive trees. We do not distinguish between a node and the corresponding cluster when the context makes it clear.

17.2.2 Static Trees and Dynamic Queries

RC-Trees can be used to answer dynamic queries on static trees. The idea is to annotate the RC-Trees with application-specific information and use tree traversals on the annotated trees to answer application-specific queries. To enable annotations, we require that the applications provide `rake_data`, `compress_data`, and `finalize_data` functions that describe how data (edge weights and/or vertex labels) is combined during rake, compress, and finalize operations respectively. Using these operations, the tree-contraction algorithm tags each cluster with the value computed by the `rake_data`, `compress_data`, and `finalize_data` operation computed during the operation that forms that cluster.

Once an RC-tree is annotated, it can be used to compute various properties of the corresponding primitive tree by using standard tree traversals. For all applications considered in this paper, a traversal involves a constant number of paths between the leaves and the root of an RC-Tree. Depending on the query, these paths can be traversed top down, or bottom-up, or both ways. Since RC-Trees are balanced with high probability, all such traversals require logarithmic time in expectation.

Section 17.3 describes some applications demonstrating how RC-Trees can be used to answer various queries such path queries, subtree queries, non-local search queries in expected logarithmic time.

17.2.3 Dynamic Trees and Dynamic Queries

To support dynamic trees, *i.e.*, edge insertion/deletions, we use change propagation. Change propagation effectively rebuilds the tree-contraction by rebuilding the clusters affected by the change. For *data changes*, we update the annotations in the tree by starting at the leaf of the RC-Tree specified by the change and propagate this change up the RC-Trees. By Theorem 67 and by the fact that RC-Trees have logarithmic height in expectation, we have the following theorem.

Theorem 88 (Dynamic Trees)
For a bounded-degree forest F of n vertices, an RC-Forest F_{RC} of F can be constructed and annotated in expected $O(n)$ time using tree-contraction and updated in expected $O(\log n)$ time under edge insertions/deletions, and application specific data changes. Each tree in the RC-Forest F_{RC} has height expected $O(\log n)$.

17.3 Applications

This section describes how to implement a number of applications using RC-Trees. Experimental results with these application are given in Section 17.5. Implementing an application using RC-Trees involves

1. providing the `rake_data`, `compress_data`, and `finalize_data` operations that specify how data is combined during rake, compress, and finalize operations respectively, and

2. implementing the queries on top of RC-Trees using standard tree traversals.

This section considers the following applications: path queries, subtree queries, diameter queries, center/median queries, least-common-ancestor queries, and nearest-marked-vertex queries. Of these queries, path queries, subtree queries, diameter queries, and nearest-marked-vertex queries all require simple tree traversal techniques that traverse a fixed number of bottom-up paths in the RC-Tree. Other queries, including centers/medians and least-common ancestor queries, require traversing a fixed number of bottom-up and top-down paths. Queries that require top-down traversal are sometimes called non-local search queries [11], because they cannot be computed by only using local information pertaining to each cluster.

Throughout this paper, we work with undirected trees. Applications that rely on directed trees specify an arbitrary root vertex with each query. For the discussion, we define the *cluster path* of a binary cluster as the path between its two boundary vertices.

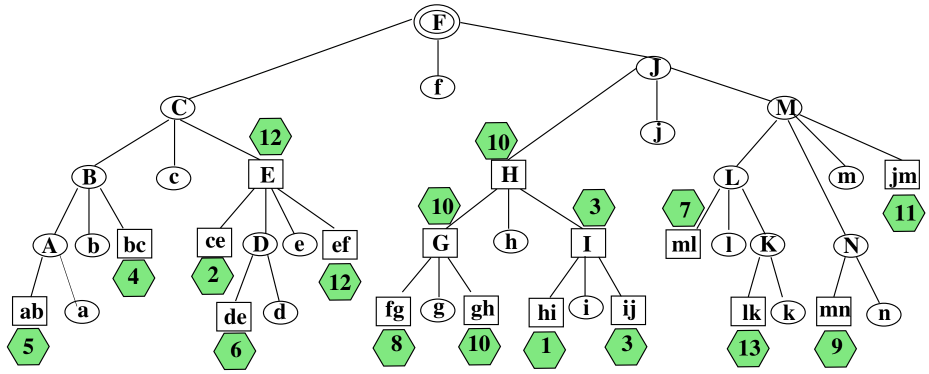


Figure 17.5: An RC-Tree with tags.

17.3.1 Path Queries

A path query [91, 92] asks for the heaviest edge on a path. Consider, as an example, the tree shown in Figure 17.1. The answer for the path query with vertices c and m is the edge (e, f) with weight 12. In general path queries can be defined on arbitrary associative operations on weights. The techniques described here apply in this general setting.

For this description, we assume that a path query asks for the weight of the heaviest edge on a given path—the heaviest edge itself can be computed by keeping track of edges in addition to weights. To answer path queries, we annotate each binary cluster with the weight of the heaviest edge on its cluster path. All other clusters (unary and final) will have no tags. These annotations require that the application programmer specify the functions `rake_data`, and `finalize_data` as "no-ops", and `compress_data` as the maximum operation on edge weights.

Figure 17.5 shows the RC-Tree for the contraction shown in Figure 17.3 where each binary cluster is annotated with the weight of the heaviest edge on its cluster path. For example, the cluster e is tagged with 12, because this is the largest weight on its cluster path consisting of the edges (c, e) and (e, f). The tag of each binary cluster is written in a hexagon. The RC-Tree is an annotated version of the tree in Figure 17.4.

By using the annotations, the maximum edge weight on the path between u and v can be found by simultaneously walking up the tree from u and v until they meet. For each cluster from u, we compute the maximum weight between u and the boundary vertices of that cluster. Similarly, for each cluster on the path from v, we compute the maximum weight between v and the boundary vertices of that cluster. Consider the cluster C that two paths meet and let C_u be the child of C on the path from u, and let C_v be the child of C on the path from v. The answer to the query is the maximum of the values computed for the common boundary vertex of C_u and C_v.

As an example, consider the primitive tree shown in Figure 17.1, its clustering Figure 17.3, the RC-Tree shown in Figure 17.5. Consider the path query with c and m. Consider the path from c to the root. The cluster C will be tagged with 12 because this is the heaviest weight between c and f (the boundary of C), and the cluster F will be tagged with 12. Consider now the path from m. The cluster M will be tagged with 11 and the cluster J will be tagged with 11. Since f is the first common boundary between the two paths explored, the result is 12, the maximum of the values associated with f, *i.e.*, 11 and 12.

17.3.2 Subtree Queries

A subtree query asks for the heaviest edge in a subtree. The subtree specified by a tree root r and a root v of the subtree. Consider, as an example, the tree shown in Figure 17.1. The answer for the path query with tree root c and subtree root m is the edge (l, k) with weight 13. In general subtree queries can be defined on arbitrary associative operations on weights. The technique described here applies in this general setting.

To answer subtree queries, we annotate each cluster with the weight of the heaviest edge in that cluster. These annotations require that the application programmer specifies the functions `rake_data`, `compress_data`, and `finalize_data` as the maximum operation.

By using the annotations, the maximum edge weight in a subtree specified by a tree root r and a subtree root v can be found by simultaneously walking up the tree from r and v until the paths meet. For the clusters on path from r, we compute no information. For each cluster on the path from v, we compute the heaviest edge in the subtree rooted at v with respect to each boundary vertex as the tree root. Consider the RC-Tree node C that two paths meet and let C_r be the child of C on the path from r, and let C_v be the child of C on the path from v. The answer to the query is the maximum of the tag for C_r and the value computed for C_v for the common boundary vertex of C_r and C_v.

This technique for finding the heaviest edge in a subtree can be used to compute other kinds of subtree queries, such as queries that can be computed by Euler-Tour Trees [96, 46], by replacing the maximum operator with the desired associative operator.

17.3.3 Diameter

The diameter of a tree is the length of the longest path in the tree. For example, the longest path in the tree shown in Figure 17.1 is the path between a and k; therefore the diameter is 80. To compute the diameter of a tree, we annotate

— each unary cluster with its diameter, and the length of the longest path originating at its boundary vertex,

— each binary cluster with the length of its cluster path, its diameter, and length of the longest path originating at each boundary vertex, and

— the final cluster with its diameter.

It is relatively straightforward to specify the `rake_data`, `compress_data`, and `finalize_data` operations to ensure that the clusters are annotated appropriately.

Since the root of the RC-Tree is tagged with the diameter of the tree, computing the diameter requires no further traversal techniques. Note that, since change-propagation updates the annotations of the RC-Trees, the diameter will be updated appropriately when edge insertions/deletions take place.

17.3.4 Distance to the Nearest Marked Vertex

This type of query asks for the distance from a given query vertex to a set of predetermined marked vertices and has applications to metric optimization problems [11]. As an example, consider the tree shown in

Figure 17.1 and suppose that the vertices a and n are marked. The answer to the nearest marked vertex query for vertex d is 17, because the length of the path between d and a is 17, whereas the length of the path between d and a is 60.

For this application, we annotate 1) each binary cluster with the length of its cluster path, and 2) each cluster with the distance between each boundary vertex and the nearest marked vertex in that cluster. The annotations are easily computed by specifying `rake_data`, `compress_data`, and `finalize_data` operations based on the minimum and addition operations on weights.

Given an annotated RC-Tree, we compute the distance from a query vertex v to the nearest marked vertex by walking up the RC-Tree from v as we compute the following values for each cluster on the path: 1) the minimum distance between v and the nearest marked vertex in that cluster, 2) the minimum distance between v and each boundary vertex. The answer to the query will be computed when the traversal reaches the root of the RC-Tree.

17.3.5 Centers and Medians

Centers and medians are non-local-search queries [11] and therefore require both a bottom-up and a top-down traversal of RC-Trees. They are quite similar, and therefore, we describe center queries only here.

For a tree T with non-negative edge weights, a center is defined as a vertex v that minimizes the maximum distance to other vertices in the tree. More precisely, let $d_T(v)$ be the maximum distance from vertex v to any other vertex in the tree T. A center of T is a vertex c such that $d_T(c) \leq d_T(v)$ for any vertex v.

To find the center of a tree, we annotate 1) each cluster C with $d_C(v)$ for each boundary vertex v of C, and 2) each binary cluster with the length of its cluster path. The annotations are easily computed by specifying `rake_data`, `compress_data`, and `finalize_data` operations based on the maximum and addition operations on weights.

Given an annotated RC-Tree, we locate a center by taking a path from the root down to the center based on the following observation. Consider two clusters C_1 and C_2 with a common boundary vertex v. Assume without loss of generality that $d_{C_1}(v) \geq d_{C_2}(v)$ and let u be a vertex of C_1 farthest from v. Then a center of $C_1 \cup C_2$ is in C_1, because any vertex in C_1 is no closer to u than v.

We find the center of the tree by starting at the root of the support tree and taking down a path to the center by visiting a cluster that contains a center next. To determine whether a cluster contains a center, we use the annotations to compute the distance from each boundary vertex to the rest of the tree and to the cluster and use the property of centers mentioned above.

17.3.6 Least Common Ancestors

Given a root r and vertices v and w, we find the least common ancestor of v and w with respect r by walking up the tree from all three vertices simultaneously until they all meet and then walking down two of the paths to find the least common ancestor.

Consider the cluster C that the paths from r, v, and w meet. If this is the first cluster that any two paths meet, then the vertex c joining into C is the least common ancestor. Otherwise, one of the children of C contains two clusters; move down to that cluster and follow the path down until the two paths split. If paths split at a binary cluster, proceed to the cluster pointing in the direction of the vertex whose path has joined last and follow the path down to the first unary cluster U; the vertex u joining into U is the least common ancestor. If the paths split at a unary cluster U, and both paths continue to unary clusters, then the vertex joining into U is the least common ancestor. Otherwise, continue to the binary cluster and follow the path down to the first unary cluster U; the vertex joining to U is the least common ancestor.

17.4 Implementation and the Experimental Setup

We implemented a library for dynamic trees based on RC-Trees and implemented a general purpose interface for dynamic trees. The code for the library is publicly available at the home page of the Aladdin Center `http://www.aladdin.cs.cmu.edu`.

We performed an experimental evaluation of this implementation by considering a semi-dynamic minimum-spanning-tree algorithm, the max-flow algorithm of Sleator and Tarjan [91], and some synthetic benchmarks. The synthetic benchmarks start with a single primitive tree and its RC-Tree and apply a sequence operations consisting of links/cuts, data changes, and queries. Proper forests arise as a result of edge deletions.

Since we rely on tree-contraction to map unbalanced trees to balanced tree, the primitive trees (the trees being mapped) must be bounded degree. Therefore, all our experiments involve bounded-degree trees. Since any tree can be represented as a bounded-degree tree, this restriction causes no loss of generality. In particular, RC-Trees can be applied to arbitrary trees by mapping an arbitrary tree T to a bounded-degree tree T' by splitting high-degree nodes into bounded-degree nodes [37], and by building an RC-Tree based on the bounded-degree tree T'. To support dynamic-tree operations, the mapping between T and T' must be maintained dynamically as the degree of the vertices in T change due to insertions/deletions by joining and splitting the vertices of T'. In certain applications, it is also possible to preprocess the input to ensure that bounded-degree dynamic-tree data structures can be used directly (without having to split and merge vertices dynamically). For example, the input network for a max-flow algorithm can be preprocessed by replacing high-degree vertices with a network of bounded-degree vertices that are connected via infinite capacity edges.

17.4.1 The Implementation

We implement a self-adjusting version of tree contraction algorithm of Miller and Reif [62]. The implementation represents each tree as a linked list of nodes ordered arbitrarily. Each node has an adjacency list of pointers to its neighbors in the tree. Since the library requires that each location be written at most once, the complete tree from each round is remembered. Data dependences are created by a special *read* operation that reads the contents of a vertex, performs a rake or compress operation on the vertex, and copies the contents to the next round if the vertex remains alive. Since the contracted tree from each round is stored in the memory the implementation uses expected $O(n)$ space.

For the implementation, we use a version of the general-purpose library that is specialized for tree contraction. Instead of general-purpose time stamps, the specialized library uses round numbers for time-stamping reads. The change-propagation algorithm remains correct without time-stamps because vertices can be processed in any order in each round (the tree-contraction algorithm is data-parallel). The specialized library also employs no general-purpose memoization scheme. This is because tree-contraction only relies on loops and not general recursion. Since each iteration of the main tree-contraction loop processes on vertex, the iterations can be created for each vertex affected by a change. The final difference between the specialized library and the general-purpose library concerns the creation of closures. Since the C++ language does not support closures, we create our own closures using standard data structural techniques.

Change propagation maintains a first-in-first-out queue of vertices affected by the changes and reruns the rake and compress operations of the original algorithm on the affected vertices until the queue becomes empty. The edge insertion and deletions initialize the change propagation queue by inserting the vertices involved in the insertion or deletion. During change propagation, additional vertices are inserted to the queue when a vertex that they read is written.

17.4.2 Generation of Input Forests

Our experiments with synthetic benchmarks take a single tree as input. To determine the effect that different trees might have on the running time, we generate trees based on the notion of *chain factor*. Given a chain factor f, $0 \leq f \leq 1$, we employ a tree-building algorithm that ensures that at least a fraction f of all vertices in a tree have degree two as long as $f \leq 1 - 2/n$, where n is the number of vertices in the tree. When the chain factor is zero, the algorithm generates random trees. When the chain factor is one, the algorithm generates a degenerate tree with only two leaves (all other vertices have degree two). In general, the trees become more unbalanced as the chain factor increases.

The tree-building algorithm takes as input the number of vertices n, the chain factor f and the bound d on the degree. The algorithm builds a tree in two phases. The first phase starts with an empty tree and grows a random tree with $r = \max(n - \lceil nf \rceil, 2)$ vertices by incrementally adding vertices to the current tree. To add a new vertex v to the current tree, the algorithm randomly chooses an existing vertex u with degree less than d, and inserts the edge (u, v). In the second phase, the algorithm adds the remaining $n - r$ vertices to the tree T obtained by the first phase. For the second phase, call each edge of the tree T a *super edge*. The second phase inserts the remaining vertices into T by creating $n - r$ new vertices and assigning each new vertex to a randomly chosen super edge. After all vertices are assigned, the algorithm splits each super edge (u, v) with assigned vertices $v_1, \ldots, v_l$ into $l + 1$ edges (u, v_1), (v_1, v_2), $\ldots (v_{l-1}, v_l)$, (v_l, v). Since all of the vertices added in the second phase of the construction have degree two, the algorithm ensures that at least a fraction f of all vertices have degree two, as long as $f \leq 1 - 2/n$.

Our experiments show that the performance of RC-Trees is relatively stable for primitive trees with varying chain factors, except for the degenerate tree with only two leaves. When measuring the effect of input size, we therefore generate trees with the same chain factor (but with different number of vertices). For these experiments, we fix the chain factor at 0.5. Since the data structure is stable for a wide range of chain factors, any other chain factor would work just as well.

For all our experiments that involve synthetic benchmarks, we use degree-four trees. We use degree-eight trees for the semi-dynamic minimum spanning trees, and the max-flow applications.

17.4.3 Generation of Operation Sequences

For the synthetic benchmarks, we generate a sequence of operations and report the time per operation averaged over 1024K operations. All operation sequences contain one of the following three types of operations

1. application-specific queries,

2. application-specific-data changes, and

3. edge insertions/deletions (link and cuts).

We generate all application-specific queries and data changes randomly. For queries, we randomly pick the vertices and edges involved in the query. For data changes, we pick random vertices and edges and change the labels (for vertices) and weights (for edges) to a randomly generated label or weight respectively.

For generating edge insertions/deletions, we use two different schemes. The first scheme, called *fixed-chain-factor scheme* ensures that the chain factor of the forest remains the same. Most experiments rely on this scheme for generating the operations. The second scheme, called *MST-scheme*, relies on minimum spanning trees, and is used for determining the performance for change propagation under batch changes.

— *Fixed-chain-factor scheme:* This scheme generates an alternating sequence of edge deletions and insertions by randomly selecting an edge e, deleting e, and inserting e back again, and repeating this process. Since the tree is restored after every pair of operations, this scheme ensures that the chain factor remains the same.

— *MST Scheme:* The MST scheme starts with an empty graph and its MST, and repeatedly inserts edges to the graph while maintaining its MST. To generate a sequence of operations, the scheme records every insertion into or deletion from the MST that arise as a result of insertions to the graph.

In particular, consider a graph G, and its minimum-spanning forest F, and insert an edge (u, v) into G to obtain G'. Let m be the heaviest edge m on the path from u to v. If the weight of m is less than the weight of e, then F is a minimum-spanning forest of G'. In this case, the forest remain unchanged and the scheme records nothing. If the weight of m is larger than that of e, then a minimum-spanning forest F' of G' is computed by deleting m from F and inserting e into F. In this case, the scheme records the deletion of m and insertion of e.

Since our implementation of RC-Trees assumes bounded-degree trees, we use a version of this scheme that is adapted to bounded-degree trees. If an inserted edge e causes an insertion that increases the degree of the forest beyond the predetermined constant, then that insertion is not performed and not recorded; the deletion, however, still takes place.

17.5 Experimental Results

We ran all our experiments on a machine with one Gigabytes of memory and an Intel Pentium-4, 2.4 GHz processor. The machine runs the Red Hat Linux 7.1 operating system.

17.5.1 Change Propagation

We measure the time for performing a change propagation with respect to a range of chain factors and a range of sizes. Figures 17.6 and 17.7 show the results for these experiments. To measure the cost of change propagation independent of application-specific data, this experiment performs no annotations—- `rake_data`, `compress_data`, and `finalize_data` operations are specified as “no-ops”.

Figure 17.6 shows the timings for change propagation with trees of size 16K, 64K, 256K, 1024K with varying chain factors. Each data point is the time for change propagation after one cut or one link operation averaged over 1024K operations. The operations are generated by the fixed-chain-factor scheme. As the timings show, the time for change propagation increases as the chain factor increases. This is expected because the primitive tree becomes more unbalanced as the chain factor increases, which then results in a larger RC-Tree. When the chain factor is one, the primitive (degenerate) tree has only two leaves. Trees with only two leaves constitute the worst-case input for change propagation, because this is when the depth of the corresponding RC-Trees are (probabilistically) large compared to a more balanced tree of the same size. As additional data points for $0.91, 0.92, \ldots, 0.99$, show the steepest increase in the timings occurs when the chain factor increases from 0.99 to 1.0. This experiment shows that change-propagation algorithm is stable for a wide range chain factors for both small and larger trees.

Figure 17.7 shows the timings for varying sizes of trees with chain factors $0, 0.25, 0.50, 0.75$, and 1.0. Each data point is the time for change propagation after one link or cut operation averaged over 1024K operations. The operations are generated by the fixed-chain-factor scheme. The experiments show that the time for change propagation increases with the size of the input, and that the difference between trees of differing chain factors are relatively small, except for the degenerate tree (chain factor of one). This experiments suggests that the time for change-propagation is logarithmic in the size of the input and proven by previous work [5].

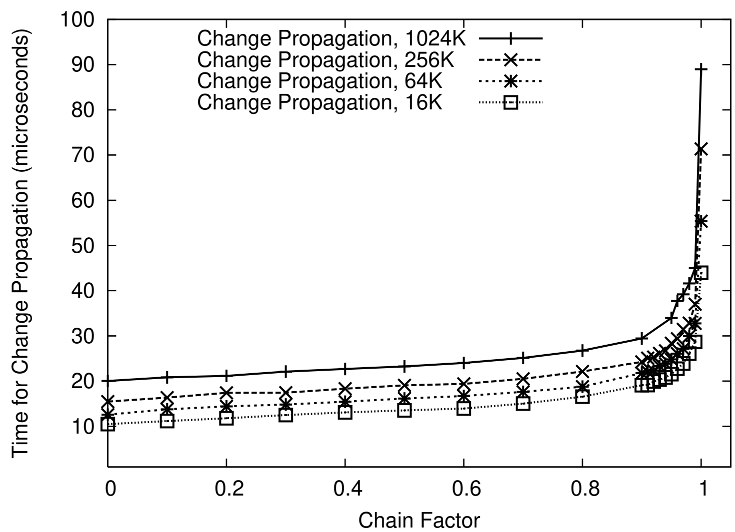


Figure 17.6: Change Propagation & chain factor.

17.5.2 Batch Change Propagation

An interesting property of change propagation is that it supports batch operations directly without any change to the implementation. Figure 17.8 shows the timings for change propagation with varying number cut and link operations in batch and in standard cascaded (one-by-one) fashion. The sequence of operations are generated by the MST scheme for a graph of 1024K vertices. The x axis (in logarithmic scale) is the

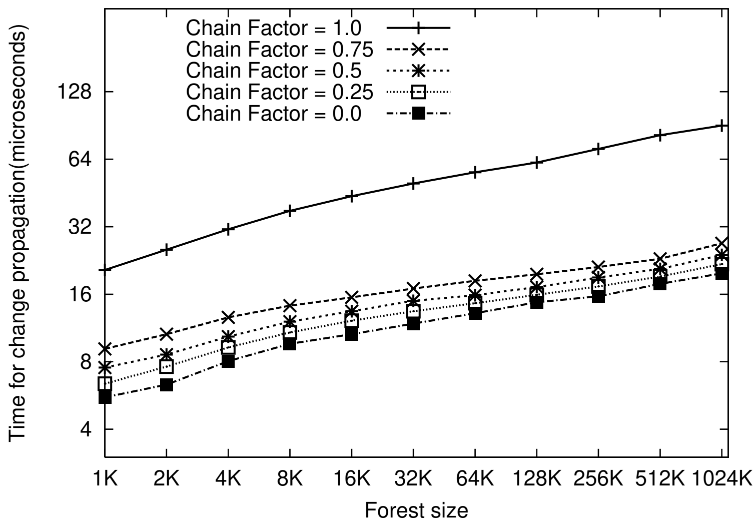


Figure 17.7: Change propagation & input size.

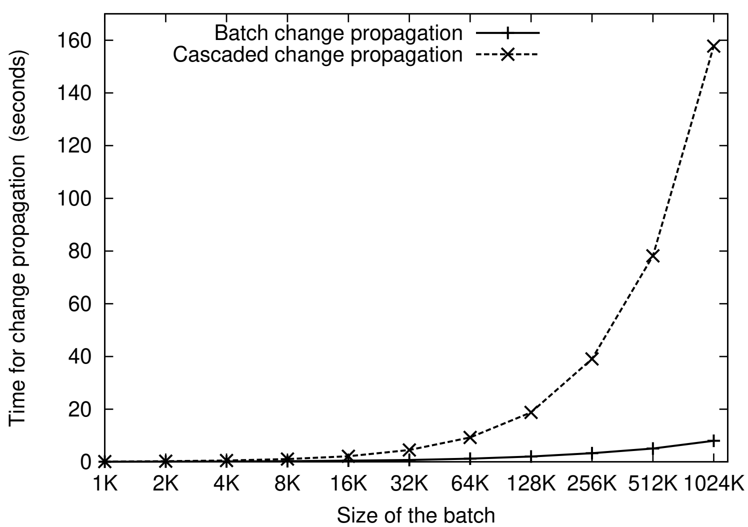


Figure 17.8: Batch change propagation.

size of each batch increasing from 1K to 1024K. Each data point represents the total time for processing the given number of cut and link operations in a batch or in a cascaded fashion.

The figure suggests a logarithmic factor asymptotic performance gap between processing changes in batch and in cascaded modes. This is expected, because, as the size of the batch approaches n, the change-propagation algorithm takes expected $O(n)$ time in the size of the input forest. Processing a x link or cut operations one-by-one, however, requires expected $O(x \log n)$ time.

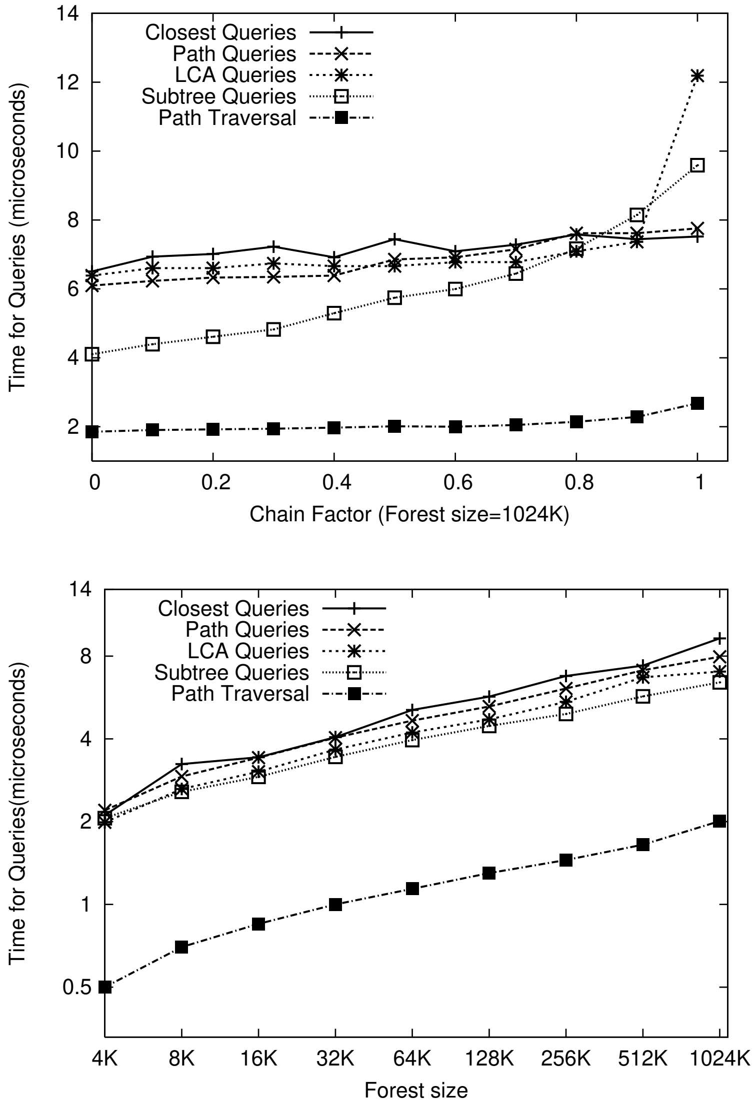


Figure 17.9: Queries versus chain factor and input size.

17.5.3 Application-Specific Queries

Figure 17.9 shows the time for various application-specific queries for varying chain factors and input sizes. The queries for diameters and medians are not included in the figure, because they are very fast—they simply read the value at the root of the RC-Tree.

In these experiments, each data point is the time for one query averaged over 1024K randomly generated queries. For comparison the data line "path traversal" shows the time for starting at a leaf of the support tree and walking up to the root of that tree without doing any operations. The path-traversal time is measured by randomly selecting leaves and averaging over 1024K leaves. Since most interesting queries will at least walk up the RC-Tree (a standard constant-degree tree), the path traversal time can be viewed as the best possible for a query. As the figure shows, the time for all queries are within a factor of five of the path-traversal time.

The time differences between different queries is a result of the specific computations performed by each query. The least common ancestor (LCA) queries, for example, traverse three different paths bottom up until they meet and one path down. Subtree queries traverse two paths bottom up, and also touch the clusters neighboring the paths.

The timings suggest that all queries are logarithmic time. This is expected because all queries take time proportional to the height of the underlying RC-Tree.

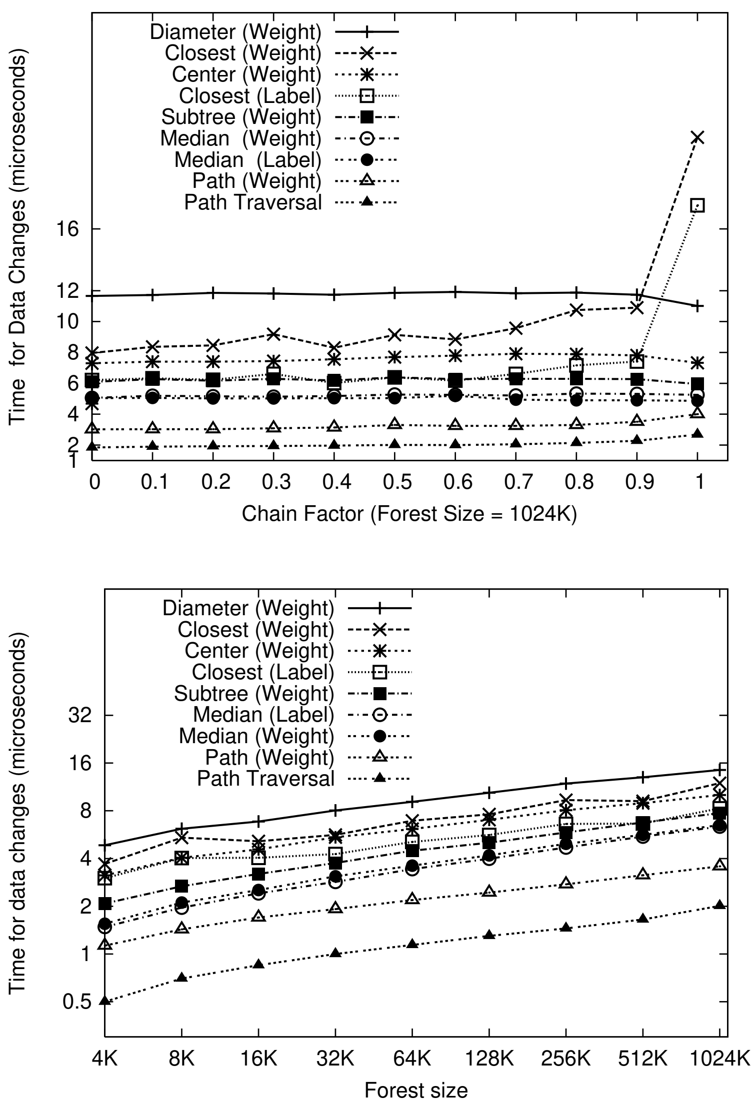


Figure 17.10: Weight changes vs chain factor & forest size.

17.5.4 Application-Specific Data Changes

We consider two types of application-specific data changes: label changes (for vertices), and weight changes (for edges). Processing a data change involves changing the weight or the label in the primitive tree and propagating the change up the RC-Tree.

Figure 17.10 shows the timing for edge weight and vertex label changes versus the chain factor and the size of the primitive forests. All our applications, except but for the Least-Common Ancestors, involve weighted trees. Weight changes are therefore relevant to all these applications. Vertex label changes however are only relevant to nearest-marked vertex, and the median applications. In the nearest-marked-vertex application, a label change can change an unmarked vertex to a marked vertex or vice versa. The mark indicates if that vertex is in the special set to which the distance of a given query node is measured. In the median application, a label change can change the contribution of a vertex to the weighted median.

As Figure 17.10 shows the time for data changes are relatively stable across a range of chain factors. For comparisons purposes, the figure also shows the time for path traversals. This measure is taken by randomly selecting leaves in the RC-Tree and measuring the time for traversing the path to the root (averaged over 1024K leaves). Since each data update traverses a path from a leaf to the root of the tree, the path-traversal time can be viewed as a lower bound for processing a data change. Figure 17.10 shows that the time for data changes is within an order of magnitude of the time for path traversals for all but the diameter application.

The timings suggest that all data changes take logarithmic time to process. This is expected, because RC-Trees have logarithmic height in expectation.

17.5.5 Comparison to Link-Cut Trees

We implemented the Link-Cut Tree interface of Sleator and Tarjan [91, 92] as an application. The interface supports link and cut operations, path queries, and the *addCost* operation for adding weights to paths. We compare our implementation to Renato Werneck's [98] implementation of Link-Cut Trees using splay trees.

Figure 17.11 shows the timing for link and cut operations. As the figure shows, Link-Cut Trees (written LC-Trees in figures) are up to a factor of five faster than the RC-Trees for links and cuts.

Figure 17.12 shows the timings for data changes. For operations that add a weight to a path, Link-Cut Trees are up to 50% faster than RC-Trees. For operations that change the weight of one edge, RC-Trees are up to 50% faster than Link-Cut Trees.

Figure 17.13 shows the timings for queries. For path queries that ask for the heaviest edge on a path, RC-Trees are up to 60% faster that Link-Cut Trees. For comparisons purposes, Figure 17.13 also shows the timing for path queries, for a version of our interface that does not support adding weights on paths. These queries are up to a factor of three faster than Link-Cut Trees. Certain applications of path queries, such as Minimum-Spanning Trees, do not require adding weights on paths.

These comparisons show an interesting trade-off. Whereas RC-Trees tend to perform better in application-specific queries and data changes, LC-Trees perform better for structural changes (link and cut operations). As the experiments with minimum-spanning trees, and max-flow algorithms show, this trade-off between structural and data changes makes one data structure preferable to the other depending on the ratio of link-cut operations to queries. Note also that LC-Trees support path queries only and must be extended internally to support other type of queries such as subtree queries [23, 84] that RC-Trees support.

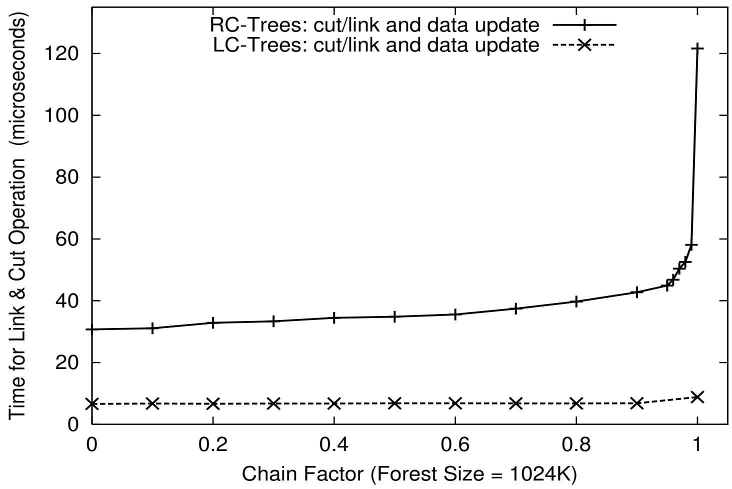


Figure 17.11: Link and cut operations.

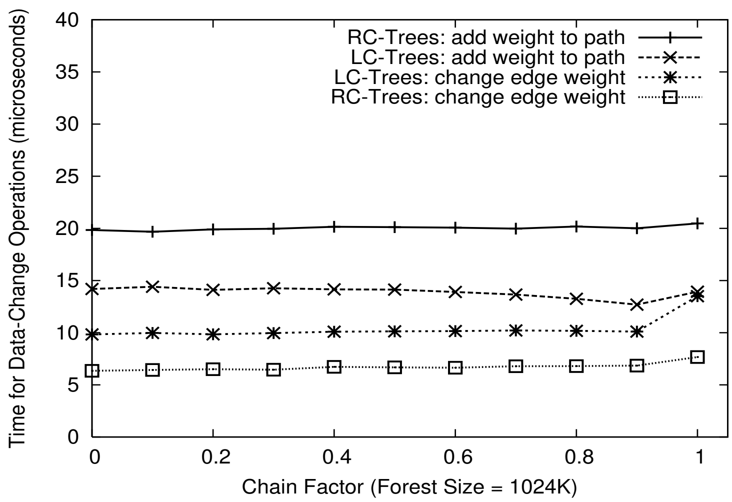


Figure 17.12: Data Changes.

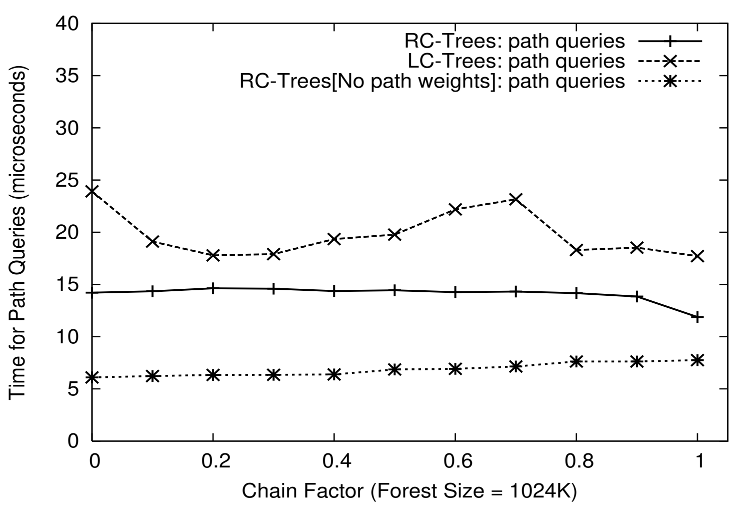


Figure 17.13: Path queries.

17.5.6 Semi-Dynamic Minimum Spanning Trees

We compare the performance of the implementations of a semi-dynamic minimum spanning tree algorithm using RC-Trees and Link-Cut Trees. The semi-dynamic algorithm maintains the minimum-spanning tree of a graph under edge insertions to the graph. When an edge e is inserted, the algorithm finds the maximum edge m on the path between the two end-points of e via a path query. If the weight of e is larger than that of m, the algorithm replaces m with e in the MST. If the weight of e is larger than that of m, then the MST remains the same. To find the maximum edge on a path quickly, the algorithm uses a dynamic-tree data structure to represent the MST; this enables performing path queries in logarithmic time. An edge replacement requires one cut and one link operation on the dynamic-trees data structure.

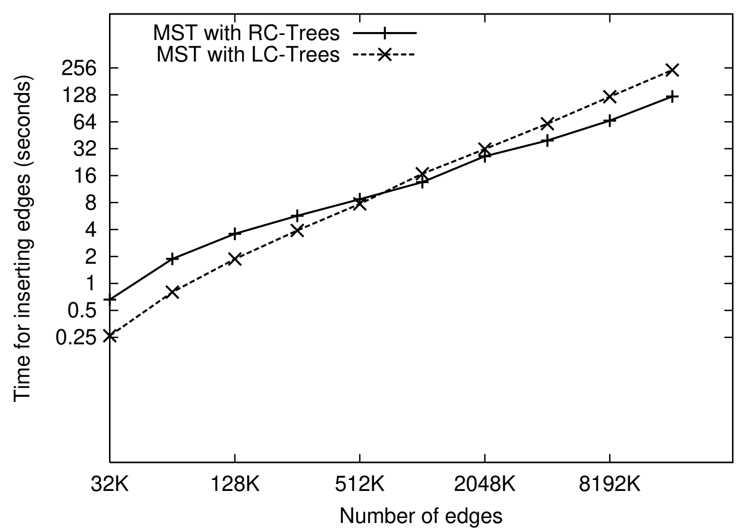


Figure 17.14: Semi-dynamic MST.

For this experiments, we compare two implementations of the semi-dynamic MST algorithm that only differ in their choice of the dynamic-trees data structure. We start with a graph of 32K vertices and no edges and randomly insert edges to the graph while updating its MST. We generate the sequence of insertions offline by randomly selecting a new edge to insert, and by randomly generating a weight for that edge. Since our implementation of RC-Trees supports constant-degree trees only, we generate a sequence of insertions that ensures that the underlying MST has constant degree. For this experiment, we used degree-eight RC-trees.

Figure 17.14 shows the timings using RC-Trees and Link-Cut Trees. Since the edge weights are randomly generated, insertions are less likely to cause replacements as the graph becomes denser. Therefore the number of queries relative to the number of link and cut operations increase with the number inserted edges. Since path queries with RC-Trees are faster than with Link-Cut Trees, the semi-dynamic MST algorithm performs better with RC-Trees for dense graphs.

17.5.7 Max-Flow Algorithms

As another application, we implemented the max-flow algorithm of Sleator and Tarjan [91]. The algorithm uses dynamic trees to find blocking flows. For the experiments, we used the DIMACS's Washington Generator in *fct 2* mode to generate *random level graphs* with 5 rows and $n/5$ columns, with capacity up to 1024K. The sum of the in-degree and the out-degree of any vertex in a random level graphs generated with these parameter is bounded by 8. We therefore use degree-eight RC-Trees for this experiment.

A random level graphs with r rows and l levels consists of $rl + 2$ vertices. The vertices consists of a source, a sink, and r rows. Each row consists of l vertices, one vertex for each level. A vertex at level $1 \leq i < l$ is connected to three randomly chosen vertices at level $i + 1$ by an edge whose capacity is determined randomly. In addition, the source u, is connected to each vertex at the first level by an edge, and each vertex at level l is connected to the sink v by an edge. The capacities of the edges incident to the source and sink are large enough to accommodate any flow.

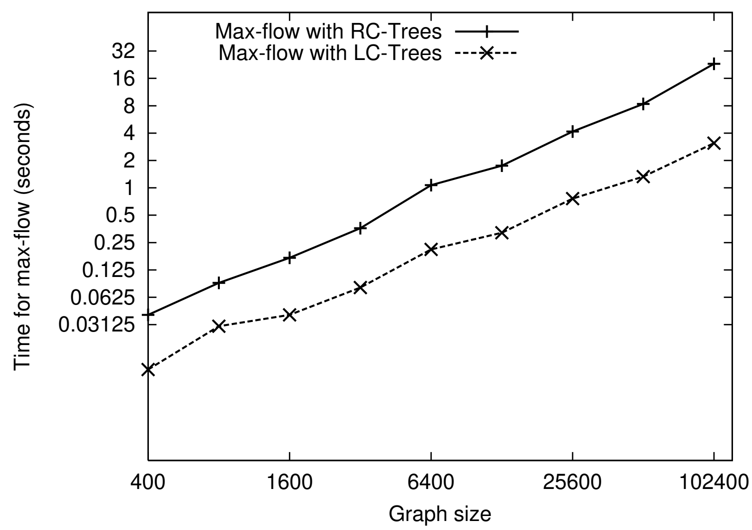


Figure 17.15: Max-flow

As Figure 17.15 shows the timings for two implementation of the max-flow algorithm of Sleator and Tarjan that differ only in their choice of the dynamic-tree data structure being used. As the timings show the LC-Trees implementation is faster than the RC-Trees implementation. This is expected because the algorithm performs more structural changes (link and cuts) than queries.

Chapter 18

Conclusion

This thesis presents techniques for devising, analyzing, and implementing programs that adjust to external changes automatically. The key properties of the techniques are

- generality,
- simplicity,
- analyzability, and
- efficiency.

The techniques are general-purpose; using our techniques any program can be made self-adjusting. Self-adjusting programs adjust to any change in their data, including multiple simultaneous (batch) changes, and changes to the outcomes of comparisons. The key to achieving generality is our dependence tracking techniques based on dynamic dependence graphs and memoization and a change-propagation algorithm for adjusting computations to external changes.

A key property of dependence tracking techniques and change propagation is that they can be applied automatically. In particular, as a program executes, a run-time system can track the dependences in the computation, and adjust the computation to changes automatically.

Self-adjusting programs are analyzable, *i.e.*, the asymptotic complexity of self-adjusting programs can be determined via analytical techniques. We prove a general theorem for determining the performance of change propagation. For a particular class of computations, we introduce an analysis technique, called trace stability, that helps bound the time for change propagation in terms of the distance between the execution traces of a program. Informally speaking, we show that if a program has similar or stable traces on similar inputs, then it will adjust to small changes efficiently.

Our techniques are asymptotically efficient. We prove that the overhead of our dependence tracking techniques is $O(1)$. We show that change propagation is effective by considering a number of applications. For these applications, we show that change propagation yields computations that adjust to changes within an expected constant factor of the best known bounds. It is important to note that we compare our bounds to bounds that are obtained by special-purpose (dynamic) algorithms.

We show that our techniques are practical by developing language techniques that enable applying dependence tracking selectively. The language techniques enable transforming an ordinary (non-self-adjusting) program into a self-adjusting program by making small, methodical changes to the code. By giving a semantics to change propagation, we prove that the self-adjusting programs adjusts to changes correctly. We presents two implementations of our techniques, one as a general-purpose ML library, and another as a C++ library specialized for certain data-parallel applications. We implement a number of applications using our libraries and present a thorough experimental evaluation. Our experiments show that the overhead of our dependence techniques is small, and that self-adjusting programs can perform well even when compared to special-purpose algorithms, both in theory and in practice.

18.1 Future Work

We discuss several directions for future research.

Multiple Writes

In this thesis, we focus on single-assignment and purely functional programs, where every memory location is written at most once. The key reason for this restriction is that single-assignment and purely functional programs are automatically persistent [31]. Persistence enables tracking the dependences in a computation and adjusting the computation to changes.

We show, in Chapter 14, that our techniques can be generalized to arbitrary, multiple-write programs by making computations persistent automatically. To achieve persistence, we rely on versioning techniques proposed by Driscoll *et. al.* [31]. We present a language with multiple writes (side effects), and give dependence tracking techniques and a change-propagation algorithm for this language. This shows that there are no fundamental difficulties in extending our techniques to languages that allow multiple writes or side effects. We plan to extend our algorithmic techniques to support multiple writes and study their effectiveness.

Compilers

The language techniques presented in this thesis are implemented as libraries for the ML language. These libraries help us gain insight into the practical effectiveness of our techniques. They are, however, unsatisfactory because of their weak typing guarantees and because of their lack of support for optimizations. Support for types is important, because our language techniques rely on sophisticated type systems to ensure both efficiency (by selective dependence tracking) and correctness (by ensuring that all appropriate dependences are tracked). Support for optimizations is important, because our experiments indicate that even very simple optimization, such as tail call optimizations, can reduce the overhead of dependence tracking significantly (by a factor of two or more). We plan to build language-extension, or a compiler, for writing self-adjusting programs.

Kinetic Data Structures

As described in Chapter 16, our techniques make obtaining kinetic versions of self-adjusting programs trivial. Although we present some experimental evidence that shows that our techniques are effective, our experience with kinetic data structures thus far is relatively limited. In the future, we plan to extend the trace-stability technique to enable determining the complexity of change propagation for kinetic computations, and show stability bounds for some applications. We have some early results and experiments that show that, as with dynamic algorithms, our approach yields efficient kinetic computations both in theory and in practice.

Garbage collection

Most garbage collection algorithms rely on the assumption that most objects have a short lifetime. This assumption does not hold for self-adjusting computations, because self-adjusting computations are persistent—much of the computation data has a long lifespan. An interesting avenue of research is to devise garbage collection techniques for self-adjusting or more generally for persistent programs. Self-adjusting computation has a particular structure that lends itself well to garbage collection, because the semantics makes clear what becomes garbage when.

Applications

Self-adjusting computation techniques can be used to build computing systems that adapt to and interact with continuously changing environments. Darema's proposal [26, 27] for creating "dynamic data driven application systems" and the references thereof highlight the importance of and the need for such systems. We plan to apply our techniques to specific application domains such as robotics, simulation systems, scientific computing, real-time systems, *etc.*

Appendix A

The Implementation

We briefly describe our implementation of the general-purpose library presented in Chapter 15 and give the code for the library. The library consists of the following modules: boxes, combinators, memo tables, meta operations, modifiables, order-maintenance data structure, and priority queues. We give the complete code for all modules except for priority queues and memo tables, which are standard data structures.

A.1 Boxes

The code for the box module is shown below.

```
signature BOXED_VALUE =
sig
  type index
  type 'a t

  val init: unit -> unit
  val new: 'a -> 'a t
  val eq: 'a t * 'a t -> bool
  val fromInt: int -> int t
  val fromOption: 'a t option -> 'a t option t
  val valueOf: 'a t -> 'a
  val indexOf: 'a t -> index
end

structure Box : BOXED_VALUE =
struct
  type index = int
  type 'a box = int * 'a
  type 'a t = 'a box

  val next = ref 0

  fun init () = next := ~1
  fun new v = (next := (!next) + 1; (!next,v))
  fun fromInt i = (i,i)
  fun  fromOption ob =
    case ob of
      NONE => (~1,ob)
    | SOME (i,_) => (i,ob)
  fun eq (a as (ka,_),b as (kb,_)) = (ka=kb)
  fun valueOf (_,v) = v
  fun indexOf (k,_) = k
end
```

A.2 Combinators

The code for the combinator module is shown below.

```
signature COMBINATORS =
sig
  type 'a modref   = 'a Modref.t
  type 'a cc

  val modref : 'a cc -> 'a modref
  val new : 'a -> 'a modref
  val write : 'a Box.t -> 'a Box.t cc
  val write' : ('a * 'a -> bool) -> 'a -> 'a cc
  val read : 'b Modref.t * ('b -> 'a cc) -> 'a cc
  val memoize: 'a  MemoTable.t -> int list -> (unit -> 'a) -> 'a
  val mkLift : ('b *'b -> bool) -> (int list * 'b) -> ('b Modref.t -> 'd) ->  'd
  val mkLiftCC : (('b * 'b -> bool) * ('d * 'd -> bool))  ->
                 int list * 'b -> ('b Modref.t -> 'd cc) -> 'd cc
end
```

```
structure Comb :> COMBINATORS =
struct

  type 'a modref = 'a Modref.t
  type 'a cc = 'a modref -> Modref.changeable
  type ('b, 'g) liftpad = ('b -> 'g) MemoTable.t * 'g MemoTable.t
  type ('b, 'g) liftpadCC = ('b -> 'g cc) MemoTable.t * 'g Modref.t MemoTable.t

  fun write x d = Modref.write d x
  fun write' eq x d = Modref.write' eq d x
  fun read (r, recv) d = Modref.read r (fn x => recv x d)
  val modref = Modref.modref
  val new = Modref.new
  fun memoize (pad:'a  MemoTable.t) key (f: unit -> 'a) =
    let
      fun run_memoized (f,r) =
        let val t1 = !(Modref.now)
            val v = f()
            val t2 = !(Modref.now)
            val nt1o = TimeStamps.getNext t1
            val _ =
              case nt1o of
                NONE => r:=SOME(v,NONE)
              | SOME(nt1) =>
                  if (TimeStamps.compare (nt1,t2)=LESS) then
                    (r := SOME(v,SOME(nt1,t2)))
                  else
                    (r := SOME(v,NONE))
        in v end
      fun  reuse_result (t1,t2) =
        let
          val _ = TimeStamps.spliceOut (!(Modref.now),t1)
          val _ = Modref.propagateUntil t2
        in ()end
      fun memoize' (r:'a MemoTable.entry)  =
        case !r of
          NONE => run_memoized (f,r)
        | SOME(v,to) =>
            case to of
              NONE => v
            | SOME(t1,t2) => (reuse_result (t1,t2); v)
      in
        memoize' (MemoTable.find(pad,key,!Modref.now))
      end

  fun lift (p1,p2) eqb  (key,b) f =
    let fun f' () = let val r = Modref.empty ()
                    in fn b => let val _ = Meta.change' eqb r b
                               in memoize p2 key (fn _ => f (r)) end

                   end
    in memoize p1 key f' b end

  fun mkLift eqb =  lift (MemoTable.new (), MemoTable.new ()) eqb
  fun mkLiftCC (eqb,eqd) =
    let fun lifted arg f =
          let fun f' (b) = let val r = modref (f b) in read (r, write' eqd) end
          in lift (MemoTable.new (), MemoTable.new ()) eqb arg f' end
    in
      lifted
    end
end
```

A.3 Memo Tables

The interface for the memo tables is shown below. As described in Section 6.1, it is easy to give an implementation for memo tables based on hash tables. We therefore omit the code for the implementation of memo tables here.

```
signature MEMO_TABLE =
sig
  type 'a memotable
  type 'a t = 'a memotable
  type 'a entry = ('a * ((TimeStamps.t * TimeStamps.t) option)) option ref

  val new: unit -> 'a memotable
  val find: 'a memotable * int list * TimeStamps.t -> 'a entry
end

structure MemoTable: MEMO_TABLE = struct ... end
```

A.4 Meta Operations

The code for the meta operations is shown below.

```
signature META_OPS =
sig
   type 'a modref = 'a Modref.t

   val init: unit -> unit
   val change: 'a Box.t modref -> 'a Box.t -> unit
   val change': ('a * 'a -> bool) -> 'a modref  -> 'a -> unit
   val deref : 'a modref -> 'a
   val propagate : unit -> unit
end

structure Meta :> META_OPS =
struct
  type 'a modref = 'a Modref.t

  val change  = Modref.change
  val change' = Modref.change'
  fun init () = (Box.init (); Modref.init ())
  val deref = Modref.deref
  val propagate = Modref.propagate
end
```

A.5 Modifiables

The code for the modifaibles is shown below.

```
signature MODIFIABLE =
sig
  type 'a modref
  type 'a t = 'a modref
  type changeable
  type time = TimeStamps.t

  val init : unit -> unit

  val empty: unit -> 'a modref
  val new: 'a -> 'a modref
  val modref: ('a modref -> changeable) -> 'a modref
  val read : 'a modref  -> ('a -> changeable) -> changeable
  val write : 'a Box.t modref -> 'a Box.t -> changeable
  val write' : ('a * 'a -> bool) -> 'a modref -> 'a -> changeable

  val change: 'a Box.t modref -> 'a Box.t -> unit
  val change': ('a * 'a -> bool) -> 'a modref -> 'a  -> unit
  val deref : 'a modref -> 'a
  val propagate : unit -> unit
  val propagateUntil :  time -> unit

  val now : time ref
  val finger: time ref
  val isOutOfFrame: time*time -> bool
end

structure Modref : MODIFIABLE=
struct
  type time = TimeStamps.t
  type changeable = unit

  exception UnsetMod

 (**********************************************************************
  ** Time Stamps
  **********************************************************************)
  val now = ref (TimeStamps.add (TimeStamps.init ()))
  val frameStart = ref (!now)
  val finger = ref (!now)

  fun insertTime () =
    let val t = TimeStamps.add (!now)
    in now := t; t
    end

 (**********************************************************************
  ** Priority Queue
  **********************************************************************)
  structure Closure =
  struct
    type t = ((unit -> unit) * time * time)
    fun compare (a as (ca,sa,ea), b as (cb,sb,eb)) = TimeStamps.compare(sa,sb)
    fun isValid (c,s,e) = not (TimeStamps.isSplicedOut s)
  end
  structure PQueue = PriorityQueue (structure Element=Closure)
```

```
type pq = PQueue.t
val PQ = ref PQueue.empty
fun initQ() = PQ := PQueue.empty
fun insertQ e =  PQ := PQueue.insert (e,!PQ)
fun findMinQ () =
  let val (m,q) = PQueue.findMin (!PQ)
      val _ = PQ := q
  in m end
fun deleteMinQ () =
  let val (m,q) = PQueue.deleteMin (!PQ)
      val _ = PQ := q
  in m end

(*********************************************************************
** Modifiables
*********************************************************************)
type 'a reader = ('a -> unit) * time * time
datatype 'a readers = NIL  | FUN of ('a reader * 'a readers)
datatype 'a modval = EMPTY | WRITE of 'a * 'a readers
type 'a modref = 'a modval ref
type 'a t = 'a modref

fun empty () = ref EMPTY
fun new v = ref (WRITE (v,NIL))
fun modref f =
  let val r = (ref EMPTY)
  in (f (r); r)  end
fun read modr f =
  case !modr of
    EMPTY => raise UnsetMod
  | WRITE (v,_) =>
    let
      val t1 = insertTime()
      val _ = f(v)
      val t2 = insertTime ()
      val WRITE (v,rs) = !modr
      val rs' = FUN((f,t1,t2),rs)
    in
      modr := WRITE(v,rs')
    end
fun readAtTime(modr,r as (f,_,_)) =
  case !modr of
    EMPTY => raise UnsetMod
  | WRITE(v,rs) => (modr := WRITE(v,FUN (r,rs)); f v)
fun addReadersToQ (rs: 'a readers, modr : 'a modref) =
  let
    fun addReader (r as (f,t1,t2)) =
      if TimeStamps.isSplicedOut(t1) then ()
      else insertQ(fn () => readAtTime(modr,r),t1,t2)
    fun addReaderList rlist =
      case rlist of
        NIL => ()
      | FUN(r,rest) => (addReader (r); addReaderList rest)
  in addReaderList rs
  end
fun write' comp modr v =
  case !modr of
    EMPTY => modr := WRITE (v,NIL)
  | WRITE(v',rs) =>
      if comp (v,v') then ()
      else
        let val _ = modr := WRITE(v,NIL)
```

```
          in addReadersToQ (rs,modr)
          end
  fun write modr v = write' Box.eq modr v

  fun deref modr =
    case !modr of
      EMPTY => raise UnsetMod
    | WRITE (v,_) => v

 (*******************************************************************
  ** Change propagation
  *******************************************************************)
  fun propagateUntil (endTime) =
    let fun loop  () =
          case  (findMinQ ()) of
            NONE => ()
          | SOME(f,start,stop) =>
            if (TimeStamps.isSplicedOut start) then loop ()
            else if (TimeStamps.compare(endTime,stop) = LESS) then ()
            else let val _ = deleteMinQ ()
                     val finger' = (!finger)
                     val _ = now := start
                     val _ = finger := stop
                     val _ = f()
                     val _ = finger := finger'
                     val _ = TimeStamps.spliceOut (!now,stop) handle e => raise e
                 in loop ()
                 end
    in (loop (); now := endTime)
    end

  fun propagate () =
    let fun loop  () =
          case  (findMinQ ()) of
            NONE => ()
          | SOME(f,start,stop) =>
              let val _ = deleteMinQ ()
                  val finger' = (!finger)
                  val _ = now := start
                  val _ = finger := stop
                  val _ = f()
                  val _ = finger := finger'
                  val _ = TimeStamps.spliceOut (!now,stop) handle e => raise e
              in loop ()
              end
    in loop ()
    end

  fun isOutOfFrame(start,stop) =
        not (TimeStamps.compare(!now,start) = LESS andalso
             TimeStamps.compare(stop,!finger) = LESS)

  fun init () = (now := TimeStamps.init();  initQ())
  fun change l v = write l v
  fun change' comp l v = write' comp l v
  fun change'' (l: 'a modref) v = write' (fn _ => false) l v
end
```

A.6 Order Maintenance

The interface and the implementation for the order-maintenance data structure is shown below. This implementation follows the description of Bender *et. al.* [16].

```
signature ORDERED_LIST =
sig
  eqtype t

  val init : unit -> t
  val new: unit -> t
  val add : t -> t
  val getNext : t -> t option
  val spliceOut: t*t -> int
  val isSplicedOut: t -> bool
  val compare: t*t -> order
end

structure TimeStamps : ORDERED_LIST =
struct
  open Word

  exception outOfTimeStamps
  exception AssertionFail
  exception badNode
  exception badTop

  val w_zero = fromInt(0)
  val w_one = fromInt(1)
  val w_two = fromInt(2)

(****************************************************************)
(**    TOP LEVEL                                              **)
(****************************************************************)

  datatype 'a TopLevelNode =
      NULL
    | TN of (('a TopLevelNode ref) (* prev *) *
             (word ref)            (* id   *) *
             ('a ref)              (* data *) *
             ('a TopLevelNode ref) (* next *) )

(* Constant for internal use.
   Smaller makes it more expensive but gives more stamps *)
  val T = 1.41421

  fun wordToReal w = Real.fromLargeInt((toLargeInt w))
  fun TNTopID NULL = raise badTop
    | TNTopID (TN(_,ref id,_,_)) = id
  fun TNTopIDRef NULL = raise badTop
    | TNTopIDRef (TN(_,id,_,_)) = id
  fun TNNextRef NULL = raise badTop
    | TNNextRef (TN(_,_,_,nxref)) = nxref
  fun TNPrevRef NULL = raise badTop
    | TNPrevRef (TN(pvref,_,_,_)) = pvref
  fun TNData NULL = raise badTop
    | TNData (TN(_,_,ref data,_)) = data

  local
    val maxVal = fromInt(~1)
    fun findMargin(mask,tau:real,lo,hi,bt,n_sofar) =
```

```
      let
        val lo_num = andb ((notb mask), bt)
        val hi_num = orb (lo_num,mask)
        fun extendLeft cur c =
            let val TN(ref pv,ref myid,_,_) = cur
                val prevID = TNTopID(pv)
            in if ((prevID>=lo_num) andalso (prevID<=myid)) then
                  extendLeft pv (c+w_one)
                else (cur,c)
            end
        fun extendRight cur c =
            let val TN(_,ref myid,_,ref nx) = cur
                val nxID = TNTopID(nx)
            in if ((nxID<=hi_num) andalso (nxID>=myid)) then
                 extendRight nx (c+w_one)
               else (cur,c)
            end
        val (lo', temp_count) = extendLeft lo n_sofar
        val (hi',count_total) = extendRight hi temp_count
        val gap_size' = mask + w_one
        val density = (wordToReal(count_total))/(wordToReal(gap_size'))
      in
        if (Real.<(density, tau)) then
           (lo',hi',count_total,gap_size',lo_num,hi_num)
        else let val newmask = mask*w_two + w_one
             in if (newmask = maxVal) then raise outOfTimeStamps
                else findMargin(newmask, tau/T, lo', hi', bt, count_total)
             end
      end
  fun scanUpdate (cur,last,num,incr) =
    ((TNTopIDRef(cur)):= num;
      if (cur=last) then ()
      else scanUpdate (!(TNNextRef(cur)), last, num+incr, incr))
  fun issueTopLevelNumber p =
    let val left = TNTopID(!(TNPrevRef(p)))
        val t' = TNTopID(!(TNNextRef(p)))
        val right = if (t'<=left) then left+w_two else t'
    in
      if ((left+w_one) = right) then
        let val (lo,hi,total,gap_size,start_tag,_) =
                               findMargin(w_one,1.0/T,p,p,TNTopID(p), w_one)
            val incr = gap_size div total
        in scanUpdate (lo,hi,start_tag,incr)
        end
      else ((TNTopIDRef(p)) := ((left+right) div w_two))
    end
in
  fun addTop(node, value) =
    let val TN(prevRef,ref pID,_,nextRef as ref next) = node
        val newNode = TN(ref(node), ref(pID), ref(value), ref(next))
        val TN(nextPrevRef,_,_,_) = next
        val _ = nextRef := newNode
        val _ = nextPrevRef := newNode
        val _ = issueTopLevelNumber newNode
    in newNode
    end
  fun deleteTop node =
    ((TNPrevRef(!(TNNextRef node))) := !(TNPrevRef node);
     (TNNextRef(!(TNPrevRef node))) := !(TNNextRef node))
end
```

```
(***********************************************************)
(**    BOTTOM LEVEL                                       **)
(***********************************************************)

  val maxVal = (fromInt(~1) div 0w2)+0w1;
  val lSize = fromInt(32)
  val gapSize = maxVal div lSize
  val endV = maxVal - gapSize
  val slackRef = ref SLACK

  datatype ChildNodeV =
    SLACK
  | CN of (ChildNodeV ref TopLevelNode * (* parent *)
           word *                        (* ID *)
           ChildNodeV ref)               (* next *)
  type ChildNode = ChildNodeV ref
  type t = ChildNode

  fun isSplicedOut node = ((!node) = SLACK)

  local
    fun balanceList (child, parent) =
      let fun processBlock (node, num) =
            case !node of
              SLACK => node
            | CN(_, _, next) =>
               (if (num>=endV) then
                  (node := CN(parent, num, slackRef); next)
                else
                  (node := CN(parent, num, next);
                   processBlock (next, num+gapSize)))
          val nextBlock = processBlock(child,0w0)
      in case !nextBlock of
           SLACK => ()
         | _ => balanceList(nextBlock, addTop(parent, nextBlock))
      end
  in
    fun add node =
      case !node of
        SLACK => raise badNode
      | CN(parent, ID, next) =>
          let val nextID = case !next of
                              CN(_, nextID, _) => nextID
                            | SLACK => maxVal
              val newID = Word.>>(ID + nextID, 0w1)
              val newNode = ref(CN(parent, newID, next))
          in
             (node := CN(parent, ID, newNode);
              (if (newID = ID) then
                  balanceList(TNData parent, parent)
               else ());
              newNode)
          end
    fun addNew (node,n) =
      case !node of
        SLACK => raise badNode
      | CN(parent, ID, next) =>
          let
            val nextID = case !next of
                            CN(_, nextID, _) => nextID
                          | SLACK => maxVal
            val newID = Word.>>(ID + nextID, 0w1)
```

```
          val _ = n := CN(parent, newID, next)
        in
          (node := CN(parent, ID, n);
           (if (newID = ID) then
               balanceList(TNData parent, parent)
            else ());
           n)
        end
end
fun new () = ref SLACK
fun compare (node1, node2) =
  case (!node1,!node2) of
   (CN(parent1, ID1, _), CN(parent2, ID2,_)) =>
    let val TN(_,ref parent1ID,_,_) = parent1
        val TN(_,ref parent2ID,_,_) = parent2
        val comp = Word.compare(parent1ID, parent2ID)
     in  case comp of
           EQUAL => Word.compare(ID1, ID2)
         | x => x
     end
  | _ => raise badNode
fun getNext node =
  let val n = case !node of
                SLACK => NONE
              | CN(parent, myid, next) =>
                  case !next of
                    SLACK => SOME(TNData(!(TNNextRef parent)))
                  | _ => SOME(next)
  in case n of
      NONE => NONE
    | SOME(next) =>
        case compare (node,next) of
          LESS => n
        | _ => NONE
  end
fun next node =
  case !node of
    SLACK => raise badNode
  | CN(parent, myid, next) =>
      case !next of
        SLACK => TNData(!(TNNextRef parent))
      | _ => next
fun spliceOut (start,stop) =
  let
    fun deleteRange’ (next,parent) : ChildNode =
      if (next=stop) then stop
      else
        case !next of
          CN(_, _, nextNext) =>
             (next := SLACK;
              deleteRange’(nextNext,parent))
        | SLACK =>
           let
             val nextParent = !(TNNextRef parent)
             val TN(_,_,nextChildRef,_) = nextParent
             val newNext = deleteRange’(!nextChildRef,nextParent)
           in
             ((case !newNext of
                 SLACK => deleteTop(nextParent)
               | _ => nextChildRef := newNext);
              slackRef)
           end
```

```
    in
      case !start of
        SLACK => raise badNode
      | CN(parent, ID, next) =>
          case (compare(start,stop)) of
              LESS => (start := CN(parent,ID,deleteRange'(next,parent)); 0)
            | _ => 0
    end
  fun spliceOut'(start, stop) =  spliceOut(start,next(stop))
  fun getFirst node =
    let fun fstP p =
        let val TN(ref prev, ref ID, ref child,_) = p
            val TN(_ , ref pID,_,_) = prev
        in if pID >= ID then child
           else fstP prev
        end
    in case !node of
         SLACK => raise badNode
       | CN(p,_,_) => fstP p
    end
  fun init () =
    let val newNode = ref SLACK
        val TNprevRef = ref(NULL)
        val TNnextRef = ref(NULL)
        val newParent = TN(TNprevRef, ref(w_zero), ref(newNode), TNnextRef)
        val _ = TNprevRef := newParent;
        val _ = TNnextRef := newParent;
        val _ = newNode := CN(newParent, w_zero, slackRef)
     in newNode
     end
end
```

A.7 Priority Queues

The interface for the priority-queue data structure is show below. The library requires the priority-queue data structure to support the `deleteMin` operation but not a general-purpose `delete` operation. The `findMin` operation is required to return a valid, (not spliced out) read. Based on standard priority queues, it is easy to give an implementation for this interface.

```
signature PRIORITY_QUEUE =
sig
  type elt
  type t
  exception Empty

  val empty : t
  val isEmpty : t -> bool
  val findMin : t -> elt option * t
  val insert : elt * t -> t
  val deleteMin : t -> elt * t
end
functor PriorityQueue (structure Element:
                       sig type t
                           val compare: t*t -> order
                           val isValid: t -> bool
                       end) : PRIORITY_QUEUE =
struct ... end
```

Bibliography

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

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *To appear in ACM Transaction on Programming Languages and Systems*.

[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[4] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.

[5] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

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

[7] P. K. Agarwal, D. Eppstein, L. J. Guibas, and M. R. Henzinger. Parametric and kinetic minimum spanning trees. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 596–605, 1998.

[8] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[9] J. Allen. *Anatomy of LISP*. McGraw Hill, 1978.

[10] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming*, pages 270–280, 1997.

[11] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully-dynamic trees with top trees, 2003. The Computing Research Repository (CoRR)[cs.DS/0310065].

[12] A. W. Appel and M. J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

[13] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.

[14] J. Basch, L. J. Guibas, C. D. Silverstein, and L. Zhang. A practical evaluation of kinetic data structures. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 388–390, New York, NY, USA, 1997. ACM Press.

[15] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[16] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Lecture Notes in Computer Science*, pages 152–164, 2002.

[17] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[18] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, Dec. 1980.

[19] M. Blume. Toyota Technological Institute at Chicago.

[20] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 617–626, 2002.

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

[22] N. H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[23] R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–61, 1991.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[25] A. Czumaj and C. Sohler. Soft kinetic data structures. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 865–872, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[26] F. Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In *Proceedings of Thirteenth International Symposium on Algorithms and Computation (ISAAC)*, volume 3038 of *Lecture Notes in Computer Science*, pages 662 – 669. Springer, 2004.

[27] F. Darema. Dynamic data driven applications systems: New capabilities for application simulations and measurements. In *Proceedings of Thirteenth International Symposium on Algorithms and Computation (ISAAC)*, volume 3515 of *Lecture Notes in Computer Science*, pages 610 – 615. Springer, 2005.

[28] M. de Berg, O. Schwarzkopf, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[29] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

[30] 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.

[31] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, Feb. 1989.

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

[33] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.

[34] J. Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems*. PhD thesis, Department of Computer Science, Cornell University, Nov. 1991.

[35] 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.

[36] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14:781–798, 1985.

[37] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Computing*, 26:484–538, 1997.

[38] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997.

[39] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *JACM*, 35(4):921–940, 1988.

[40] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. *SIAM Journal of Computing*, 28(2):612–636, 1998.

[41] E. Goto and Y. Kanada. Hashing lemmas on time complexities with applications to formula manipulation. In *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 154–158, 1976.

[42] R. L. Graham. An efficient algorithm for determining the convex hull of a finete planar set. *Information Processing Letters*, 1:132–133, 1972.

[43] L. J. Guibas. Kinetic data structures—a state of the art report. In *Proceedings of the Third Workshop on Algorithmic Foundations of Robotics*, 1998.

[44] J. Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[45] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 594–604, London, UK, 1997. Springer-Verlag.

[46] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[47] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on PLDI*, pages 311–320, May 2000.

[48] J. Hilden. Elimination of recursive calls using a small table of randomly selected function values. *BIT*, 16(1):60–73, 1976.

[49] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

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

[51] R. Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the 1992 ACM SIGPLAN Conference on PLDI*, pages 261–272, June 1992.

[52] R. J. M. Hughes. Lazy memo-functions. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, 1985.

[53] S. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[54] H. Kaplan, R. E. Tarjan, and K. Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 836–844, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[55] M. I. Karavelas and L. J. Guibas. Static and kinetic geometric spanners with applications. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 168–176. Society for Industrial and Applied Mathematics, 2001.

[56] Y. A. Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach*. PhD thesis, Department of Computer Science, Cornell University, Jan. 1996.

[57] Y. A. Liu, S. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

[58] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *European Symposium on Programming*, pages 288–305, 1999.

[59] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.

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

[61] D. Michie. 'memo' functions and machine learning. *Nature*, 218:19–22, 1968.

[62] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 487–489, 1985.

[63] G. L. Miller and J. H. Reif. Parallel tree contraction, part i: Fundamentals. *Advances in Computing Research*, 5:47–72, 1989.

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

[65] J. Mostov and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 165–172, Aug. 1985.

[66] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[67] K. Mulmuley. Randomized multidimensional search trees (extended abstract): dynamic sampling. In *Proceedings of the seventh annual symposium on Computational geometry*, pages 121–131. ACM Press, 1991.

[68] K. Mulmuley. Randomized multidimensional search trees: Further results in dynamic sampling (extended abstract). In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 216–227, 1991.

[69] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling (extended abstract). In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 180–196, 1991.

[70] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.

[71] T. Murphy, R. Harper, and K. Crary. The wizard of TILT: Efficient(?), convenient and abstract type representations. Technical Report CMU-CS-02-120, School of Computer Science, Carnegie Mellon University, Mar. 2002.

[72] P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, pages 91–98, 1991.

[73] M. H. Overmans and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[74] M. H. Overmars. Dynamization of order decomposable set problems. *Journal of Algorithms*, 2:245–260, 1981.

[75] M. H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[76] M. Pennings. *Generating Incremental Attribute Evaluators*. PhD thesis, University of Utrecht, Nov. 1994.

[77] M. Pennings, S. D. Swierstra, and H. Vogt. Using cached functions and constructors for incremental attribute evaluation. In *Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, pages 130–144, 1992.

[78] F. Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166. Computer Society Press, 1995.

[79] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[80] J. Polakow and F. Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 130–144. Springer-Verlag LNCS 1581, 1999.

[81] W. Pugh. *Incremental computation via function caching*. PhD thesis, Department of Computer Science, Cornell University, August 1988.

[82] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

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

[84] T. Radzik. Implementation of dynamic trees with in-subtree operations. *ACM Journal of Experimental Algorithms*, 3:9, 1998.

[85] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 502–510, Jan. 1993.

[86] J. Reppy. University of Chicago.

[87] T. Reps. *Generating Language-Based Environments*. PhD thesis, Department of Computer Science, Cornell University, Aug. 1982.

[88] T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on POPL*, pages 169–176, Jan. 1982.

[89] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 197–206, 1991.

[90] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

[91] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[92] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[93] J. M. Spitzen and K. N. Levitt. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, 1978.

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

[95] R. Tarjan and R. Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[96] R. E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:167–177, 1997.

[97] M. N. Wegman and L. Carter. New classes and applications of hash functions. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 175–182, 1979.

[98] R. Werneck. Princeton University.

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

[100] Y. Zhang and Y. A. Liu. Automating derivation of incremental programs. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, page 350. ACM Press, 1998.