Incremental Computation and the Incremental Evaluation of Function Programs

William Worthington Pugh, Jr.
Ph.D. Thesis

TR 88-936
August 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

INCREMENTAL COMPUTATION AND THE INCREMENTAL EVALUATION OF FUNCTIONAL PROGRAMS

William Worthington Pugh, Jr., Ph.D.
Cornell University, 1988

Incremental computation is generally thought of as the technique of efficiently updating the result of a computation when the input is changed. This idea is used in doing semantic checking in programming environments, document formatting in WYSIWYG editors and other applications. More generally, incremental computation is the technique of efficiently applying a function to a series of similar inputs.

Much of the previous work on incremental computation has centered on incremental attribute grammar and incremental dependency graph evaluation schemes, but these techniques are only suitable for certain applications. This thesis examines an alternative method for providing incremental computation. Our results provide practical methods that can be used for applications such as theorem provers for which attribute grammars are unusable. Even for those problems for which attribute grammars are best suited, our methods perform almost as well as attribute grammars.

We describe an incremental evaluator for functional programs that makes use of function caching. Function caching, or memoising, allows reuse of solutions to problems that were solved previously. We examine how function caching can be effectively used when solving problems that are similar to problems that were solved previously.

In order for function caching to provide incremental evaluation, two similar problems must be solved by decomposing them into sub-problems in such a way that they share many common sub-problems. We formalize and quantify this idea with the notion of a stable decomposition, and we present data structures for representing sets and sequences that have stable decompositions.

We solve several problems related to the efficient implementation of function caching. To perform function caching efficiently, one must be able to determine if two values are equal in constant time and perform updates applicatively. The data structures we present for sets and sequences support these features. This was previously an open problem for representations that also supported efficient updates. We also examine how to calculate hash keys and perform fast equality tests for S-expressions and how to determine what to purge from a function cache when it is full. We report benchmarks that show our function caching implementation produces significant speed-ups in complex programs such as incremental theorem provers.

Biographical Sketch

William W. Pugh, Jr. was born in 1960 in Cincinnati, Ohio to William W. Pugh and Elizabeth R. Pugh. He received a B.S. degree in Computer Science from Syracuse University in 1980. Having decided that school was not appropriately challenging, he started work for Texas Instruments in Dallas. It soon became apparent that the real world was not appropriate for William, and in 1983, he entered the graduate program in Computer Science at Cornell University. In 1988, he received a Ph.D. in Computer Science with a minor in Acting. At last report, he was seen headed into the wilds of academia.

to my parents

Acknowledgments

My years at Cornell were the most educational years of my life, both in Computer Science as well as in many other areas. Many people deserve thanks for this.

Tim Teitelbaum, my major advisor, helped me mature into a researcher and taught me how to survive in the world of academia. Tim gave me the freedom to explore my many areas of interest and to develop a thesis that was difficult to propose until it was almost finished.

David Gries provided good feedback on my thesis and on an earlier but abortive thesis proposal. Alan Demers and Charlie Fisher showed me that you can be a professor and still think something is "neat". Steven Cole, David Feldshuh and Bruce Levitt of the theater department were all supportive of my desire to obtain a minor in Acting.

Many special friends were a part of my years at Cornell.

Susan Shoaf did to me "what spring does with the cherry trees" [– Pablo Neruda]. She taught me to appreciate licorice, musicals, art museums, Borzi and much more, and inspired me to grow in many ways.

Brad Vander Zanden was ready to discuss either the meaning of life or closed forms for complicated formulas at the drop of the hat. Many discussions with him were invaluable.

Courtney Nicholson and Jayne Uerling convinced me that I wasn't the only person in the world who thought that dancing should be something more than having an epileptic fit on the dance floor. Without them, my last year at Cornell would have been much less fun. Special thanks to Jayne for lessons and a sympathetic ear, and to Courtney for general craziness and some truly intense experiences.

In addition to being a good friend, Hillary Ford encouraged me become involved with grad orientation and Grads for Grads; as a result, I made many special friends.

Special thanks to the entire Grads for Grads gang, who helped make it happen.

Bell Labs provided me with a fellowship for most of my stay at Cornell. This fellowship allowed me to devote my full attention to research, and was deeply appreciated. General Electric provided me with a fellowship for my first year at Cornell, and this was also helpful and appreciated.

And very special thanks to my parents. The atmosphere they provided launched me on my journeys, intellectual and otherwise.

Table of Contents

1 Introduction 1

1.1 Incremental Methods 1

1.2 This Thesis 3

1.3 Terminology 4

2 Incremental Dependency Graph Evaluation 6

2.1 A Formal Definition of Dependency Graphs 7

2.2 Graph Evaluators 7

2.3 Cyclic Dependency Graphs 8

2.4 Incremental Evaluation of Dependency Graphs 8

2.5 Problem Solving with Dependency Graphs 11

2.6 The Limitations of Incremental Dependency Graph Evaluators 12

3 An Incremental Evaluator for Functional Languages 14

3.1 Function Caching 14

3.2 Incremental Evaluation via Function Caching 14

4 The Implementation of Function Caching 18

4.1 Hash functions 20

4.2 Equality tests for non-atomic values 23

4.3 Deciding which functions to cache 27

4.4 Obsolescence-based purging schemes 27

4.5 Predictive-based purging schemes 28

5 Randomized Data Structures 37

5.1 Probabilistic upper bounds on random variables 37

5.2 The probabilistic big-Oh 38

6 Data Structures and Algorithms for Incremental Computation 39

6.1 Representation schemes 39

6.2 Data structures and algorithms for function caching 39

6.3 Data structures and algorithms for incremental evaluation 40

6.4 Hash keys for data structures 42

6.5 Tagged Tuples 43
6.5 Discussion 43

7 A Decomposition and Representation Scheme for Sequences 44
7.1 The chunky decomposition scheme 44
7.2 Duplicate elements 48
7.3 The chunky list representation scheme 48

8 A Decomposition and Representation Scheme for Sets 51
8.1 Set operations 52
8.2 Analysis 53
8.3 Relationship to previous work 61
8.4 Using binary hash tries to implement other abstract data types 63

9 Other Approaches to Incremental Computation 65
9.1 Incremental Constraint Solving 65
9.2 Supply-driven evaluation 65
9.3 Incremental Attribute Grammar and Dependency Graph Evaluation 66

10 Conclusions 69
10.1 Future research 69
10.2 Summary 71

Appendix – Negative Binomial Distributions 73

Bibliography 129

List of Figures

2.1 Sample dependency graph 6

2.2 Sample dependency graph after a change 6

2.3 A demand driven graph evaluator 8

2.4 A propagation style graph evaluator 8

2.5 Nullification and reevaluation algorithm 9

2.6 Naive propagation algorithm 10

2.7 A dependency graph with poor performance under naive propagation 10

2.8 An optimal propagation algorithm 10

3.1 An algorithm to produce a list of the squares of a list of numbers 14

3.2 Computations results from the use of the algorithm in Figure 3.1 15

3.3 A divide and conquer version of the algorithm in Figure 3.1 15

3.4 Calls arising from a call on $ss[1, 2, 3, 4, 5, 6, 7, 8]$ 16

3.5 Calls arising from a call on $ss[1, 2, 3, 4, 5, 6, 7, 9]$ 16

3.6 Calls arising from a call on $ss[0, 1, 2, 3, 4, 5, 6, 7, 8]$ 17

4.1 A description of an extension to an apply function so that all calls are cached 19

4.2 Graphs of experimental results of the use of different hash functions 22

4.3 A description of an algorithm to perform hashed consing 24

4.4 Use of signatures to obtain fast inequality tests 24

4.5 The addition of lazy structure sharing to the algorithm in Figure 4.4 25

4.6 Two equal but duplicate values 25

4.7 The results of testing the values in Figure 4.6 for equality using lazy structure sharing 25

4.8 Steps involved in purging obsolete cache entries from a function cache 27

4.9 Partial computation graph for a symbolic derivation function 29

4.10 Equations used in defining the potential of a cache 29

4.11 An example showing non-optimal performance for eliminating multiple cache entries 31

4.12 Number of calls required to compute *A(85, 9, 9)* with different cache sizes and policies 31

4.13 Graphs of benchmarks Incremental 1a and 1b 34

7.1	The chunky decomposition of a sequence	45
7.2	The chunky decomposition of a sequence similar to the one in Figure 7.1	46
7.3	Description of an algorithm to append the representations of two sequences	49
7.4	Description of algorithms extract the first or last *i* elements of a sequence	50
7.4	Description of algorithms to access, change, delete or insert elements in a sequence	50
8.1	The representation of a set as a binary hash trie	52
8.2	A description of the algorithm for set union	53
8.3	The result of a dyeing operation	55
8.4	The coloring that results when nodes are dyed as described	57
9.1	An example of a conversion from an attribute grammar to a set of recursive functions	67

List of Tables

2.1 Operations that modify a dependency graph 9

4.1 Benchmarks comparing two cache purging strategies 34
4.2 Real-time benchmarks comparing two cache purging strategies 35

8.1 Hash keys of the elements in Figure 8.1 52
8.2 Results of mixing dyes 57
8.3 Operations that can be supported on finite functions 63
8.4 Implementation of finite function operators 64

List of Definitions

5.1	A partial order on the probability distribution of random variables	37
5.2	Negative binomial distributions — $NB(s,p)$	37
5.3	Probabilistic order bounds — O_{prob}	38
6.1	Representation function	39
6.2	Transformation	40
6.3	Distance between abstract values — $tDistance_T(x, x')$	40
6.4	Decomposition scheme	41
6.5	Decomposition — $d_D(x)$	41
6.6	Distance between decompositions — $dDistance_D(x, x')$	41
6.7	Stable decompositions	41
7.1	Level of an element	44
7.2	Chunky decomposition scheme	44
8.1	Decomposition scheme for annotated-sets	51
8.2	$select(S,L)$	54
8.3	Level of an node in a binary hash trie	54
8.4	Nodes in a binary hash trie matched by an element	54
8.5	Dyes and colors	55
8.6	$Overlap(A, B, k)$	56

List of Notations

Notation	Definition	Defined in
$\langle x_0, x_1, ..., x_n \rangle$	The tuple that contains the values $x_0, x_1, ..., x_n$.	§ 4
$\mathsf{Id}\langle x_0, x_1, ..., x_n \rangle$	The tuple tagged with Id that contains the values $x_0, x_1, ..., x_n$.	§ 6.5
Prob{ E }	The probability of event E.	
$X =_{prob} Y$	$\forall\, x$, Prob{ $X \le x$ } = Prob{ $Y \le x$ }.	Def. 5.1
$X \le_{prob} Y$	$\forall\, x$, Prob{ $X \le x$ } $\le$ Prob{ $Y \le x$ }.	Def. 5.1
$NB(s, p)$	A variable with a negative binomial distribution: the probability distribution of the number of failures seen in in a series of random trials before the s^{th} success, were the probability of success in a trial is p.	Def. 5.2
$NB(s)$	Denotes $NB(s, 0.5)$.	Def. 5.2
$g(n)$ is $O_{prob}(f(n))$	$\forall\, \varepsilon$ s.t. $\varepsilon > 0$, $\exists\, M, n_0$ s.t. $\forall\, n$ s.t. $n \ge n_0$, Prob{ $\lvert f(n) \rvert \le M \lvert g(n) \rvert$ } $\ge 1-\varepsilon$}	Def. 5.3
$f \circ g$	Functional composition: $(f \circ g)(x) = f(g(x))$	
ε	The empty string	
lg	log base 2	
$\lvert S \rvert$	The number of elements in the set or sequence S.	
$\lfloor r \rfloor$	The largest integer at most r.	
$\lceil r \rceil$	The smallest integer at least r.	

1 Introduction

One of the strongest trends in the use of computers over the past three decades has been for the user's dialog with the computer to become more interactive, more conversational, more immediate. In the days of card decks and line printers, changing a program or even the input data for a program, required submitting the change and waiting several hours for an updated answer. On-line computer systems reduced the response time to minutes or seconds. Visicalc, the first spreadsheet program, brought into wide use an even more interactive form of dialogue, *immediate computation*[1]: any change to the data caused the answer to be updated automatically to provide the user with instant feedback.

Today, immediate computation is used in many contexts. Program editors immediately detect and display semantic errors such as undeclared variables. WYSIWYG[2] word processors immediately compute changes to word wrapping, hyphenation and pagination in response to each change in the text being edited. Chess programs re-evaluate their board position and strategy after each move. Telephone message routing systems immediately compute changes in routing schemes in response to changes in message traffic.

1.1 Incremental Methods

To be useful, immediate computation must not introduce long and annoying delays while the system updates the results. Visicalc recalculates the entire spreadsheet after each change. This is satisfactory for small spreadsheets, but since it becomes unacceptably slow for larger ones, an option is provided to turn off automatic recalculation. As immediate computation is applied to larger and more difficult problems, the time required to recalculate the result from scratch after each change becomes longer, rendering immediate computation an annoyance instead of a desirable feature. One way to allow immediate computation of more difficult problems is to use a better algorithm or a faster machine. However, after making a small change to the input, much of the computation required to produce the result may be similar or identical to the computation that produced the result for the previous version of the problem. Techniques that save time by reusing or updating the results of previous computations are known as *incremental methods*.

Many people have studied incremental algorithms for different problems such as compilation, data flow analysis, parsing and other classic problems. Generally, these problems have been tackled with an explicitly incremental algorithm: an algorithm that not only specifies how to compute the result from the input, but also specifies how to update the result in response to changes in the input.

Explicitly Incremental Algorithms

Consider the problem of sorting a list of numbers. An explicitly incremental algorithm for this problem could handle updates as follows. Assume the list contains n elements.

- If an additional number is inserted into the input, search the sorted output to find the appropriate place to insert the new value, requiring $O(\log n)$ time.

[1] also called *continuous evaluation* in [Henderson & Weiser]

[2] What You See Is What You Get - pronounced *Wiz-ee-wig*

- If a number is deleted from the input, search the sorted output to find the value to delete, requiring $O(\log n)$ time.
- A modification of a number is treated as a deletion followed by an insertion.

Design and maintenance of explicitly incremental algorithms

Regardless of how the update is performed, the user expects that the current answer is exactly what would be obtained if the algorithm were run from scratch on the current input data; the series of modifications that created the current input data is irrelevant. We must ensure that changing x and then y always produces the same result as changing y and then x and that making a change and then undoing it always produces the original result. For complicated problems, this makes explicitly incremental algorithms hard to derive, debug and maintain.

As computers become more powerful, people's expectations of computers rise as well. As the immediate computation model is applied to more difficult problems, the technique of writing explicitly incremental algorithms does not scale up easily, because the problems involve more complicated algorithms, requiring prohibitively complex explicitly incremental algorithms.

Incremental Evaluators

The difficulties inherent in designing explicitly incremental algorithms motivate a desire for an evaluator that can incrementally execute an algorithm that simply specifies how to compute the result from scratch. The goal of designing and building an incremental evaluator is to make it easier to construct a program that can incrementally update its result. The designer of an algorithm for use with an incremental evaluator can put aside the question of incremental updates when worrying about the correctness of the algorithm.

An incremental evaluator will not necessarily be able to obtain the best incremental performance obtainable for that problem; the incremental performance obtained will depend on both the evaluator and the algorithm used, in much the same way that the amount of parallelism that can be extracted from an algorithm by a compiler depends on the compiler, the target machine and the algorithm chosen. An examination of incremental computation involves not only incremental evaluators but also the design and performance of algorithms and data structures for incremental evaluators.

As an example of a situation where incremental computation would be useful, one might wish to create a program verification editor that allowed the user to annotate a program with post-conditions, pre-conditions and loop invariants. Such an editor would determine the proof obligations: statements that need to be proved to verify that the program is correct. As the user edited his program, the proof obligations would be updated after each change to show the user what still needed to be taken care of. Without some form of logic simplification, the proof obligations would be long and unreadable, and include trivial obligations such as proving that $x = 0$ implies $x = 0$. For the editor to be usable, it would need to incorporate a logic simplifier that would simplify the proof obligations as much as possible without user intervention.

The Synthesizer Generator [RT84] is a system for building context-sensitive editors such as program editors or proof editors. The Synthesizer Generator provides an incremental attribute grammar evaluator to allow incremental semantic checking. Incremental attribute grammar evaluation works well for applications such as incremental type checking and compilation. However, it appears impossible to encode a logic simplifier in a form suitable for incremental attribute grammar evaluation.

1.2 This Thesis

Most of the previous work in incremental evaluators has centered on incremental dependency graph schemes. There appear to be many problems, such as theorem proving, for which incremental dependency graph schemes are unusable. *Function caching*, or *memoising* [Mic68] is the technique of remembering the solutions to problems that were solved previously. Function caching offers a different approach to incremental evaluation that is suitable for many of the problems for which incremental dependency graph techniques are unusable. We describe and formalize the situations in which function caching can be used to achieve incremental evaluation. We also solve a number of problems related to the efficient implementation and utilization of function caching.

An incremental evaluator for functional programs can be a valuable component of a tool set for building interactive systems. This thesis argues that:

- The need for incremental evaluators in interactive systems is increasing.
- Previous work on incremental evaluators has centered on incremental dependency graph evaluators [DRT81] [Rep84]. This technique appears to be unusable for many problems.
- The addition of function caching to an evaluator for a functional language produces an incremental evaluator that is suitable for many problems for which incremental dependency graph evaluators are unsuitable.
- Several simple design principles lead to algorithms that have good incremental performance when evaluated using function caching.
- Even for those problems best suited for use with Reps's incremental attribute grammar evaluator [Rep84], function caching can provide almost equivalent incremental performance.

The major contributions of this thesis are:

- an understanding of how function caching provides incremental evaluation,
- solutions to several problems involved in the implementation of hashing and of function caching,
- data structures and algorithms for representing sets and sequences that are applicative, can be updated quickly, can be tested for equality in constant time, and are well suited for use with function caching for providing incremental evaluation, and
- new techniques for analyzing probabilistic algorithms and random data structures.

Chapter 2 reviews the technique of incremental dependency graph evaluation, which is based on the idea of efficiently updating a specific previous solution. For some applications, we can predict that sub~problems similar to previous sub~problems will arise, but we cannot establish *apriori* an appropriate one~to~one correspondence between new sub~problems and previous sub~problems. For these applications, incremental dependency graph evaluation techniques seems unusable.

Function caching, or memoising, is the technique of remembering the results of previous function calls and saving time by avoiding their recomputation. This saves time when we solve problems that are identical to problems solved previously. Chapter 3 explores the situations in which function caching saves time when solving problems that are *similar*, but not identical, to problems that were solved previously. To do this, we must structure our algorithms and data structures so that the computations involved in solving two similar problems include common sub~problems; if we can do this, we can reuse results from *any* similar problem solved previously.

Function caching is an old idea but its efficient implementation requires solutions to several problems that earlier work on function caching left unsolved or that need new solutions because of our particular application. These results are discussed in Chapter 4.

Chapter 5 describes some new techniques for analyzing probabilistic data structures and algorithms that are used extensively in Chapters 7 and 8.

Chapter 6 formalizes the idea discussed in Chapter 3. We define the idea of *stable decompositions*: if a scheme for representing and decomposing problems has a stable decomposition, any two similar problems have similar decompositions. The idea of a stable decomposition is contingent on what we consider similar problems and similar decompositions, so we formally define these ideas.

The discussions in Chapter 6 raise a problem: the standard representations for abstract data types such as sets or sequences do not satisfy our requirements. Both as examples and to provide a set of basic tools, new data structures and algorithms for sequences and sets are described in Chapters 7 and 8. These data structures have applications outside of incremental computation; the data structures are efficient, simple and allow constant~time equality tests. The data structures and algorithms for sets improve on previous lower bounds for non~incremental set operations involving two similar sets. Analyzing these algorithms and data structures required the development of several new analysis techniques that should prove widely useful for analyzing probabilistic algorithms.

Chapter 9 contains a comparison of function caching and a number of other incremental evaluation methods. Chapters 2 and 3 discuss why many problems are suitable for function caching but not for incremental dependency graph techniques; Chapter 9 includes a comparison of function caching and incremental attribute-grammar evaluation techniques on those problems best suited for incremental attribute grammar evaluation. Chapter 10 summarizes the results of this thesis and suggests future research directions. Appendix A presents some lemmas and theorems concerning negative binomial distributions that are used in this thesis, but not of direct interest to most readers.

1.3 Terminology

Batch evaluation — Performing a computation from scratch (i.e., not using the results of previous similar problems).

Explicitly incremental algorithm — An algorithm that specifies not only how to compute the output from the input, but also how to update the output in response to changes to the input.

Immediate computation — A computational paradigm in which any change to the input data causes the answer to be updated automatically to provide the user with instant feedback about the effects of his changes.

Incremental computation	The technique of saving time by reusing or updating the results of previous computations.
Incremental evaluator	An evaluator that can evaluate incrementally an algorithm that only specifies how to compute the output from the input.
Incremental performance	The speed with which the result of a computation can be updated in response to a change in the input.

2 Incremental Dependency Graph Evaluation

Much of the previous work in incremental evaluators has revolved around incremental dependency graph evaluators, starting with a paper by Reps, Teitelbaum and Demers [RTD81]. Incremental graph evaluation has proven useful for problems such as type-checking and compilation. However, the dependency graph approach appears ill-suited to some problems. This chapter reviews previous work in incremental dependency graph evaluators and explores their limitations in order to motivate the exploration of other approaches for incremental evaluation.

A dependency graph evaluation scheme assigns values to each vertex of a *dependency graph*. Informally, each vertex v has a value function that determines the value of v based on the values of the vertices with incoming edges incident to v. A vertex with no incoming edges is a constant-valued vertex. Figure 2.1 shows a dependency graph with a consistent assignment of values to vertices. This graph would be used to calculate the sum of a set of integers. Once solved, a dependency graph can be modified by changing the function at a vertex or by adding or deleting edges or vertices. The change can be propagated through the dependency graph to reestablish a consistent assignment of values to vertices. Figure 2.2 shows how the graph in Figure 2.1 is updated after a modification to the graph.

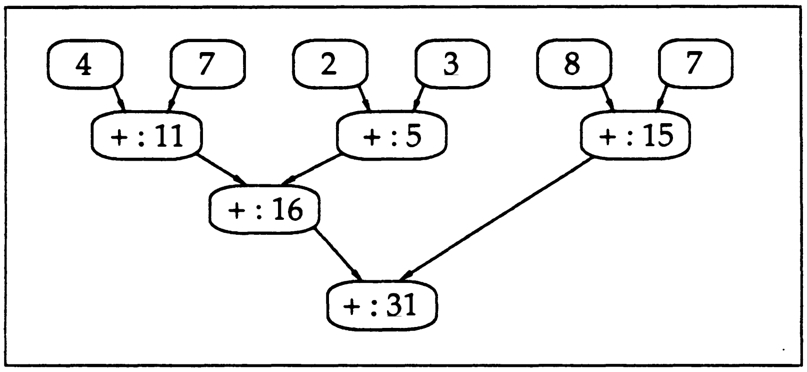

FIGURE 2.1 - Sample dependency graph

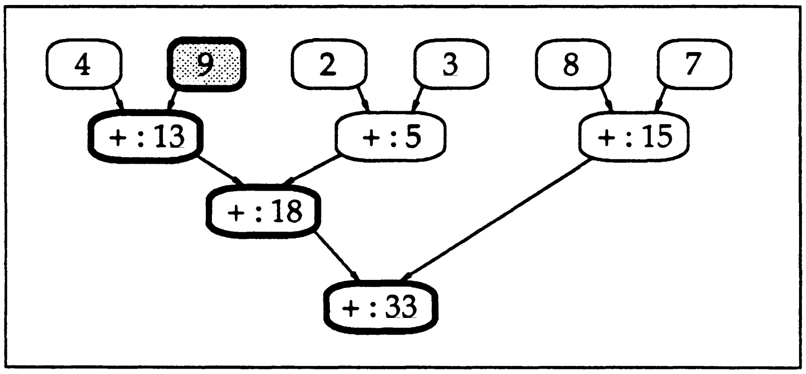

FIGURE 2.2 - Changing the value of the grey vertex results in a change being propagated through the dependency graph

2.1 A Formal Definition of Dependency Graphs

Much of this description is adapted from that offered by Hoover [Hoo87]. An unevaluated dependency graph is denoted by

$D = \langle V, pred, F, R \rangle$

where

V is a finite set of vertices,

pred maps a vertex to its predecessors in the dependency graph (i.e., $pred(v) = [v_1, v_2, ..., v_k]$, where $v_1, v_2, ..., v_k$ are the predecessors of v),

F is a mapping from vertices to functions, and

R is the set of result vertices, $R \subseteq V$.

Although $pred(v)$ is defined as a sequence, we also refer to it as a set when appropriate. The transitive closure of the predecessor relation is denoted by $pred^+$: $pred^+(v) = \{ p \mid p \in pred(v) \vee \exists\, w$ s.t. $w \in pred(v) \wedge p \in pred^+(w) \}$. The set of successors of a vertex v are defined as $succ(v) = \{ u \mid v \in pred(u) \}$ and the transitive closure of the successor relation is denoted by $succ^+$.

We use the notation $g(map\, f\, [x_1, x_2, ..., x_n])$ to denote $g(f(x_1), f(x_2), ..., f(x_n))$.

A dependency graph is evaluated by assigning values to the vertices in the graph. The value of a vertex can be $\perp$, indicating that it is currently unknown. The value of a vertex, $Val(v)$, is consistent if and only if either $Val(v) = \perp$ or ($\forall\, p$ s.t. $p \in pred(v)$, p is consistent) and $Val(v) = F(v)(map\ Val\ pred(v))$ (i.e., $Val(v)$ is $F(v)$ applied to the values of v's predecessors). If a vertex v has no predecessors, $F(v)$ is a constant valued function.

If $Val(v)$ is known and consistent for all vertices v in R, then Val is a solution for D. If $Val(v)$ is known and consistent for all vertices v, then Val is the *complete* solution for D. It may be possible to compute a solution for D without having to find the complete solution. This can happen if some vertices do not affect the result vertices or if some vertex functions are non-strict (a function is non-strict if its value does not always depend on all the arguments). Most incremental graph evaluation schemes attempt to keep the graph in a state in which the values of all vertices are consistent (although the values of some vertices may be unknown). If the dependency graph includes no cycles there is only a single consistent known value for each vertex and a single complete solution.

The algorithms given in this chapter should be viewed as denotational descriptions of the algorithms, instead of operational specifications of the algorithms. An efficient implementation of these algorithms may require the use of auxiliary data structures and algorithms. For example, to perform

choose $v \in NeedEval$ s.t. $pred(v) \cap NeedEval = \emptyset$

efficiently may require using additional data structures to maintain a topological ordering on the vertices in *NeedEval* with respect to the ordering specified by *pred*.

2.2 Graph Evaluators

Two simple graph evaluators are given below. These evaluators assume there are no cycles in the dependency graph, all functions are well-defined and that $V, F, pred, R$ and Val are globally defined. The algorithm in Figure 2.3, if used with a lazy interpreter, can take advantage of non-strict vertex functions and will only evaluate those vertices that need to be evaluated to compute known and consistent values for the vertices in R. The algorithm in Figure 2.4 will compute a known and consistent value for all vertices.

```
Solve()
  ∀ v ∈ V, Val(v) := ⊥
  ∀ v ∈ R, eval(v)

eval(v)
  If Val(v) = ⊥ then Val(v) := (F(v))(map eval pred(v))
  return Val(v)
```

FIGURE 2.3 - A demand driven graph evaluator

```
Solve()
  NeedEval := V
  while NeedEval ≠ ∅ do
      choose v ∈ NeedEval s.t. pred(v) ∩ NeedEval = ∅
      Val(v) := (F(v))(map Val pred(v))
      NeedEval := NeedEval - {v}
```

FIGURE 2.4 - A propagation-style graph evaluator

2.3 Cyclic Dependency Graphs

Cycles in dependency graphs raise additional complications. Finding a set of consistent assignment of values for the vertices in a cycle requires finding a fixed point of the cycle. Finding a unique and meaningful assignment requires finding the unique *least* fixed point of the cycle. This can be done, although it complicates the graph evaluation algorithms. In incremental graph evaluation, additional problems are raised. When finding a fixed point for a cycle in a dependency graph, we can't just settle on any fixed point; we must compute the *same* fixed point that would be computed if the evaluation was done from scratch (i.e., the least fixed point).

In current methods of handling incremental computation in cyclic dependency graphs [JW88], when change propagates into a cycle, the evaluation of the entire cycle is done from scratch, assuring that the unique least fixed point of the cycle is found. Hence, computations within the cycles are totally non-incremental. Some recent work has attempted to avoid the need to restart completely the cycle [JW88], but a complete reevaluation of the cycle can be avoided only in special circumstances. Since cycles lead to such poor incremental performance and complicate the algorithms involved, we assume in this chapter that all dependency graphs are acyclic.

2.4 Incremental Evaluation of Dependency Graphs

We allow the operations shown in Table 2.1 to update a dependency graph. We might wish to add additional operations to perform such actions as deleting a connected component of the dependency graph. However, all the desired operations can be constructed from the above three operations. The only one of these operations that can cause change to propagate is changeVertex. The operation addVertex calculates a value for the new vertex and may demand the value of vertices that previously were not needed. The algorithms given below take as an argument a vertex that has been changed and update the value assignments for the dependency graph. These algorithms can also be written to take as an argument a set of vertices that have been changed, to allow multiple changes to be performed before updating the values of the dependency graph. The nullification and reevaluation algorithm in Figure 2.5 works by nullifying (i.e., erasing the values of) all the vertices that depend on v and then using the demand evaluation technique of Figure 2.3.

changeVertex(v, f, p)	Set $F(v) = f$ and $pred(v) = p$.
addVertex($v, f, p, result$)	Add a new vertex v and set $F(v) = f$ and $pred(v) = p$. If $result$ = true, add v to R.
deleteVertex(v)	Only defined if $succ(v) = \emptyset$. Delete vertex v.

TABLE 2.1 - Operations that modify a dependency graph

```
Nullify_and_reevaluate(v)
  Val(v) := ⊥
  ∀ w ∈ succ⁺(v), Val(w) ← ⊥
  ∀ w ∈ R, eval(w)

eval(v) - defined as in Figure 2.3
```

FIGURE 2.5 - Nullification and reevaluation algorithm

The problem with the nullification and re-evaluation updating scheme is that the change may quiess: although the value of a predecessor of a vertex v changes, the value of v may not, and if not, the successors of v need not be updated. The naive propagation algorithm in Figure 2.6 handles this by pushing changes forward from the location of the change and checking to see if propagation quiesses. It assumes that the values of all vertices are known. Naive propagation can do quite poorly. For example, examine the graph in Figure 2.7. Assume the vertex functions for vertices $2a$ and $2b$ are simply the identity function and the the vertex function for vertex 3 computes the difference between the values of vertices $2a$ and $2b$. Therefore, the only consistent known value for vertex 3 is 0. If a change is made to the value of vertex 1, naive propagation might propagate the change along the vertices [1, $2a$, 3, 4, 5, ..., 8, $2b$, 3, 4, 5, ..., 8], evaluating the vertices 3 through 8 twice. If the new values for vertices $2a$ and $2b$ were both computed before a new value for vertex 3 was computed, change would quiess after the new value for vertex 3 was computed and found to be unchanged.

```
NaivePropagation(v)
  Changed := {v}
  while Changed ≠ ∅ do
      choose w ∈ Changed
      Changed := Changed - {w}
      newValue := (F(w))(map Val pred(v))
      if Val(w) ≠ newValue then
          Val(w) := newValue
          Changed := Changed ∪ succ(w)
      end do
```

FIGURE 2.6 Naive Propagation algorithm

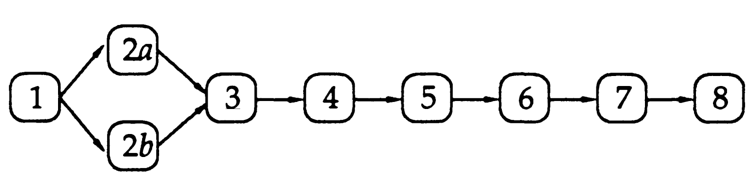

FIGURE 2.7 - Example dependency graph with possible poor performance using Naive Propagation

```
OptimalPropagation(v)
  Changed := {v}
  while Changed ≠ ∅ do
      choose w ∈ Changed s.t. Changed ∩ pred+(w) = ∅
      Changed := Changed - {w}
      newValue := (F(w))(map Val pred(w))
      if Val(w) ≠ newValue then
          Val(w) := newValue
          Changed := Changed ∪ succ(w)
      end do
```

FIGURE 2.8 - Optimal propagation algorithm

To solve the problem with naive propagation, we can use the optimal propagation algorithm of Figure 2.8, as originally suggested by Reps [Rep84a]. Optimal propagation is the same as naive propagation, except that at each step the next vertex chosen is one for which all its predecessors, immediate and transitive, have their final values. Obviously, performing optimal propagation efficiently requires an efficient method for maintaining a topological ordering of the nodes in the dependency graph. Some methods dealing with attribute grammars use information from the attribute grammar to determine a topological ordering of the vertices in the dependency graph [Reps 83] [Reps 84]. Another approach is to maintain a topological ordering of the vertices in the dependency graph and use an explicitly incrementally algorithm to update the topological ordering whenever the dependency graph changes [Hoo87] [ACRSZ87]. A third approach is to use heuristics to determine an approximate topological ordering [Hoo87].

Incremental Evaluation of Vertex Functions

So far, we have ignored the question of what type of vertex functions to allow and how the value of a vertex is updated when the value of one of its predecessors changes. The choices for updating the result of a vertex function are the same as our choices for updating the solution to any computation: recompute the result from scratch, use an explicitly incremental algorithm to update the result, or use an incremental evaluator. The simplest situation is where each vertex function requires only a short, constant time to compute, making it efficient to simply recalculate each vertex function from scratch when necessary. Several existing systems, such as the Synthesizer Generator [RT84], always recalculate vertex functions from scratch, regardless of the cost of the vertex functions.

Several researchers have examined the possibility of using explicitly incremental algorithms for updating vertex functions that are simple computations on sets or relations [HT86] [YS88]. Allowing explicitly incremental algorithms for updating arbitrary vertex functions is a harder problem that has been largely unexamined.

Using an incremental evaluator for updating the results of vertex functions seems promising. However, attempts to use an incremental dependency graph evaluator for this purpose have proven unsuccessful (some reasons for this are discussed in Section 2.6).

2.5 Problem Solving with Dependency Graphs

Any computation, or problem instance, can be represented by a dependency graph with a single vertex, whose vertex function computes the result from the input. However, such a dependency graph would provide no help in obtaining incremental evaluation. To use a dependency graph evaluation scheme, we need a method of converting a computation into an appropriate dependency graph and a method of incrementally updating the dependency graph when the computation to be solved changes. The original papers on incremental dependency graph evaluation concentrated on dependency graphs that arose from attribute grammars [RTD81] [Rep82] [RT84]. Attribute grammars can be thought of as a way of specifying how to compute a dependency graph based on an underlying abstract syntax tree. Whenever the underlying syntax tree is changed, it is easy to compute the necessary changes to the dependency graph (i.e., the attribute grammar is a specification used by an incremental evaluator to maintain the dependency graph). More recently, several papers have attempted to break away from the use of attribute grammars and look at the general problem of updating a dependency graph [Hoo87] [ACRSZ87].

Several techniques other than attribute grammars have been proposed for creating or updating dependency graphs. One technique is to use a framework similar to attribute grammars for specifying the vertices of the dependency graphs, but to allow some edges to be non-local and to be computed dynamically [JF82] [Hoo86] [Hoo87]. In a program editor, non-local edges could be used to connect the use of a variable directly to the definition of the variable. Another possibility is to extend the attribute grammar idea from trees to graphs [ACRSZ87]. Carroll and Ryder [CR88] describe a system in which an explicitly incremental algorithm is used to maintain the dominator graph of a program. A method similar to an attribute grammar is used to maintain a dependency graph based on the dominator graph. This dependency graph is used, in turn, to incrementally maintain data flow information for the program.

2.6 The Limitations of Incremental Dependency Graph Evaluators

The incremental dependency graph evaluation technique seems to have a fundamental problem dealing with algorithms with super-linear time requirements and with functional composition.

Super-linear Time Requirements

If the number of vertices in the dependency graph is linear in the size of the input and the lower bound on the time required to solve the problem is greater than linear, then a mismatch exists. As the problem grows large, more and more computation will be forced onto each vertex of the dependency graph. Unless the vertex computations can be incrementally updated, the incremental performance of the system will degrade as the problem grows larger. Although this can be done for certain vertex functions such as functions involving sets and relations, it seems far from a general solution.

If the number of vertices in the dependency graph is super-linear in the size of the input, then updating the dependency graph for a problem when the input changes becomes a major problem. What is needed is an incremental algorithm for producing the dependency graph from the input. It would be possible to use an explicitly incremental algorithm for maintaining the dependency graph, but we would like to avoid explicitly incremental algorithms if possible. Since a dependency graph that is linear in the size of the input cannot incrementally produce a result whose size is larger than linear in the size of the input, we cannot boot-strap the process by using a linear dependency graph to produce the super-linear dependency graph.

Functional Composition

One of the difficulties involved with using a dependency graph scheme is that the new dependency graph must share as much as possible of the old dependency graph in such as way that most of the values in the old dependency graph can be reused. In practice, this means that to change the dependency graph that computes $f(x)$ into the dependency graph that computes $f(x')$, the system must have available a concise description of the difference between x and x'. If x' and x are similar, but the system does not know how they differ, it will be very difficult to update the dependency graph incrementally. It is generally very difficult to produce a concise description of the difference between two values x' and x [Nis88].

It might seem that we could live with this limitation. The problem is that the output produced by an incremental dependency graph scheme is *the new output* – the *difference* between the old output and the new output is not computed. This means that we can't use this output as input to another incremental dependency graph evaluator and therefore cannot do functional composition (i.e., combine incremental evaluators for $g(y)$ and $f(x)$ to produce an incremental evaluator for $g(f(x))$). It may be possible to extend dependency graph schemes to produce a concise description of the difference between $f(x)$ instead of $f(x')$, but this is currently an open problem.

Conclusions

Taken together, these limitations suggest that current incremental dependency graph techniques are unsuitable for many problems. To be widely applicable, we need two things:

- either a general purpose technique to update vertex functions incrementally or a general purpose technique to update super-linear dependency graphs incrementally, and
- either a method of performing incremental evaluation when simply given a list of similar problems, or a method of incremental computation that produces a description of the difference between the old output and the new output.

3 An Incremental Evaluator for Functional Languages

In its most elementary form, incremental computation is the technique of saving time by reusing or updating the results of previous computations. Function caching, or memoising [Mic68], is the technique of remembering the results of previous function calls and saving time by avoiding their recomputation. Considering the similarity of our goal with the technique of function caching, it seems promising to examine the usefulness of function caching for incremental evaluation.

3.1 Function Caching

Function caching, or memoising, is a method of adding recollection to a program. When the result of a function call is computed, the call and the result are stored in a function cache. A later function call need not be performed if the result is stored in the cache.

Function caching has typically been advocated for uses other than incremental evaluation. Many recursive functions have a pattern of repetitively requesting the value of certain function calls. The classical example of a function whose evaluation can be improved by function caching is the recursive definition of the Fibonacci function, given below. Function caching reduces the time requirements for this function from exponential to linear.

$$fib(n) \equiv \textbf{if } n < 2 \textbf{ then } 1 \textbf{ else } fib(n-1) + fib(n-2) \textbf{ fi}$$

3.2 Incremental Evaluation *via* Function Caching

The best way to get a quick understanding of situations in which function caching gives us incremental computation is to look at some examples. The function *sl* (for *square_list*) in Figure 3.1 yields a list of the squares of a list of numbers. Figure 3.2 shows the results of a sample use of this program with function caching. The center column shows the function calls produced by an initial request for the value of *sl*[1, 2, 3, 4, 5, 6, 7, 8]. After the computation is complete, the left column shows the results of making a request for the value of *sl*[0, 2, 3, ..., 8] – reflecting a change in the first element. The request for *sl*[0, 2, 3, ..., 8] invokes a request for the value of *sl*[2, ..., 8], which is in the cache. Here, function caching provides incremental performance. The right column shows what would happen if, instead, the last element of the list were changed to 9, and a request for the value of *sl*[1, ..., 6, 7, 9] were made. No significant results from the computation of *sl*[1, 2, 3, 4, 5, 6, 7, 8] could be used.

```
sl(L) ≡          { square list }
  if null(L) then L
     else cons(head(L) * head(L), sl(tail(L)))
```

FIGURE 3.1 - Algorithm to produce the squares of the numbers in a list

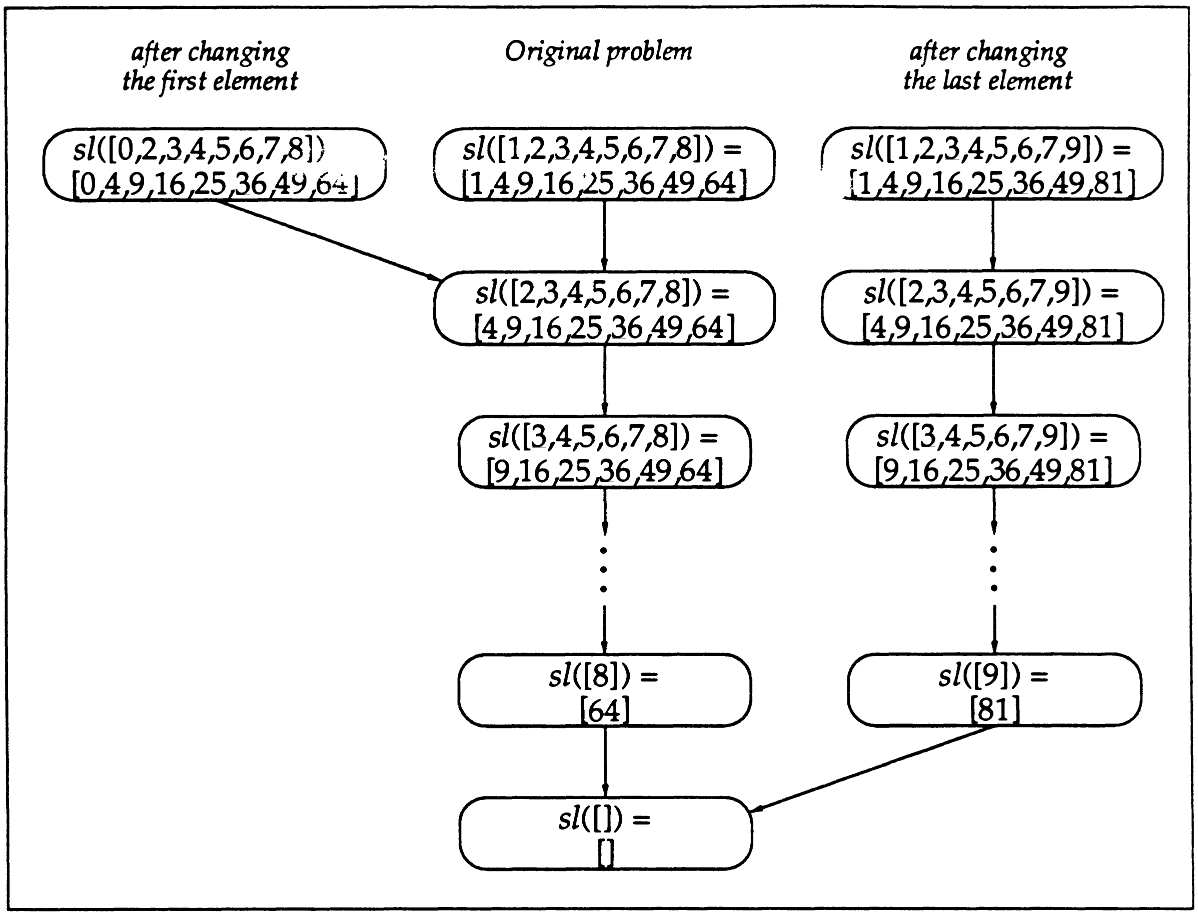

FIGURE 3.2 - Computation resulting from the use of the algorithm in Figure 3.1

Assume we represent sequences in a way that allows us to efficiently divide them in two and we rewrite the algorithm in a divide and conquer style, as in the function *ss* (for *square_sequence*) shown in Figure 3.3. This produces the results shown in Figure 3.4 and Figure 3.5. If function caching imposes only a constant factor overhead and first_half, second_half, and append are constant-time functions, only *O*(log *n*) time will be required to compute the new answer. Note that this example is intended only as an illustrative example; it is simple enough that we would probably use an explicitly incremental algorithm.

What happens if an element is inserted into or deleted from the sequence? Assume the decomposition operators first_half and second_half split the sequence exactly in half if the sequence has an even number of elements and if the sequence has an odd number of elements, the extra element goes into the second half. When an element is inserted at the front of the sequence, this produces the results shown in Figure 3.6; almost the entire computation needs to be redone.

```
ss(S) ≡          { square sequence }
  if length(S) = 1 then list(head(S) * head(S))
      else append(ss(first_half(S)),
            ss(second_half(S)))
```

FIGURE 3.3 - Divide and conquer version of the algorithm in Figure 3.1.

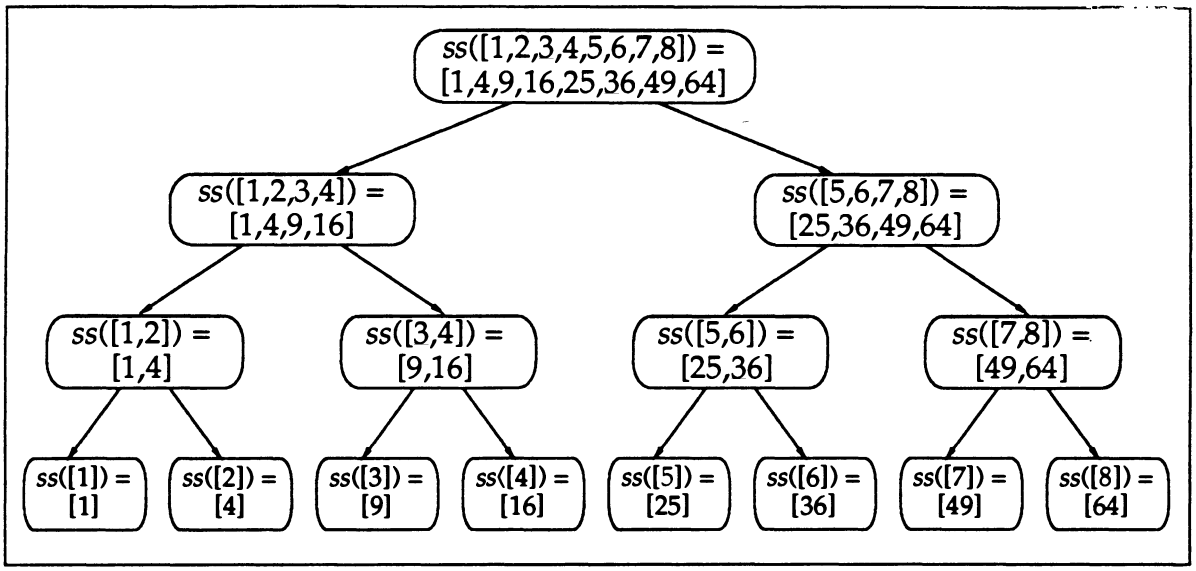

FIGURE 3.4 - Calls arising from a call on ss*[1, 2, 3, 4, 5, 6, 7, 8].*

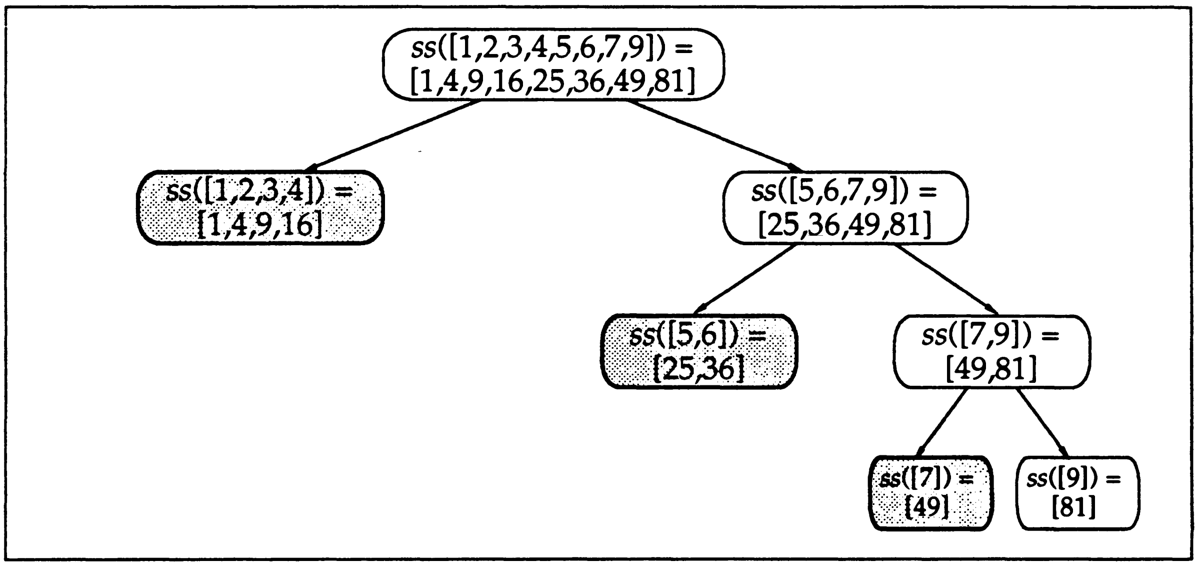

FIGURE 3.5 - Calls arising from a call on ss*[1, 2, 3, 4, 5, 6, 7, 9].*
Entries found in the cache are shown in grey.

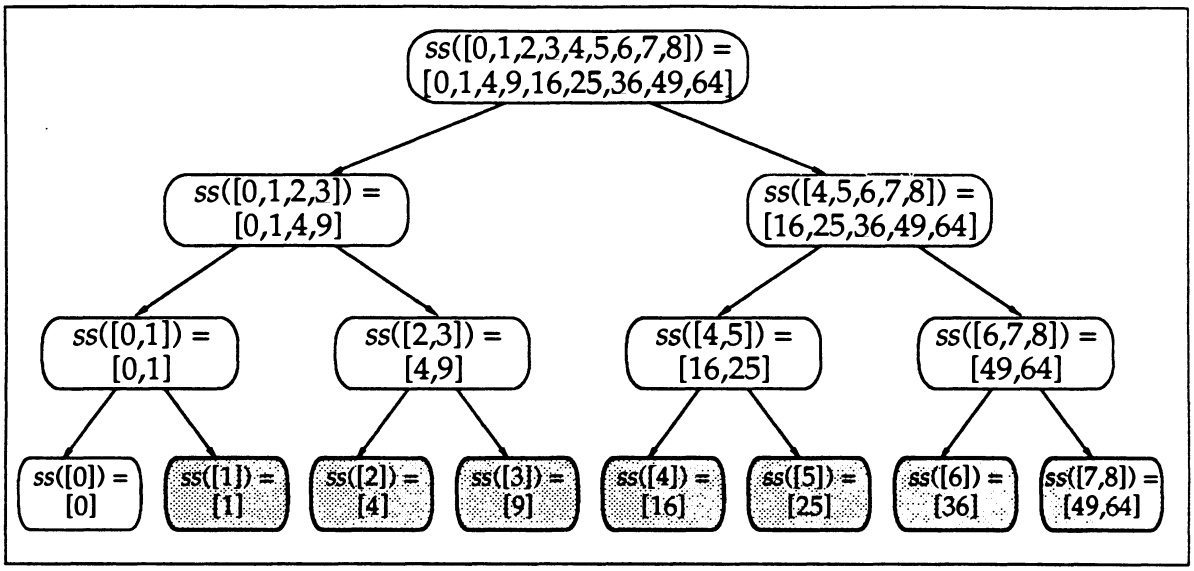

FIGURE 3.6 - Calls arising from a call on ss*[0, 1, 2, 3, 4, 5, 6, 7, 8].*
Entries found in the cache are shown in grey.

We might represent a sequence as a balanced binary tree and use the obvious decomposition rule that splits the sequence represented by a tree T into the sequences represented by the left and right children of T. If T represents the sequence [1, 2, ..., 8], and T' is the tree returned by a function that inserted an 0 at the front of the sequence represented by T, then computations involving T and T' would share common subproblems. However, if T represented the sequence [1, 2, ..., 8], and T' represented the sequence [0, 1, 2, ..., 8], but T' were not derived from T, the decompositions of T and T' might not be similar, and computations involving T and T' might not share any common subproblems.

To exploit function caching, two similar problems must be broken down into sub-problems such that they share many common sub-problems. This explains much of the behavior seen above; divide and conquer algorithms should be used, because they solve a problem by breaking it down into sub-problems, allowing for the possibility of sharing the results to common sub-problems. The scheme shown in Figures 3.2, 3.3 and 3.4 failed to work when elements were inserted or deleted from the sequence because the decomposition scheme did not break the problems into *common* sub-problems. A decomposition scheme that decomposes two similar problems in a way such that they share common sub-problems is called a *stable decomposition*. Whether a decomposition scheme is stable depends on what we consider *similar* problems; our definition of similar problems should operate at the abstract level and not depend on how the problems are represented. Creating data structures and algorithms with a stable decomposition is an interesting problem that is examined in detail and more formally in Chapter 6.

This approach avoids the limitations of incremental dependency graph evaluation discussed in Section 2.6. Since no dependency graph is maintained, there is no need to worry about how to keep it up-to-date. This technique works well for any series of similar problems; the system does not need or care about descriptions of the difference between successive versions of a problem.

4 The Implementation of Function Caching

If we hope to use function caching to provide incremental computation, we must have an efficient implementation of function caching. Some previous implementations of function caching impose large overheads that make function caching only appropriate for combinatorial functions like the Fibonacci function. In this chapter, we describe solutions to a number of problems related to the efficient implementation of function caching. In an experimental implementation using the techniques described in this chapter, we found that function caching imposed about a 50% overhead on the speed of execution in situations where no hits were obtained. In empirical tests of applications such as incremental theorem proving, function caching provide overall real-time speed-ups of 4 to 6.

Function caching is an old idea, but one that few researchers have taken seriously. Part of the reason for this is that many previous function caching implementations were either restricted to or intended for functions that took only atomic values (e.g., integers, real numbers and strings) as arguments. To allow arbitrary S-expressions[1] as arguments to functions requires hash functions and constant-time equality tests for arbitrary S-expressions. In Section 4.1, we discuss hash functions for S-expressions. We describe some of the potential pitfalls that must be avoided, and describe a nearly optimal hash function for S-expressions that only imposes a small, constant-time overhead on the construction of new values. Using a standard equality test is clearly prohibitive — comparing two values of size n can take $O(n)$ time. Hashed consing, a method described in Section 4.2.1, has been used previously to provide constant-time equality tests, but the overhead induced by hashed consing seems prohibitive. Lazy structure sharing, presented in Section 4.2.2, involves much less overhead than hashed consing and in practice provides efficient, amortized constant-time equality tests.

Previous discussions of function caching have often ignored the problem of purging entries from the function cache or required the user to explicitly purge entries from the cache. The analysis of function caching in Section 4.5 suggests guidelines for designing efficient purging algorithms for function caches. Experimental evidence presented in Section 4.5.4 showed an example where our purging algorithm can give as much as a four-to-one real-time speed-up in performance over purging algorithms that have been used previously.

In one of the most recent papers that describes an implementation of function caching [Rob87], the system was implemented using hashing and equality testing methods that both took $O(n)$ time for arguments of size n. Not surprisingly, Robison notes that "for most programs, the extra lookup operations slows down the interpreter, but for certain combinatorial programs the cache can change the program's asymptotic time." The only example Robison gives of a function that is sped up by his implementation of function caching is the Fibonacci function, which is perennially used to justify function caching. The techniques we have developed for hashing and equality testing are much more efficient and, as the benchmarks in Section 4.5.4 show, can

[1] An S-expression is either an atomic value or a pair of S-expressions. We use the notation $<X, Y>$ to denote a pair of values, the *head* of which is X and the *tail* of which is Y. The term S-expression stands for Symbolic expression [McC65].

produce significant real-time speed-ups even in programs not particularly well-suited to function caching.

This chapter discusses the implementation of function caching in an interpreter for a simple functional language that is highly suited for function caching. The functional language we discuss allows no side-effects or non-local references and supports only S-expressions as values. These limitations can be eliminated, although doing so may complicate the function-caching implementation[2].

If a function has a small finite domain, the function can be cached using a dynamic programming technique. For example, assume the domain of a function f is the set 1..10. The function f can be cached by using an array *cache*, indexed 1 to 10, that is defined by saying $cache[i]$ is equal to a predefined value *notCached* if $f(i)$ is not yet cached, otherwise $cache[i] = f(i)$. We then define f', a cached version of f, as

$$f'(i) \equiv \textbf{if } cache[i] = notCached \textbf{ then } cache[i] := f(i) \textbf{ fi; return } cache[i].$$

We often wish to cache functions with infinite domains. Since we cannot allocate an array to hold all possible cache results, some sort of associative memory must be used to implement the cache. Typically, and in this thesis, a hash table is used to implement the cache. Although it would be possible to maintain a separate cache for each function, we describe a method in which a single cache is used to store the cache results for all functions. This is equivalent to simply caching the calls to *apply*, as in Figure 4.1.

cached_apply($f, (x_1, x_2, ..., x_n)$) ≡
  index := hash($f, (x_1, x_2, ..., x_n)$)
  ∀ ⟨ function, arguments, result ⟩ ∈ cache[index] **do**
    if function = f **and** arguments = $(x_1, x_2, ..., x_n)$
      then return(result)
  result := $f(x_1, x_2, ..., x_n)$
  cache[index] := cache[index] ∪ {($f, (x_1, x_2, ..., x_n)$, result)}
  return(result)

FIGURE 4.1 – A version of apply that caches all function calls. This algorithm does not deal with purging items from the cache.

To implement function caching, we need efficient solutions to several problems. We need to be able to

- compute a hash index based on a function name and an argument list,
- determine if a pending function call matches a cache entry, which requires testing very quickly for equality between the arguments in the pending function call and in a candidate match,
- decide which function calls to cache, and
- decide how and when to discard old cache entries.

---

[2] Non-local references can be handled by lambda-lifting (lambda-lifting transforms a function so that non-local references are passed as arguments) or by recognizing when changes to non-local variables invalidate cache results [MC85]. Certain kinds of side effects can also be accommodated [MC85]. Extending the language to handle typed tuples, as in ML, requires only trivial extensions.

Hash functions for atomic values have been discussed extensively in other literature [Knu73]. Section 4.1 discusses hash functions for arbitrary S-expressions.

We need a method of performing a very fast equality test between any two S-expressions. In Section 4.2, we describe two methods for doing this. One method, *hashed consing*, is described by Goto [GK76] and has been used in some LISP implementations. We introduce another method, *lazy structure sharing*, which involves considerably less overhead than hashed consing in practice, although the worst-case performance of lazy structure sharing is less well understood.

In the function caching system we have implemented, all function calls are cached; in Section 4.3, we briefly mention some of the issues involved in caching only a portion of the function calls made.

Sections 4.4 and 4.5 discuss when and how to decide which entries to purge from the function cache. Two methods are described, one that purges "obsolete" entries from the cache and another that uses a predictive purging strategy.

Readers not particularly interested in the details of the implementation of function caching may skip most or all of the rest this chapter. The primary material in the rest of this chapter that is important for understanding later chapters is the fact that we have a constant-time equality test for S-expressions and a method of generating hash keys for S-expressions.

4.1 Hash functions

A hash function is a function mapping elements in a universe of values to a range of integers. Let *hash* be a hash function mapping elements in a universe of values U to $0..m\text{-}1$. Let S be a typical set of input data ($S \subseteq U$). If, for randomly-chosen distinct values x and y in S, Prob$\{$ $hash(x) = hash(y)$ $\}$ $\leq 1/m$, then *hash* is a good hash function (if S is an infinite set, we can do no better than Prob$\{$ $hash(x) = hash(y)$ $\} = 1/m$). This definition is, of course, dependent on what a typical set of input data looks like, which is the reason that devising a hash function that is good for all applications is difficult.

Hash functions for atomic values have been discussed extensively in other work on hashing [Knu73]. What can we use as a hash function for the pair $\langle X, Y \rangle$? Assume that we want a hash function that maps pairs to $0..m\text{-}1$, and we wish to compute the hash key for a pair based on the hash keys of the head and tail of that pair. For any decent hash function *hash* and randomly-chosen S-expressions X and Y, Prob$\{$ $hash(X) = hash(Y)$ $\} = 1/m$. If S is a "typical set of S-expressions", we would like to have the property that for randomly-chosen distinct values x and y in S, Prob$\{$ $hash(x) = hash(y)$ $\} \leq 1/m$. What does this tell us about the hash function? Since we can't precisely define what a "typical set of S-expressions" is, it is difficult to define a "good" hash function for S-expressions; however, we can point out some traps a good hash function should avoid.

Let X_0 and Y_0 be randomly-chosen S-expressions. Let $Z_0, Z_1, \ldots$ be any sequence of S-expressions. Define $X_{i+1} = \langle X_i, Z_i \rangle$ and $Y_{i+1} = \langle Y_i, Z_i \rangle$. We would like to have the property that for all $i \geq 0$, Prob$\{$ $hash(X_i) = hash(Y_i)$ $\} = 1/m$. If $hash(X_0) = hash(Y_0)$ then for all $i \geq 0$, $hash(X_i) = hash(Y_i)$. Therefore, to obtain Prob$\{$ $hash(X_i) = hash(Y_i)$ $\} = 1/m$, we must have $hash(X_0) \neq hash(Y_0)$ implies that for all $i \geq 0$, $hash(X_i) \neq hash(Y_i)$. This gives us $hash(\langle X, Z \rangle) = hash(\langle Y, Z \rangle)$ iff $hash(X) = hash(Y)$. Since the hash key of an S-expression is based only on the hash keys of the components, we can generalize this to

$$hash(Z) = hash(Z') \Rightarrow (\, hash(\langle X, Z \rangle) = hash(\langle Y, Z' \rangle) \text{ iff } hash(X) = hash(Y)).$$

By making a similar argument, we also get

$$hash(Z) = hash(Z') \Rightarrow (hash(\langle Z, X \rangle) = hash(\langle Z', Y \rangle) \text{ iff } hash(X) = hash(Y)).$$

Another feature we want is that if X, X', Y and Y' have been chosen randomly, Prob{ $hash(\langle X, Y \rangle) = hash(\langle X', Y' \rangle)$ } = $1/m$. This is fairly easy to verify and obtain. If X, X', Y and Y' have been randomly chosen, then the hash keys for X, X', Y and Y' are random integers in the range $0..m$-1.

One obvious hash functions that has all of the features discussed so far is to use $hash(\langle X, Y \rangle) = (hash(X) + hash(Y) \bmod m)$. This hash function works well when values are chosen randomly. Of course, values are not chosen randomly, and that could cause problems. The problem with this hash function is that for any two values X and Y, $hash(\langle X, Y \rangle) = hash(\langle Y, X \rangle)$. This situation may or may not occur in practice; since we cannot predict how often this will occur, we would prefer a hash function that is not subject to such abuse.

Knott [Knu73] has suggested avoiding this problem by using a cyclic shift function (either left or right) on the hash key of one of the components before performing the addition – $hash(\langle X, Y \rangle) = (circularShift(hash(X)) \text{ xor } hash(Y))$ (assuming m is a power of 2). The problem with this hash function is that for twin pairs, this hash function generates only half of the possible hash keys and therefore for randomly chosen values X and Y, Prob{ $hash(\langle X, X \rangle) = hash(\langle Y, Y \rangle)$ } = $2/m$.

A hash function that has neither of these problems is $hash(\langle X, Y \rangle) = (2\ hash(X) + hash(Y) \bmod m)$. If m is relatively prime to 6, this seems to work for pairs of atomic values. However, this function has the problem that for any values W, X, Y and Z, $hash(\langle \langle W, X \rangle, \langle Y, Z \rangle \rangle) = hash(\langle \langle W, Y \rangle, \langle X, Z \rangle \rangle)$.

If the mountain won't come to Mohammed, ...

Things would be a lot simpler if we only needed to worry about hash functions for random values. However, we need hash functions that work for non-random values. Since we cannot hope for random values, we will therefore use a random hash function. Define the array *shuffle* to be a random permutation of the integers $0..m$-1. We will use $hash(\langle X, Y \rangle) = (2\ shuffle[hash(X)] + shuffle[hash(Y)] \bmod m)$, for m relatively prime to 6, as our hashing function[3].

This sounds like it should work, but that does not constitute a formal proof that it works. We have verified that there are no simple patterns that cause this hash function to produce an excessive number of collisions, and that it seems to work very well in practice – for almost any distinct two values, Prob{ $hash(X) = hash(Y)$ } = $1/m$. There are, in fact, some patterns that allow the creation of distinct values X and Y such that Prob{ $hash(X) = hash(Y)$ } $\geq 1/m$. One of these patterns in described below. However, the pattern described below is contrived and unlikely to cause many problems in practice. Although some hash function might avoid problems with the pattern described below, there are many similar patterns. We doubt any hash function could work well on all of these patterns.

Define $f(X)$ to be some S-expression constructed from X (e.g., $f(X) = \langle X, X \rangle$). Let $hash_f$ denote the function such that $hash_f(hash(X)) = hash(f(X))$. If $hash_f$ is not a one-to-one function then for randomly chosen values X and Y, Prob{ $hash(f(X)) = hash(f(Y))$ } > $1/m$.

For the hash function we have specified there are some simple patterns that produce this behavior. If $f(X) = \langle X, X \rangle$, $hash_f$ is one-to-one. But for more complicated patterns such as $f(X) = \langle X, \langle X,X \rangle \rangle$, $hash_f$ is not one-to-one. The problem is that having the property that for all simple functions f the function $hash_f$ is one-to-one is a very specific property, one that is unlikely to occur

[3] This can be implemented slightly more efficiently by defining $shash(Z) = shuffle[hash(Z)]$, or $shash(\langle X, Y \rangle) = shuffle[(2\ shash(X) + shash(Y) \bmod m)]$.

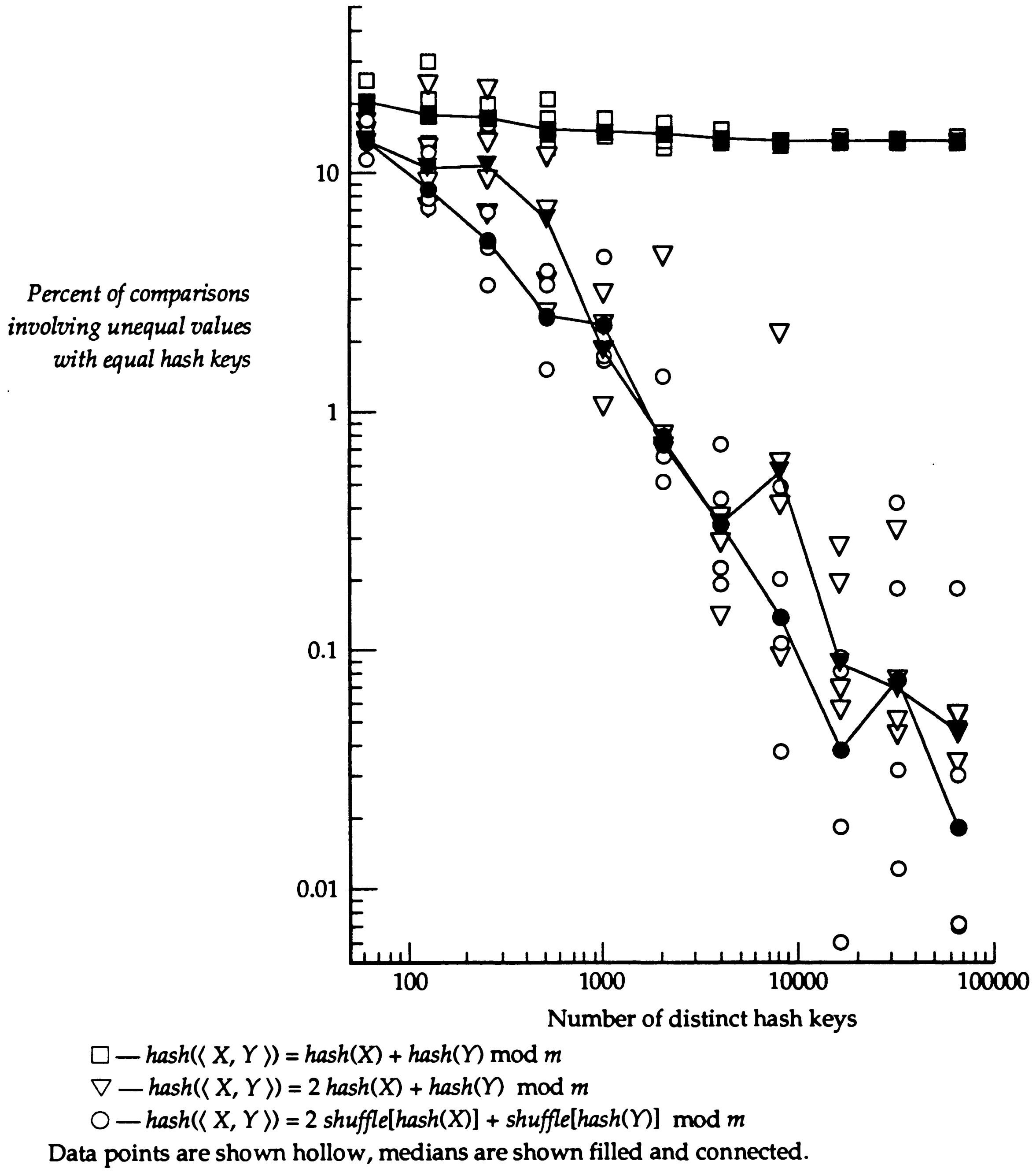

FIGURE 4.2 - Graph of performance of different hash functions

is we use a random hashing function. There are simple hash function which hash this property. Let $hash(\langle X, Y \rangle) = (hash(X) + hash(Y) \bmod p)$, where p is prime. For this hash function, $hash_f$ is one-to-one for all functions f such that $|f(X)| \leq p\ |X|$. But as we have noted, this hash function has other problems.

It seems difficult to completely balance the desires for a hash function that has certain very specific properties and works as well on non-random values as on random values. Such hash functions for S-expressions may exist, but we have been unable to discover them. However, our examinations, both theoretical and empirical, have convinced us that our hash function works almost optimally for practical use. The graphs in Figure 4.2 show empirical results on the use of 3 different hashing functions in an large program and different random seeds (hashing functions for atoms are always shuffled). For this application, using $hash(\langle X, Y \rangle) = hash(X) + hash(Y) \bmod m$ was a disaster and $hash(\langle X, Y \rangle) = 2\ hash(X) + hash(Y) \bmod m$ worked fairly well, although not as well as the randomized version. Using a randomized hash function helps to avoid disastrous failure.

Computing hash keys for function calls

We can compute a hash key for the function call $f(x_1, x_2, ..., x_n)$ just as we would compute a hash index for the value $\langle x_n, \langle x_{n-1}, ..., \langle x_1, f \rangle ... \rangle \rangle$. This same technique could be used to generate hash keys for typed tuples.

4.2 Equality tests for non-atomic values

To be able to perform efficient function caching, we must be able to test equality quickly. In LISP, two predicates are defined: EQ and EQUAL. EQUAL(x, y) is true iff x and y are equal values. EQ(x, y) is true iff x and y either are equal atoms or are the same instance of a non-atomic value (i.e., if x and y are non-atomic values, EQ(x, y) is true iff both x and y point to the same data structure). EQ has the nice feature of requiring only constant time (assuming the comparison of two atomic values requires constant time). In contrast, comparing two values x and y using EQUAL can require time proportional to min($|x|$, $|y|$). The overhead of using EQUAL for equality testing in a function caching scheme would almost certainly cancel any benefit gained from function caching. Hughes [Hug85] suggests a function caching implementation that uses the EQ predicate for equality testing. Assume x and y are two equal but duplicate values (i.e., (EQUAL(x, y) is true but EQ(x, y) is not). The function cache will regard the computations $f(x)$ and $f(y)$ are two distinct computations that can not be shared. This degrades the performance of the function cache by an amount that depends on the application. We prefer to use a method of equality testing that is almost as fast as EQ and yet tests true equality. This section describes and contrasts two such methods: hashed consing and lazy structure sharing.

4.2.1 Hashed consing

Hashed consing is a technique that has been used in LISP implementations to store cons cells uniquely and provide constant-time equality tests [Got74] [All78] [SL78]. In LISP, cons cells are created only by function *cons*. When hashed consing is used, CONS is modified so that it never allocates a new cons cell that would be equal to an already existing cons cell – instead, a pointer to that already existing cons cell is returned. Hashed consing can be obtain by using an algorithm such as the one described in Figure 4.3, or by simply caching all calls to *cons* and never discarding any cache entries. When hashed consing is used, EQ tests true equality (i.e., EQ and EQUAL are the same predicate). Thus, hashed consing allows constant-time equality tests.

```
hashed_cons(H, T) ≡
  s := hash(( H, T ))
  i := s mod number_of_buckets
  ∀ c ∈ hash_bucket[i] do
      If EQ(head(c), H) and EQ(tail(c), T) then return(c)
  c := allocate_cons_cell()
  head(c) := H
  tail(c) := T
  hash(c) := s
  hash_bucket[i] := hash_bucket[i] ∪ {c}
  return(c)
```

FIGURE 4.3 – Description of an algorithm for hashed consing.

Hashed consing cannot be used in a system in which destructive updates of cons cells occur; such updates would have unintended side effects. Hashed consing also complicates garbage collection. A cons cell *c* should be collected if the only pointers to *c* are from the hash table.

Hash keys and hashed consing

If hashed consing is used, we can use a method for assigning hash keys to non-atomic values that is simpler than the methods described in Section 4.1. Whenever a new non-atomic value is allocated, a random number generator is consulted and the random number generated is used as the hash key of the newly created value. This has all the properties we need and is conceptually simpler.

4.2.2 Lazy structure sharing

Hashed consing has nice theoretical properties and it is easy to show that it works well. However, it involves significant overhead. Each time a new value is created, a check must be made to see if an equal one already exists. Maintaining the hash table of all non-atomic values incurs substantial space overhead. Lazy structure sharing is a practical alternative that achieves the same goals as hashed consing, but with lower overhead.

Hash keys allow fast inequality tests, as in Figure 4.4. With this algorithm, the comparison of two unequal values almost always takes constant time and testing two equal but duplicate values of size n for equality takes $O(n)$ time. We can't avoid this, but can make this acceptable by performing *lazy structure sharing*: as a side effect of comparing two equal but distinct values, we update them so that they share structure. A second comparison of those two values will therefore take constant time. An equality test that performs lazy structure sharing is shown in Figure 4.5. Figures 4.6 and 4.7 show the effect of comparing two equal but duplicate values using lazy structure sharing.

```
EQUAL(x, y) ≡
  If hash(x) ≠ hash(y) then    {fast inequality test}
      return(false)
  If EQ(x, y) then
      return(true)
  return(EQUAL(head(x), head(y))
          and EQUAL(tail(x), tail(y)))
```

FIGURE 4.4 – Use of hash keys to obtain fast inequality tests.

```
EQUAL(x, y) ≡
  If hash(x) ≠ hash(y) then   {fast inequality test}
      return(false)
  If EQ(x, y) then
      return(true)

  If not EQUAL(head(x), head(y))
      then return(false)
  {make x and y share heads}
  If prefer(head(x), head(y))
      then head(y) := head(x)
      else head(x) := head(y)

  If not EQUAL(tail(y), tail(x))
      then return(false)
  {make x and y share tails}
  If prefer(tail(x), tail(y))
      then tail(y) := tail(x)
      else tail(x) := tail(y)

  return(true)
```

FIGURE 4.5 – The addition of lazy structure sharing to the algorithm in Figure 4.4.

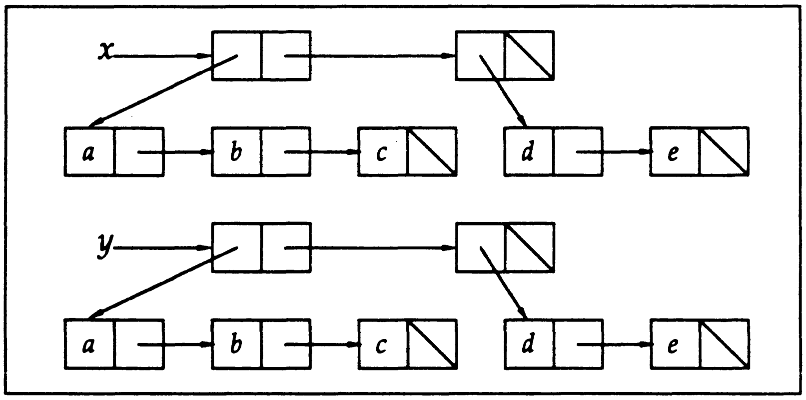

FIGURE 4.6 – Two equal but duplicate values, x and y, before being compared for equality.

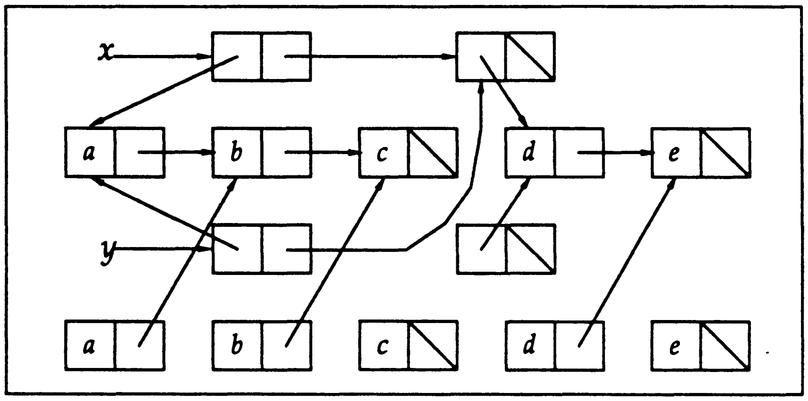

FIGURE 4.7 – The result of testing x and y in Figure 4.6 for equality using lazy structure sharing. It has been assumed that the values pointed to by x are preferable to the ones in y (have more tenure, larger reference counts, ...).

When doing lazy structure sharing, we need some way of determining which of two values is preferred. The preferred value is the one that is expected to be the more widely used of the two values. If reference counting is used, the value with the higher reference count is the preferred one. If generation scavenging is used, the value with the greater tenure is preferred. If no other criterion is available, we can prefer the value stored at the lower address, which will tend to force values at low memory addresses to be widely used. We could simply prefer the first argument, but the worst-case performance with this method is worse than that obtained when simply choosing the value stored at the lower address. It is also possible to introduce some randomness into the decision of which value to prefer. This will make it impossible for any sequence of operations to consistently produce worst-case performance, while not significantly affecting the average performance of lazy structure sharing.

Lazy structure sharing has the same problems with destructive updates as hashed consing does. Garbage collection is not effected by lazy structure sharing.

Average case and worst-case performance of lazy structure sharing

It is hard to characterize the average performance of lazy structure sharing because it depends on the sequence of operations. Performance also depends on the method used for deciding which of two values is "preferred" – reference counting seems to give the best performance and primitive methods such as preferring the value stored at the lower address are subject to poor worst case performance. With a random sequence of operations, lazy structure sharing seems almost always to perform nearly optimally, with a sequence of n comparisons and *cons* operations taking $O(n)$ time.

The worst-case behavior of lazy structure sharing is never worse than using standard equality testing (except for small constant-factor overheads) and it is almost always much better.

Number of distinct hash keys

We do not need distinct hash keys for each S-expression. We only require that a very low percentage of comparisons involve two unequal values with equal hash keys. The percentage of "false positives" depends almost entirely on the number of distinct hash keys and largely independent of the number of values. If the false positive rate is substantially less than 1%, false positives will have no significant impact on the performance of the comparison operation. The results in Figure 4.2 give some suggestions about the number of distinct hash keys needed.

For function caching, the hash keys must be large enough to cover the hash table representing the function cache uniformly. This is satisfied if the number of distinct hash keys is at least equal to the number of cache entries.

4.2.3 Comparing hashed consing and lazy structure sharing

The advantages of hashed consing are:

- Memory is never wasted for duplicate structures.
- Equality tests can always be performed by pointer equality.

The disadvantages of hashed consing are:

- The overhead of checking the hash table each time a new non-atomic value is created.
- The memory overhead required to maintain the hash table (at least one, perhaps two extra pointers for each non-atomic value).

The advantages of lazy structure sharing are:

- It almost always takes constant time to compare two unequal values.
- Two equal values are never exhaustively compared more than once.
- No additional storage is required.

The disadvantages of lazy structure sharing (compared to hashed consing) are:

- Duplicate copies of a value can exist, consuming excess storage
- The theoretical properties of lazy structure sharing are not well understood.

Hashed consing should be used if duplicate instances of most values would be created. In situations where the majority of values will be unique, lazy structure sharing should be used. Hashed consing has almost never been used because of the overhead associated with it and because in most applications the majority of values created are unique. In an experimental version of the Synthesizer Generator, lazy structure sharing led to overall speed improvements, even without function caching or any of the other methods discussed in this thesis.

4.3 Deciding which functions to cache

Each time the result of a function call is computed, that result can be stored in the function cache. The simplest option is always to store the result. Another option is to cache only calls to certain functions, having decided that these function are the functions that will be most profitable to cache. The functions to be cached could be indicated by the user or determined by compile-time analysis [MC85].

4.4 Obsolescence-based purging schemes

A cache entry is considered obsolete if that function call would not be made during an evaluation from scratch of the function calls currently of interest. There may exist non-obsolete cache entries that have not been used in recent evaluations. This can happen if all the functions that would lead to their being called are also in the cache.

We can purge obsolete entries using a scheme similar to that used in garbage collection. Each cache entry has a reference counter, and pointers to the cache entries for calls that are made directly from that function. The reference counter of a cache entry contains the number of cache entries that point to that cache entry. In addition, a list is kept of the top-level function calls that are considered of interest. Typically, this list might contain only the single call to re-evaluate the program on the input data. This leads to the sequence of actions shown in Figure 4.8 in response to a change in the input data.

```
Waiting for user action, list of active function calls = [f(x)].
User specifies edit operation that produces x' from x.
Evaluate f(x'),
      setting the reference count of any newly stored cache entry to 1, and
      incrementing the reference count of any cache entry that is hit.
New list of active function calls = [f(x')], new list of obsolete function calls = [f(x)].
Update display.
Collect obsolete entries while the user thinks about the updated display:
      Decrement the reference count of items on obsolete function call list;
      Whenever the reference count of a cache entry c becomes zero
            Decrement the reference count of the cache entries referenced by c
            Free c
```

FIGURE 4.8 – Steps involved in purging obsolete entries

Note that collection of obsolete entries cannot start until after evaluation of the new active function calls has stopped. Cache entries that are no longer used in one part of the computation may be used in some part that has not yet been reached. In the example above, x' might be such that the computation of $f(x')$ involves the computation of $f(x)$.

4.4.1 Destructive evaluation

The description above assumes that a pointer is kept from every cache entry c to each cache entry that would be used in evaluating c. Although this will work, the overhead is bothersome. To purge a cache entry c using destructive evaluation, the following steps are performed. First, the cache entry c is removed from the cache. Then, the function call associated with c is evaluated in destructive evaluation mode. Since c has already been purged, this will cause the evaluation of the function call to occur. During destructive evaluation, if a hit is made on any cache entry, the reference count of that cache entry is *decremented* (not incremented). If this causes the reference count of the cache entry to go to zero, the cache entry is immediately purged (using destructive evaluation). Destructive evaluation has the same effect as using pointers to link cache entries and is simply a method of trading time for space.

4.5 Predictive-based purging schemes

Previous discussions of function caching have generally relied on the user to purge items from the function cache or have proposed a strategy such as *least-recently-used*, without any analysis of the appropriateness of that strategy. This section describes a formal model that allows us to describe the potential of a function cache and use that model to develop a practical cache replacement strategy that performs substantially better than currently used strategies. In benchmarks of an incremental theorem prover, our caching strategy produces almost a factor of four improvement in running time over a system running without function caching and almost a factor of two improvement in running time over a system using a standard cache replacement strategy.

4.5.1 Comparison to paging

Deciding which element to purge from a function cache has some similarities to deciding which element to purge from a disk or memory cache. Two basic differences limit the applicability of disk and memory caching schemes. The cost to re-compute an entry not in the function cache depends both on the inherent complexity of the function call and on the other contents of the cache. The frequency of use of an entry in the function cache depends on what else is in the cache.

4.5.2 Computation graphs

This section describes a model of computation that we will use to discuss function caching. Each unique function call (a function and the arguments to a call) is represented as a vertex in an infinite computation graph. Each vertex v has an edge to all other vertices that represent function calls made from the function call represented by v. If, during the computation of (the function call associated with the vertex) v we call u twice, there will be two edges from v to u. When we refer to a unique function call, we often just refer to the vertex associated with it. Note that a cycle in this graph would represent a non-terminating computation; we consider only terminating function calls. This graph is different from a standard call graph; each vertex in a computation graph represents a call to a function with a single set of arguments. Figure 4.9 shows a partial computation graph for a symbolic derivative function; since only one function is used, vertices are labeled only by the arguments to the function call.

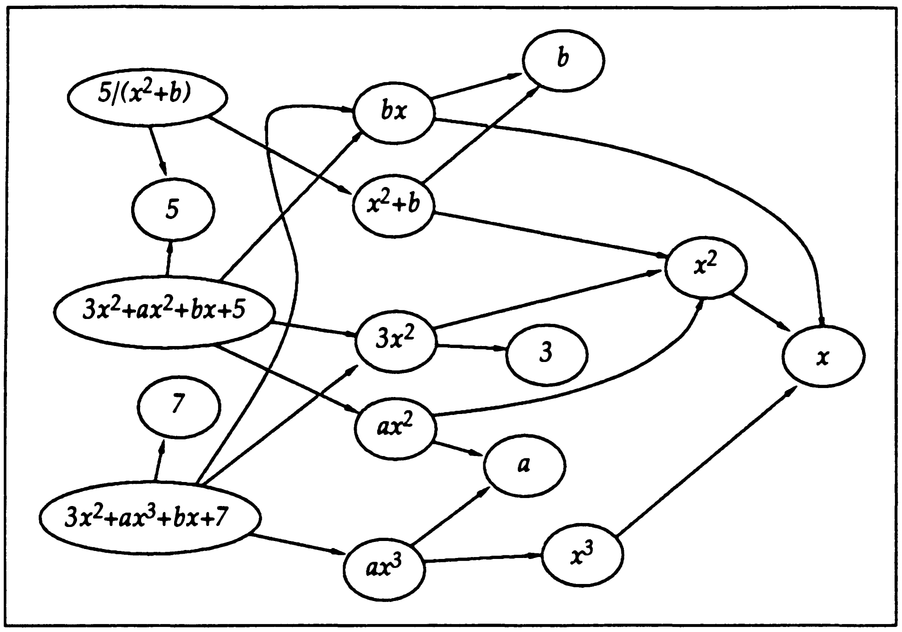

FIGURE 4.9 – Partial computation graph for a symbolic derivative function

Associated with each vertex v is a cost, $cost(v)$, that is defined to be the time required to compute that function, excluding the cost of any called functions but including the cost to see if any of those calls are in the function cache. We define $succ(v)$ to be a bag such that each value u occurs in $succ(v)$ once for each distinct (possibly non-disjoint) path from v to u. For example, for the graph shown in Figure 4.8, $succ(3x^2+ax^2+bx+5) = \{3x^2, ax^2, bx, 5, 3, x^2, a, x^2, b, x, x, x\}$.

We proceed to define a number of terms. The calculations for these terms are shown in Figure 4.10. We define $totalCost(v)$ to be the total cost to compute v, assuming that no function caching is performed. Assume that some constant subset C of all vertices is cached. Define $cachedSucc(v, C)$ to be a bag such that for each u in C, u occurs in $cachedSucc(v, C)$ once for each path from v to u that does not pass through a vertex in C. The cost to compute v, assuming the function cache C is in place, is $recomputationCost(v, C)$.

$$
\begin{aligned}
totalCost(v) &= cost(v) + \sum_{u \in succ(v)} cost(u) \\
recomputationCost(v, C) &= totalCost(v) - \sum_{u \in cachedSucc(v, C)} totalCost(u) \\
totalFreq(v, S) &= sponFreq(v, S) + \sum_{u \in pred(v)} sponFreq(u, S) \\
recomputationFreq(v, C, S) &= totalFreq(v, S) - \sum_{u \in cachedPred(v, C)} totalFreq(u, S) \\
potential(C, S) &= \sum_{v \in C} recomputationFreq(v, C, S) \times totalCost(v) \\
&= \sum_{v \in C} totalFreq(v, S) \times recomputationCost(v, C) \\
potential(C, S) - potential(C - \{v\}, S) &= recomputationFreq(v, C, S) \times recomputationCost(v, C)
\end{aligned}
$$

FIGURE 4.10 – Equations used in defining the potential of a cache

Let S be a series of spontaneous function calls – function calls that arise because they are requested by an external agent, not because they are needed to compute the value of some other function call. For each vertex v, let *sponFreq(v, S)* be the number of times that v occurs in S. We assume that S does not have any locality of reference, but simply reflects the different probabilities of different function calls. We define *pred(v)* similarly to *succ(v)*: *pred(v)* is a bag such that u occurs in *pred(v)* once for each path from u to v. We can calculate *totalFreq(v, S)* as the total number of times v would be called while evaluating the spontaneous calls in S (assuming no function caching was performed).

We define *cachedPred(v)* similarly to *cachedSucc(v, C)*: *cachedPred(v, C)* is a bag such that if u is cached, then u occurs in *cachedPred(v, C)* once for each path from u to v that does not pass through a vertex in C. We define *recomputationFreq(v, C, S)* to be the number of times we will request the value of a vertex v during the evaluation of S.

The potential of a set of cached vertices, *potential(C, S)*, is the amount of time that would be saved during execution of S by running with that cache in place, as opposed to running without any cache.

If a vertex v is a cached vertex, eliminating it from the cache will increase the recomputation frequency of vertices in *cachedSucc(v)* and the recomputation cost of vertices in *cachedPred(v)*. As a result, the additional cost to compute S without v in the cache can be calculated as shown in Figure 4.10.

4.5.3 A greedy, off-line algorithm

We can use the model presented above to devise an algorithm for maintaining a function cache. The situation we are interested in is one where the probability of a spontaneous request for the value of a particular function call has approximately the same probability in the immediate future as in the more distant future. The algorithm discussed here is mainly of theoretical interest because it assumes that we know what the sequence of future calls will be and don't care about overhead. In Section 4.5.4 we devise a practical algorithm motivated by this discussion.

We maintain a graph consisting of all cached vertices, with pointers from each vertex v to *cachedSucc(v)* and *cachedPred(v)*. With each vertex v we can calculate and store *totalCost(v)* and *recomputationCost(v)*. If we know *totalFreq(v, S)*, we can compute *recomputationFreq(v, S)* and maintain a priority queue of cache entries, indexed by *recomputationFreq(v, S)* x *recomputationCost(v, S)*. When we need to purge an entry from the function cache, we take the element v from the priority queue with the smallest index, eliminate v from the cache and update the *recomputationCost* of vertices in *cachedPred(v)* and the *recomputationFreq* of vertices in *cachedSucc(v)*.

Although this algorithm will chose the best entry to eliminate, it is not optimal for eliminating multiple vertices. For example, consider the graphs in Figure 4.11. The solid circles represent cached entries, grey circles uncached entries. The cost of each vertex is 1, and we know that in the future there will be a spontaneous call to each of the four vertices at the top. Figure 4.11*a* represents the graph when all entries are cached. After purging three entries, Figure 4.11*b* results; we can still request all four of the top entries at no cost. Purging three more entries yields Figure 4.11*c*, and all four spontaneous requests can be processed for a total cost of 9. Figure 4.11*d* shows an arrangement with a single entry cached that allows us to process all four requests for a total cost of 8. Despite this limitation, the greedy algorithm almost always does reasonably well, the decisions it makes are simple, and it is easy to analyze. We note as an open problem the development of an optimal algorithm for eliminating multiple cache entries.

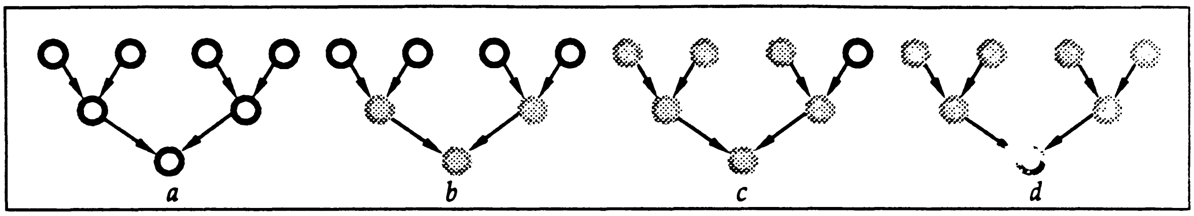

FIGURE 4.11 – an example showing non-optimal performance for eliminating multiple vertices

Case study

In this section we examine the problem discussed by Hilden [Hil76]:

$$
\begin{aligned}
&A(s, n, m) \equiv \\
&\quad \textbf{if } s < v \vee s > \mu \textbf{ then } 0 \\
&\quad \textbf{else if } s = v \vee s = \mu \textbf{ then } 1 \\
&\quad \textbf{else } A(s\text{-}n\text{-}m, n\text{-}1, m) + A(s, n, m\text{-}1), \\
&\quad \textbf{where } v \equiv n(n+1)/2, \mu \equiv n(n+1)/2 + nm.
\end{aligned}
$$

Hilden compares the results of a number of different cache strategies and cache sizes on the time required to compute *A(85, 9, 9)*. He suggests two general-purpose purging algorithms: *Latest* and *Latest-Second*. *Latest* is recommended for over-crowded cache tables, *Latest-Second* for less crowded cache tables. Although the overhead and requirements for knowledge of the future prevent our greedy algorithm described above from being practical, the graphs in Figure 4.12 compare it with Hilden's algorithm, as well as with a *least-recently-used* algorithm. The *Latest* policy uses a hash table indexed by (*s n m* mod *cacheSize*), with collisions resolved by overwriting; in *Latest-Second*, if a hash location is occupied and the following location is not, it is placed there instead. In Hilden's algorithms, the cache size is a prime number to improve the hash function.

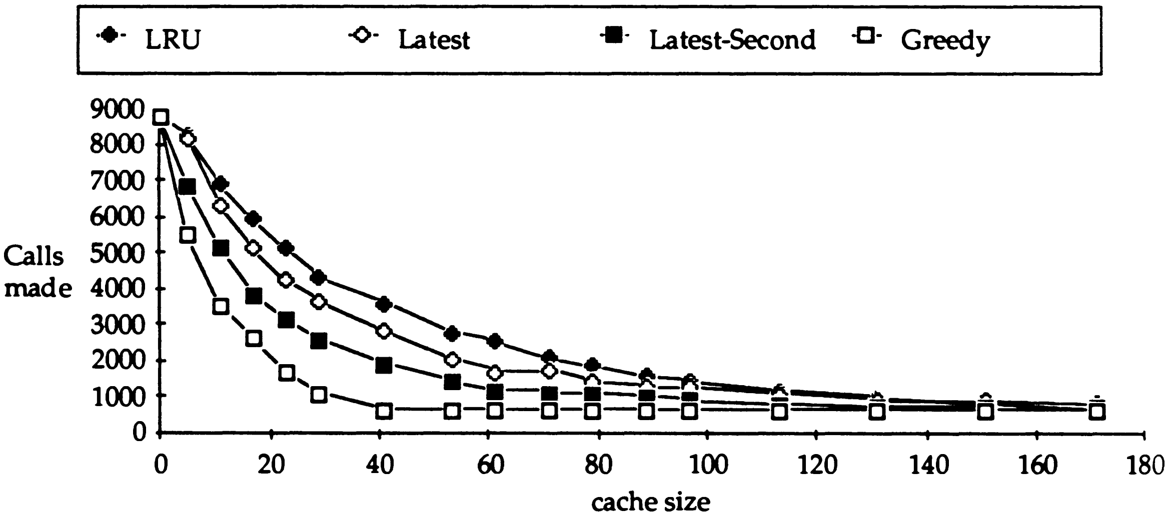

FIGURE 4.12 – Number of calls required to compute A*(85, 9, 9) with different cache sizes and policies*

4.5.4 A practical algorithm

Since many of the functions we may wish to cache have infinite domains, we will cache function calls using a hashing mechanism. Given a function and a set of arguments, we calculate a hash value that is used as an index into the function cache. We compare the function and the arguments to those already stored in the hash bucket and, if we find a matching set, return the result stored there.

The method suggested by Hilden and used in the Illinois FP interpreter is based on this method. In his scheme, each bucket can contain only one entry, and, each time a function result is computed, it is stored in the associated hash bucket, overwriting any previous entry.

In order to produce a practical algorithm, we assumed the following limitations:

- Only fixed, statically allocated memory for each cache entry.
- No operation is allowed to examine more than a small constant number of cache entries.
- We have no knowledge of the future behavior of the program and can only make predictions from past activity.

In the method we propose, each bucket can store up to a contain number of entries. When searching for a call in the cache, the algorithm determines the appropriate bucket and searches through all entries in it.

With each entry, we store the number of hits and the estimated amount of time required to recompute it. The entries in a bucket are kept sorted according to their estimated potential, where the potential of an entry *e* is estimated as *estimatedRecompFreq(e)* x *estimatedRecompCost(e)*. In our implementation we have chosen to calculate the estimated recomputation frequency of an entry based on the number of hits on the entry. The estimated recomputation cost of an entry is the time that was required to compute it (with one twist, explained below).

To store a new entry in the cache, the algorithm finds the appropriate bucket and randomly selects an entry to replace, using a probability distribution that is highly skewed towards replacing a lower ranked entry. When a new entry is stored in the cache or a cache entry is hit (which increments the hit count for that entry) the bucket is resorted.

The cost to recompute an entry is found by simply having a clock that counts the number of instructions executed and storing the start time for each function in it's activation record. If, after storing a cache entry and resorting the bucket, the entry is in the more valuable half of the bucket, the clock is reset to the time when computation of the function started. This is because the entry just stored is very likely to stay in the cache and therefore the recomputation cost of pending function calls should not include the cost of that call.

Note that this method is a generalization of the method suggested by Hilden. If the number of entries in each bucket is set to one, this algorithm is exactly the same as Hilden's. Within this model, we have experimented with different values of parameters and found that the following parameters work well for the applications we were most interested in:

> *number of entries per bucket* = 8
> *estimatedRecompFreq(e)* = *hits(e)*+0.25
> *index of entry to replace* = the maximum of 8 random numbers chosen in the range 0..7, where 0 indexes the most valuable entry and 7 indexes the least valuable (this calculation is actually done using a table lookup method).

These choices, which are blatantly empirical, were chosen to provide good performance for an incremental theorem prover (benchmarks 1*a* and 1*b*). Some other choices that seemed reasonable were examined, such as choosing the index of the entry to replace as (7 – the number of leading zeros in a random 7-bit binary integer). All led to performance that was better than that of a standard cache replacement strategy, but significantly worse than the performance provided by the choices above.

We do not expect that our choices are definitive for all function caching applications and urge others implementing this algorithm to experiment with other choices for their applications. Also, in some applications, better methods may be available for estimating the recomputation frequency of an entry.

We have some intuitive feelings as to why these choices work well:

We randomly chose which entry to replace because we can only estimate the value of a entry. If we have accidently rated an entry with too high a value, we do not want it to stay in the cache forever. By occasionally throwing out even the highest rated entry in a bucket, we know that no entry will stay around forever unless it is used; since we throw out highly rated entries rarely, we lose very little over always having them in the cache.

We calculate the estimated future frequency of a entry simply based on the number of hits on that entry. The initial evaluation of the function that stored it in the cache is counted as only one quarter of a hit, since many functions are evaluated only once.

Choosing a bucket size of 8 keeps the overhead associated with updating the cache low, while allowing some selectivity.

As mentioned before, resetting the clock after caching a highly rated entry helps keep the algorithm from over-estimating the recomputation cost of entries.

The caching algorithm described above has been implemented in the functional language used by the Synthesizer Generator [RT84], an attribute-grammar-based system for editing objects and maintaining computed information about them. The overhead associated with the function cache is moderate: with a hit rate of zero, the cache slows down the interpreter by about one third. The Synthesizer Generator is a particularly well suited application for a function cache because of its incremental nature. In benchmarks involving the creation of programs annotated to prove their correctness, and the generation and automatic simplification of proof obligations, the function cache provided a real-time speed up of almost four.

Benchmark results

Table 4.1 lists the number of op-codes executed by the interpreter for a number of benchmarks, replacement policies and cache sizes. Figure 4.13 graphs the results for two of these benchmarks. Table 4.2 lists real-time figures for one of the benchmarks. The times listed in Table 4.2 do not include about 15 seconds of time spent in routines associated with the user interface or system time for paging. The time listed is mainly the time spent in the interpreter and in memory management; the time spent calculating hash keys and comparing values is included in the figures listed and is listed separately as well. The benchmarks used were:

Incremental 1a & 1b – The incremental creation and verification of a program. The program is annotated by the user with loop invariants, preconditions and postconditions, and the system calculates a number of proof obligations that need to be proved to show that the program is correct. Each modification to the program changes one or more of these proof obligations and invokes a theorem prover that attempts to simplify the obligation as much as possible. Each time the theorem prover is invoked, it begins with no knowledge of the work done the previous time it was invoked except for what is in the function cache. These two benchmarks reflect the behavior of the system as two different programs were created and verified, involving 20-30 incremental changes to the program and re-invocations of the theorem prover.

Incremental 2 – Reading in a logic library and building a logic proof. This system only verifies proofs; it does not generate them. The performance of this benchmark was nowhere near as good

cache entries	Incremental 1a (s)	(i)	1b (s)	(i)	2 (s)	(i)	Complete 1a (s)	(i)	2 (s)	(i)	A(85,9,9) (s)	(i)
0	1,120	1,120	1,272	1,272	1,184	1,184	171	171	644	644	302	302
16	1,043	930	1,149	682	1,086	1,006	154	152	582	592	219	191
32	1,020	850	1,115	501	1,020	894	152	156	543	544	150	96
64	994	710	1,071	367	910	756	148	151	481	475	85	42
128	957	605	953	318	795	640	145	140	420	405	43	25
256	896	512	884	245	644	557	141	130	373	358	25	19
512	819	435	543	186	527	507	136	129	336	323	20	18
1,024	702	345	220	144	498	466	130	121	311	301		
2,048	430	274	154	123	464	451	120	113	296	290		
4,096	301	222	132	115	453	445	114	109	290	286		
8,192	258	210	125	113	447	445	112	108	287	284		
16,384	244	208	122	112	447	443	111	108	286	284		

(s) – standard algorithm, bucket size = 1 (i) – improved algorithm, bucket size = 8

TABLE 4.1 – Thousands of op-codes executed for different benchmarks, cache replacement algorithms and cache sizes

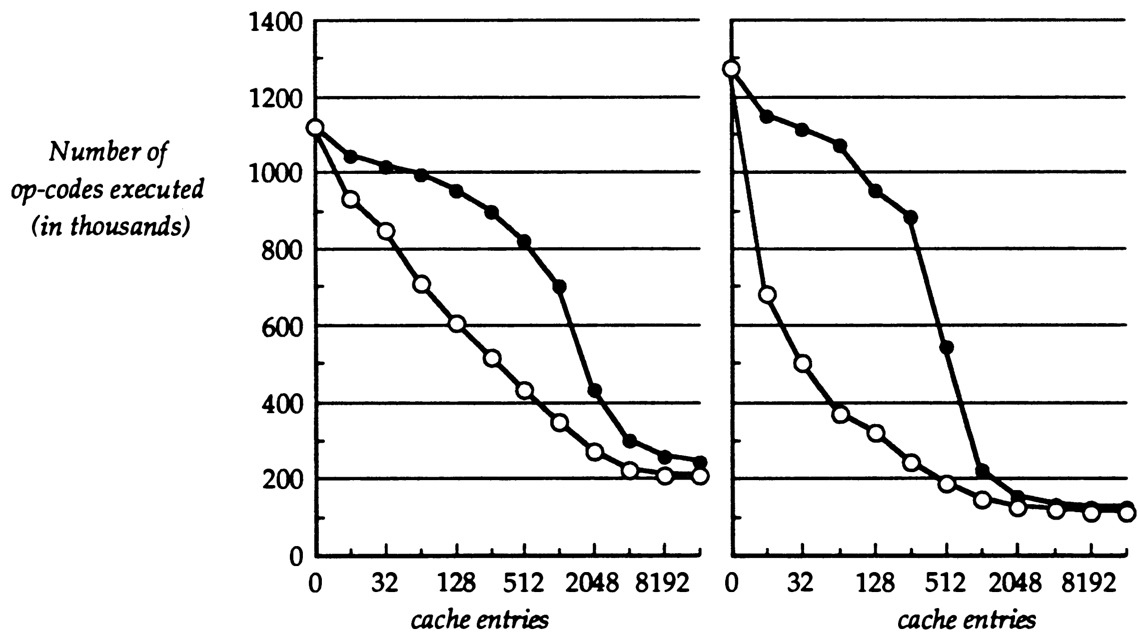

FIGURE 4.13 – Benchmarks Incremental 1a and 1b, comparing the performance of the standard (•) caching algorithm and our improved(○) caching algorithm

Notes	improved algorithm			hits ignored	without caching
Cache entries	16,384	2,048	256	163,84	none
Total time (seconds)	25	34	54	155	94
time hashing and comparing	2.5	3.9	5.6	41	
Op-codes/Second	8,400	8,100	8,000	7,200	12,000

	standard algorithm		
Cache entries	16,384	2,048	256
Total time (seconds)	27	49	96
time hashing and comparing	3.5	6.0	9.5
Op-codes/Second	9,000	8,800	9,300

TABLE 4.2 – Actual running times on a Sun3 for benchmark Incremental 1a

as for the previous benchmarks. There are two reasons for this: the computations involved are many relatively cheap function calls and almost all the functions involved take symbol tables as arguments. This reduces the hit rate of the cache because, to match an entry in the cache, every item in the symbol table must be identical, even if only one item from the table was actually examined by the function. We are pursuing future research to remove this limitation.

Complete 1a & 2 – These involve reading and attributing the files created in benchmarks *Incremental 1a* and 2. Because these benchmarks did not involve a repetitive series of similar problems, there was not enough of a pattern for our improved algorithm to detect and utilize.

A(85, 9, 9) – The function analyzed by Hilden. These results for the standard algorithm do not exactly parallel the ones in Figure 4.12 because of differences in the hash function. The hash function that Hilden used was not suitable for general application.

4.5.5 Related work

The primary result of theoretical studies are the conclusion that the usefulness of a function cache entry depends on both the numbers of hits on that entry and on the amount of time that would be required to recompute that entry. In hindsight this appears obvious, but previous examinations of function caching have assumed that the value of a cache entry depends solely on the likely number of hits on that entry. In the original paper on function caching [Mic68], Michie suggested a *least-recently-used* algorithm for discarding items from the function cache. Hilden [Hil76], as discussed in the body of this paper, considered a number of replacement schemes experimentally for a specific problem. The method he suggested for general use is compared with our algorithm in Sections 4.5.3 and 4.5.4. Friedman, Wise and Wand [FWW76] allow only functions with a finite domain to be cached and do not discuss purging.

In Hughes' [Hug85] discussion of lazy memo-functions, he avoids the cost of comparing entries by only hitting on cache entries whose arguments are identical structures, as opposed to equal structures. With this assumption, he notes that cache entries can be discarded when there are no references to one or more arguments of a cache entry.

Mostow and Cohen [MC85] examine the question of deciding which functions to cache. Much of their work is directed towards determining which functions are safe to cache in the presence of side effects. They discuss how to decide which functions are profitable to cache, but they either decide to cache every call to a function or don't cache the function at all. The work they report could profitably be combined with the work reported here.

Keller and Sleep [KS86] allow the user to specify the number of times a function value will be requested and allow the user to purge specific entries.

Robison [Rob87] reports on the function cache implemented in the Illinois FP interpreter using the replacement scheme we have compared with our method in Section 4.5.4.

The primary result of theoretical studies are the conclusion that the value of a function cache entry depends on both the numbers of hits on that entry and on the amount of time that would be required to recompute that entry. In hindsight this appears obvious, but previous examinations of function caching have assumed that the value of a cache entry depends solely on the likely number of hits on that entry.

5 Randomized Data Structures and Probabilistic Time Bounds

The data structures we present in the Chapters 7 and 8 are randomized (i.e., organized probabilistically). Algorithms that work with randomized data structures have good expected-time bounds and poor worst-case time bounds. In this chapter, we present a unified method for analyzing both the expected performance and the probability distribution of the running times of algorithms.

5.1 Probabilistic upper bounds on random variables

A *random variable* has a fixed but unpredictable value and a predictable average and probability distribution. If X is a random variable, PROB{ $X = x$ } denotes the probability that X equals x and Prob{ $X \leq x$ } denotes the probability that X is at most x. For example, if X is defined as the number obtained by throwing a perfect die, Prob{ $X \leq 3$ } = 1/2. For conciseness, we sometimes refer to a property of the probability distribution of a random variable as a property of the random variable itself.

It is often preferable to find simple upper bounds of values whose exact value is difficult to calculate. In order to discuss upper bounds on random variables, we need to define a partial ordering on the probability distributions of random variables:

Definition 5.1 ($=_{prob}$ *and* $\leq_{prob}$). Let X and Y be random variables. We define equality and a partial ordering on the probability distribution of random variables as follows:

$$X =_{prob} Y \text{ iff } \forall\, x, \text{Prob}\{\, X \leq x \,\} = \text{Prob}\{\, Y \leq x \,\} \text{ and}$$
$$X \leq_{prob} Y \text{ iff } \forall\, x, \text{Prob}\{\, X \leq x \,\} \geq \text{Prob}\{\, Y \leq x \,\}. \square$$

The algorithms we analyze in this thesis have running times that are bounded by a negative binomial distribution.

Definition 5.2 (*negative binomial distributions — NB(s, p)*). Let s be a non-negative integer and p be a probability. The term $NB(s, p)$ denotes a random variable with the *negative binomial distribution* equal to the distribution of the number of failures seen before the s^{th} success in a series of random independent trials where the probability of a success in a trial is p. We define $NB(0, p) \equiv 0$ and $NB(s) \equiv NB(s, 0.5)$. □

Several well-known properties of the negative binomial distribution are [TK84]:

$$\text{average}(NB(s, p)) = s(1-p)/p,$$
$$\text{variance}(NB(s, p)) = s(1-p)/p^2, \text{ and}$$
$$\text{Prob}\{\, NB(s, p) = k \,\} = \binom{k+s-1}{s-1} p^s (1-p)^k.$$

In Theorem A.1 (in the Appendix), we obtain the very loose upper bound

$$\text{Prob}\{\, NB(s, p) > (M-1)s \,\} < \frac{1-p}{p} \, \frac{Mp}{Mp-1} \, (eM \max(p, 1-p)^M)^s.$$

If X and Y are two independent random variables, each bounded by a negative binomial distribution, we can calculate an probabilistic upper bound on $X + Y$:

$$\begin{aligned} & X \leq_{prob} c_1 + NB(s_1, p) \\ & \quad \wedge Y \leq_{prob} c_2 + NB(s_2, p) \\ & \quad \wedge X \text{ is independent of } Y \\ & \Rightarrow X + Y \leq_{prob} c_1 + c_2 + NB(s_1 + s_2, p). \end{aligned}$$

If X and Y are *not* independent, we can still calculate an upper bound on the average of $X + Y$:

$$\begin{aligned} & X \leq_{prob} c_1 + NB(s_1, p) \\ & \quad \wedge Y \leq_{prob} c_2 + NB(s_2, p) \\ & \Rightarrow \text{average}(X + Y) \leq \text{average}(c_1 + c_2 + NB(s_1 + s_2, p)). \end{aligned}$$

5.2 The probabilistic *big-Oh*

Big-Oh notation is defined by saying that $f(n)$ is $O(g(n))$ iff $\exists\, M, n_0$ such that $\forall\, n$ such that $n \geq n_0$, $|f(n)| \leq M\ |g(n)|$. This works fine when f is a non-random function. We introduce a new notation for use when g is a probabilistic function whose result may be unbounded.

Definition 5.3 (O_{prob}). Let $X_0, X_1, \ldots$ be an infinite sequence of random variables and f be a function of n. We define X_n to be $O_{prob}(f(n))$ iff

$$\forall\, \varepsilon \text{ s.t. } \varepsilon > 0, \exists\, M, n_0 \text{ s.t. } \forall\, n \text{ s.t. } n \geq n_0, \text{Prob}\{\ |X_n\ | \leq M\ |f(n)|\ \} \geq 1\text{-}\varepsilon. \square$$

We introduce Theorem 5.1 to allow us to show probabilistic order bounds on algorithms.

Theorem 5.1. Let $X_0, X_1, \ldots$ be an infinite sequence of random variables, and p be a probability greater than 0. If $|X_n\ | \leq_{prob} (\,|f(n)|\, + NB(g(n), p))$ then X_n is $O_{prob}(\,|f(n)|\, + g(n))$.
Proof. If $g(n) = 0$ or $p = 1$, $NB(g(n), p) = 0$ and X_n is $O_{prob}(\,|f(n)|\, + g(n))$. Assume $g(n) \geq 1$ and $p \leq 1$. Given p and ε, find $M \geq 1$ such that

$$\frac{1-p}{p}\ \frac{Mp}{Mp-1}\ (eM \max(p, 1\text{–}p)^M) \leq \varepsilon \text{ (such an } M \text{ must exist).}$$

This implies that

$$\frac{1-p}{p}\ \frac{Mp}{Mp-1}\ (eM \max(p, 1\text{–}p)^M)^{g(n)} \leq \varepsilon.$$

Combined with Theorem A.1 this gives $\text{Prob}\{\ NB(g(n), p) > (M\text{-}1)\, g(n)\ \} \leq \varepsilon$, which implies that for all n such that $n \geq n_0$, $\text{Prob}\{\ |X_n\ | \leq\ |f(n)|\ + (M\text{–}1)\, g(n)\ \} \geq 1\text{–}\varepsilon. \square$

6 Data Structures and Algorithms for Incremental Computation

Chapter 4 discusses how to provide function caching in an interpreter that supports S-expressions as values. Many algorithms are designed to work with more complicated types such as sets or sequences. Rather than extend the interpreter and the function caching system to handle other data types, values in other type systems are represented using S-expressions. This chapter describes the issues involved in representing values and problems in a way such that function caching provides efficient incremental evaluation. Chapters 7 and 8 present specific solutions for sequences and sets motivated by the discussion in this chapter.

6.1 Representation schemes

A representation scheme is used to represent values of an abstract type τ as values of a concrete type σ. Typically, the abstract type τ is a type that we would like to use in an algorithm but is not directly supported by the system in use. For example, if lists are supported by the underlying system but sets are not, we can use a representation scheme in which a set is represented by a list of the elements of the set. In this representation, the concrete list values [6, 3, 'x'] and ['x', 6, 3] would both represent the abstract set value {3, 6, 'x'}.

Definition 6.1 (*representation function*). We can formalize a representation scheme by specifying a representation function that, when applied to a concrete value, returns the abstract value represented by that concrete value. □

Example. If R is a representation function that maps lists to sets, R([6, 3, 'x']) = R([3, 'x', 6]) = {3, 6, 'x'}.

Note that a representation function is not implemented, but is simply used for arguing about the correctness of algorithms designed to work with that representation. If $R_1 : \rho \to \sigma$ and $R_2 : \sigma \to \tau$ are representation functions, $R_2 \circ R_1$ is the representation function giving the value in τ represented by a value in ρ.

Typically, the representation we choose for an abstract type depends on the operations we need to perform and the efficiency with which those operations can be supported by the representation. For incremental computation, we also have to take into consideration the fact that we intend to use function caching and that we wish to obtain efficient incremental evaluation. The effects of these considerations are discussed in Sections 6.2 and 6.3.

6.2 Data structures and algorithms for function caching

Data structures that we plan to use with function caching must be updatable applicatively and should provide unique representations.

Applicative updates. The use of function caching precludes the destructive updating of data structures. Updates must produce new data structures representing the desired values instead of overwriting existing data structures.

Unique representations. Let f be a function on sets that is designed to use a representation scheme in which sets are represented as lists. Let R be the representation function that maps lists to sets. If x

and y are concrete values representing the same set (i.e., $R(x) = R(y)$), we want to be able to reuse a previously computed result $f(x)$ when computing $f(y)$. This will happen iff $x = y$.

This is a general problem for function caching: in order to make function caching happen at the abstract level, we must use representations schemes with *unique representations*. For example, we could use a representation scheme in which a set is represented by a *sorted* list of the elements of the set. A representation function R provides unique representations iff R is a one-to-one function. Note that if R is a one-to-one function, R^{-1} is well-defined.

It would be possible to side-step the requirement of unique representation by expecting our function caching implementation to match only arguments represented the same way. This would decrease the effectiveness of the cache, although in some circumstances the decrease might be small. If it were very difficult to provide data structures that had unique representations, we might wish to take this route. However, Chapters 7 and 8 provide data structures for sequences and sets that have the desired features.

6.3 Data structures and algorithms for incremental evaluation

As discussed in Chapter 3, in order to use function caching to solve quickly a new problem that is similar to a previous problem, the new problem must be broken down into sub-problems in a way such that solving the new problem involves solving sub-problems that were solved for the previous problem. Decomposing problems into sub-problems usually involves decomposing large data structures into smaller ones. We therefore wish to design data structures such that two *similar* values have *similar decompositions*.

Our definition of *similar* values depends on the type of similarities in which we are interested.

Definition 6.2 (*transformation*). A transformation on type τ is a function that maps a value of type τ (and possibly additional arguments) to a value of type τ. Transformations are not implemented; they are used only for discussing the similarity of abstract values. □

Example. One of the transformations on sequences we will be using in later examples is *changeFirst*. If *changeFirst* transforms x into x', then x and x' differ in only their first element. The meanings of other transformations we use is largely self-evident from their names. The *insert*, *delete* and *change* transformations on sequences reflect a change at any location. The *addElement* and *deleteElement* transformations on sets reflect a difference of a single element.

Definition 6.3 (*distance between abstract values — $tDistance_T(x, x')$*). Let T be a set of transformations. We define $tDistance_T(x, x') = k$ iff k is the smallest integer such that there is a sequence of k transformations chosen from T that transforms x into x'. If no sequence of transformations from T transforms x into x', $tDistance_T(x, x')$ is undefined. Depending on T, this distance measurement may or may not be symmetric. □

Example. $tDistance_{\{changeFirst,\ insertBeforeFirst\}}([5, 1], [5, 3, 1]) = 2$ and $tDistance_{\{changeFirst,\ insertBeforeFirst\}}([5, 3, 1], [5, 1])$ is undefined. For sets A and A', $tDistance_{\{addElement,\ deleteElement\}}(A, A') = |(A' - A) \cup (A - A')|$.

Definitions 6.4, 6.5 and 6.6 formalize the idea of *similar decompositions*.

Definition 6.4 (*decomposition scheme*). A *decomposition scheme* on type τ is a 3-tuple of functions $\langle$ *atomic, split, bound* $\rangle$ that have the types and properties shown below.

$$
\begin{aligned}
&atomic : \tau \to \text{Bool},\\
&split : \tau \to \langle \tau, \tau \rangle,\\
&bound : \tau \to \text{Nat},\\
&split \text{ is a one-to-one function,}\\
&\qquad \neg\, atomic(x) \Rightarrow \quad bound(x) > 0 \wedge split(x) \text{ is defined, and}\\
&\quad \langle y, z \rangle = split(x) \Rightarrow \quad bound(y) < bound(x) \wedge bound(z) < bound(x). \square
\end{aligned}
$$

Informally, *atomic*(x) is true if it is impossible to further decompose x, *split*(x) is the pair of values into which x is decomposed and *bound*(x) is an upper bound on the number of times x can be decomposed.

To implement a decomposition scheme D on abstract values, we use a representation scheme R and decomposition scheme $D' = \langle atomic', split', bound' \rangle$ such that D' is an efficient implementation of D for R (e.g., *split'* and *atomic'* are constant-time functions and $split(x) = \langle y, z \rangle$ iff $split'(R^{-1}(x)) = \langle R^{-1}(y), R^{-1}(z) \rangle$).

Example. We can define a decomposition scheme *linkedLists* on sequences as

$$
\begin{aligned}
&atomic(S) \equiv |S| \le 1,\\
&split([x_0, x_1, \ldots, x_{n-1}]) \equiv \langle [x_0], [x_1, \ldots, x_{n-1}] \rangle, \text{ and}\\
&bound(S) \equiv |S|.
\end{aligned}
$$

Definition 6.5 (*decomposition* — $d_D(x)$). The *decomposition* of a value with respect to a decomposition scheme $D = \langle$ *atomic, split, bound* $\rangle$ is the set defined (recursively) by

$$d_D(x) \equiv \textbf{if } atomic(x) \textbf{ then } \{x\} \textbf{ else } \{x\} \cup d_D(y) \cup d_D(z) \textbf{ where } \langle y, z \rangle = split(x) \textbf{ fi}. \square$$

Example. Using the decomposition scheme given in the example for Definition 6.4, $d_{linkedLists}([1, 2, 3]) = \{[1, 2, 3], [1], [2, 3], [2], [3]\}$.

Definition 6.6 (*distance between decompositions* — $dDistance_D(x, x')$). Let x and x' be values of type τ and D be a decomposition scheme on type τ.

$dDistance_D(x, x')$ denotes the distance from the decomposition of x to the decomposition of x' with respect to D: $dDistance_D(x, x') = |\, d_D(x') - d_D(x) \,|$. This distance measurement is not symmetric. $\square$

Example. $dDistance_{linkedLists}([1, 4, 3], [1, 2, 3]) = |\{[1, 2, 3], [2, 3], [2]\}| = 3$.

Now that we have defined measurements that formalize the ideas of similar abstract values and of similar decompositions, we are ready to talk about *stable decompositions* and the relevance of stable decompositions to incremental evaluation.

Definition 6.7 (*stable decompositions*). Let D be a decomposition scheme for values of type τ, T be a set of transformations on type τ, and $|x|$ be a measurement of the size of x.

The decomposition scheme D is said to be $O(f(|x'|))$ stable for T transformations if, for all x and x' in τ such that $tDistance_T(x, x')$ is defined, $dDistance_D(x, x')$ is $O(tDistance_T(x, x')\, f(|x'|))$.

If $dDistance_D(x, x')$ is $O_{prob}(tDistance_T(x, x')\, f(|x'|))$, D is said to be $O_{prob}(f(|x'|))$ stable for T transformations. $\square$

Example. The *linkedLists* decomposition scheme for sequences (defined in the example for Definition 6.4) is $O(1)$ stable for {*deleteFirst, changeFirst, insertBeforeFirst*} transformations[1], $O(0)$ stable for {*deleteFirst*} transformations[2] and $O(n)$ stable for {*changeLast, deleteLast*} transformations[3] (i.e., totally unstable).

Theorem 6.1. Let $R : \sigma \rightarrow \tau$ be a representation function and T be a set of transformations on τ. Let $D = \langle \textit{atomic}, \textit{split}, \textit{bound} \rangle$ be a decomposition scheme for τ and $D' = \langle \textit{atomic}', \textit{split}', \textit{bound}' \rangle$ be an implementation of D for R (e.g., $\textit{split}(x) = \langle y, z \rangle$ iff $\textit{split}'(R^{-1}(x)) = \langle R^{-1}(y), R^{-1}(z) \rangle$) such that *atomic'* and *split'* are constant-time functions. Let g be a function on σ defined as

$$g(x) \equiv \textbf{if } \textit{atomic}'(x) \textbf{ then } h_1(x) \textbf{ else } h_2(g(y), g(z)) \textbf{ where } \langle y, z \rangle = \textit{split}'(x) \textbf{ fi}.$$

If h_1 and h_2 can be computed in constant time, D is $O(f(\,|x|\,))$ stable for T transformations, the results of all the recursive invocations of g involved in computing $g(x)$ are stored in the function cache, and function caching imposes only a constant-factor overhead, then the time required to compute $g(x')$ is $O(\textit{tDistance}_T(R(x), R(x'))\ f(\,|x|\,))$. If D is $O_{prob}(f(\,|x|\,))$ stable, the time required to compute $g(x')$ is $O_{prob}(\textit{tDistance}_T(R(x), R(x'))\ f(\,|x|\,))$.

Proof. Computing $g(x')$ without function caching requires computing $g(y)$ for each y in $d_{D'}(x')$. Excluding the cost of performing recursive calls to g, each of these calls requires constant time (e.g., the total time to compute $g(x')$ without function caching is $O(\,|d_{D'}(x')|\,)$. The only calls to g that will not be found in the function cache are the calls with arguments from $d_{D'}(x') - d_{D'}(x)$. Because D' is defined as an implementation of D, $|d_{D'}(x') - d_{D'}(x)| = |d_D(R(x')) - d_D(R(x))|$. Since D is $O(f(\,|x|\,))$ stable for T transformations, $|d_D(R(x')) - d_D(R(x))|$ is $O(\textit{tDistance}_T(R(x), R(x'))\, f(\,|x|\,))$. The argument follows similarly for the O_{prob} case. □

6.4 Hash keys for data structures

The data structures we describe in Chapters 7 and 8 organized on the basis of the hash keys of values. Chapter 4 discusses methods for generating hash keys. The hash keys we need for the data structures in Chapters 7 and 8 are the kind usable for extendible hashing (i.e., let $\textit{hash}(x)$ be a hash function mapping U to $0..2^k-1$; if for all j such that $1 \leq j \leq k$, $\textit{hash}(x) \bmod 2^j$ is a "good" hash function, then $\textit{hash}(x)$ is acceptable for our purposes).

The hash functions described in Section 4.1 map S-expressions to $0..m-1$ where $m \bmod 6 \neq 0$. For extendible hashing, we need a hash function whose range is $0..2^k-1$ (for some k). Except for the problem with the range of the hash keys, the hash functions produced by the techniques of Section 4.1 are usable for extendible hashing. However, this is a minor problem. If we use a hash function whose range is, for example, $0..(2^{16}-3)-1$ instead of $0..2^{16}-1$, it should still be effectively usable for extendible hashing.

---

[1] The sequence of transformations that produces the largest difference between the decompositions of x and x' is a series of insertions. If x' is the result of inserting k new elements at the front of x, then $\textit{tDistance}_{\{deleteFirst,\ changeFirst,\ insertBeforeFirst\}}(x, x') = k$ and $\textit{dDistance}_{linkedLists}(x, x') = 2k$.

[2] If $\textit{tDistance}_{\{deleteFirst\}}(x, x')$ is defined x' is a suffix of x so $\textit{dDistance}_{linkedLists}(x, x') = 0$.

[3] If x' is identical to x except for the last element, $\textit{tDistance}_{\{changeLast,\ deleteLast\}}(x, x') = 1$ and $\textit{dDistance}_{linkedLists}(x, x') = |x'|+1$.

6.5 Tagged Tuples

In the next two chapters, we describe data structures using tagged tuples: the value Id⟨ x_0, x_1, ..., x_{k-1} ⟩ is the tuple tagged with the token Id and containing the values x_0, x_1, ..., x_{k-1}. Each tag is associated with a fixed arity.

Tagged tuples are used for clarity of expression. The tagged tuple Id⟨ x_0, x_1, ..., x_{k-1} ⟩ can be thought of syntactic sugar for the S-expression ⟨ Id, ⟨ x_0, ⟨ x_1, ... ⟨ x_{k-1}, NIL ⟩ ... ⟩⟩⟩. Tagged tuples are supported by many functional languages such as ML, and can be incorporated into a static typechecking scheme.

6.6 Discussion

The ideas presented in this chapter have wide application outside of incremental evaluation. A representation scheme that provides unique representations, combined with a method for constant-time equality testing of concrete values, allows constant-time equality testing of abstract values. For many abstract data types, efficient representations that allow constant-time equality testing have been unknown. Sassa and Goto [SG76] describe a set representation that allows constant-time equality tests. However in their representation, any set operation takes time at least proportional to the size of the set being created (e.g., to compute $S + \{x\}$ requires $O(|S|)$ time). It is possible to use a linked list representation and the techniques discussed in Chapter 4 to obtain constant-time equality tests for sequences, but in this representation modifying element i of a sequence requires $O(i)$ time.

The idea of a stable decomposition also has implications outside of incremental evaluation. Typically, values that have similar decompositions are represented by similar concrete values. When combined with a technique such as hashed consing, this allows two representations of similar values to share storage. Reps discusses using sharable 2-3 trees for sharing storage between similar values in his thesis [Reps83]. However, using his methods, two values can share storage only if they are similar in a "computation-oriented" sense, rather than in a "value-oriented" sense (e.g., if B has been calculated as $A + \{x\}$, A and B can share storage; if $B = A + \{x\}$ but B was not calculated that way, it might not be possible for the sets to share storage). Reps notes as an open problem a scheme that will allow "value-oriented" similar values to share storage; a problem we can solve if we can design appropriate representations.

Of course, this chapter would be somewhat pointless if data structures fulfilling these criteria did not exist. Fortunately, we have developed efficient representations for sequences and sets that have allow efficient applicative updates and have unique representations and stable decompositions. In Chapter 7, we present a representation for sequences that provides unique representations and is $O_{prob}(\log n)$ stable for {*insert*, *change*, *delete*} transformations. In Chapter 8, we present a representation for sets that provides unique representations and is $O_{prob}(\log n)$ stable for {*addElement*, *deleteElement*} transformations.

7 A Decomposition and Representation Scheme for Sequences

This chapter describes a stable decomposition scheme for sequences (i.e., lists) and a scheme for representing sequences using tagged-tuples or S-expressions. We also describe algorithms that work with this representation: the function that splits a sequence requires constant-time; functions that access, change, delete and insert an element in a sequence of length n requires $O_{prob}(\log n)$ time; and the function that appends a sequence of length n to a sequence of length m requires $O_{prob}(\log n + \log m)$ time.

7.1 The chunky decomposition scheme

The chunky decomposition scheme is $O_{prob}(\log n)$ stable for {*insert, delete, change*} transformations and is based on the *levels* of elements of sequences, which are in turn based on the hash keys of elements.

Definition 7.1 (*level*(x)). The level of an element x, *level*(x), is the largest integer i such that *hash*(x) is a multiple of 2^i. □

The properties of hash functions stated in Section 6.4 guarantee that the levels of elements have the same probability distribution as $NB(1)$ (e.g., on average, half of the elements of a sequence are level 0, one quarter are level 1, and so on). So that we can treat the level of each element as an independent random variable we assume that no duplicate elements exist. Problems caused by duplicate elements are discussed in more detail in Section 7.2. Occurrences of only a relatively small number of duplicate elements has no significant effect.

Definition 7.2 (*chunky decomposition scheme*). The chunky decomposition scheme on sequences is defined as:

$$
\begin{aligned}
&\mathit{atomic}(S) \equiv |\,S\,| \le 1,\\
&\mathit{split}([\,x_0, \ldots, x_{n-1}\,]) \equiv \langle\, [\,x_0, \ldots, x_{i-1}\,], [\,x_i, \ldots, x_{n-1}\,] \,\rangle,\\
&\quad \textbf{where } i \textbf{ uniquely satisfies}\\
&\qquad 0 < i < n\\
&\qquad \wedge (\forall j \text{ s.t. } 0 < j < i,\ \mathit{level}(x_i) \ge \mathit{level}(x_j))\\
&\qquad \wedge (\forall j \text{ s.t. } i < j \le n,\ \mathit{level}(x_i) > \mathit{level}(x_j)), \text{ and}\\
&\mathit{bound}(S) \equiv |\,S\,|. \square
\end{aligned}
$$

Informally, the chunky decomposition scheme divides a sequence $S = [\,x_0, \ldots, x_{n-1}\,]$ at the most *preferable break* in S. A *break* is a position between two elements in a sequence. The break immediately before element x_i is *preferable* to the break immediately before element x_j iff *level*(x_i) > *level*(x_j) or (*level*(x_i) = *level*(x_j) and $i > j$). The most preferable break therefore is immediately to the left of the rightmost element of S with maximal level. Figures 7.1 and 7.2 show the decompositions of two similar sequences according to this decomposition scheme.

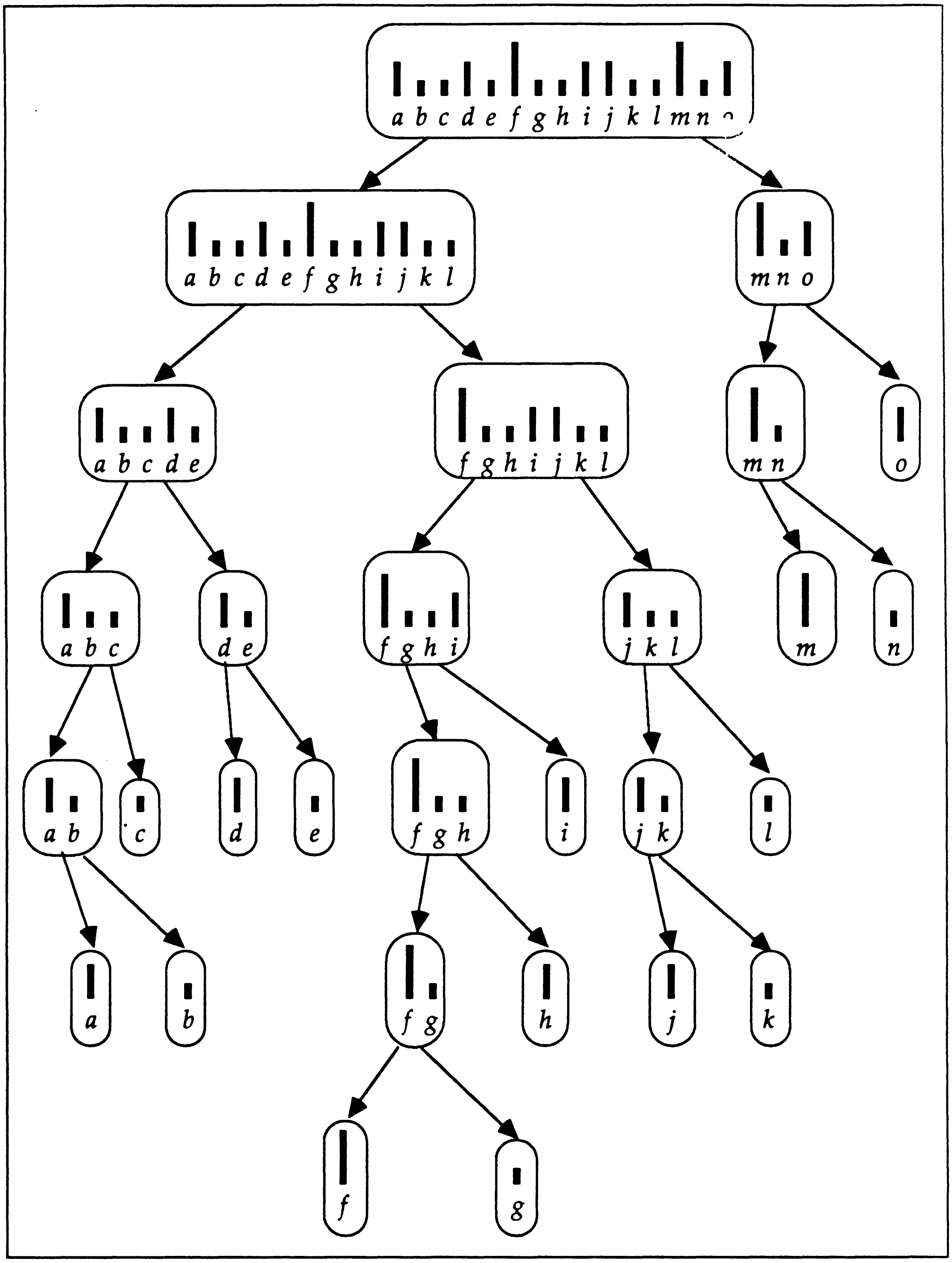

FIGURE 7.1 - Decomposition of a sequence according to the chunky decomposition scheme. The level of each element is indicated by the height of the bar above that element.

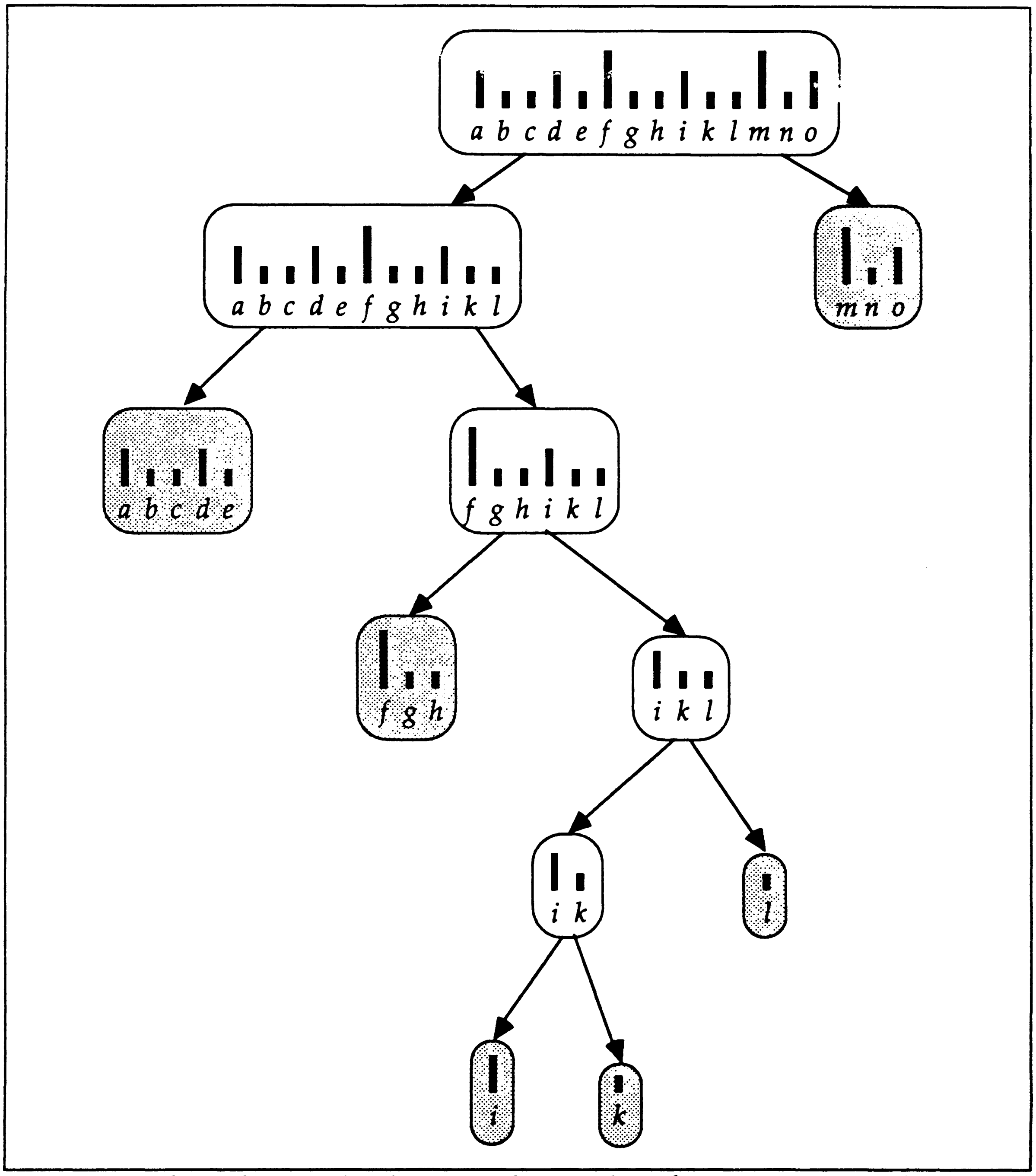

FIGURE 7.2 - Chunky decomposition of a sequence that is similar to the sequence in Figure 7.1. Sequences in this figure that also appear in the decomposition in Figure 7.1 are shown in grey here and are not further decomposed in this figure.

Theorem 7.1 characterizes the sequences that appear in the decomposition of a sequence.

Theorem 7.1. Let $S = [\ x_0, x_1, ..., x_{n-1}\ ]$. For all i, j such that $0 \le i \le j < n$, $[\ x_i, ..., x_j\ ]$ appears in the decomposition of S iff $i = j$ or $level(x_i) > \max(\ level(x_{i+1}), level(x_{i+2}), ..., level(x_j)\ ) \le level(x_{j+1})$ (to handle boundary conditions, assume that $level(x_0) = \infty$, an element x_n exists and $level(x_n) = \infty$).

Proof. The sequence $[\ x_i, ..., x_j\ ]$ appears in the decomposition of S iff the breaks immediately before x_i and before x_{j+1} are preferable to the breaks before $x_{i+1}, ..., x_j$. The break immediately before x_i is preferable to the breaks before $x_{i+1}, ..., x_j$ iff $level(x_i) > \max(\ level(x_{i+1}), level(x_{i+2}), ..., level(x_j)\ )$. The break immediately before x_{j+1} is preferable to the breaks before $x_{i+1}, ..., x_j$ iff $\max(\ level(x_{i+1}), level(x_{i+2}), ..., level(x_j)\ ) \le level(x_{j+1})$. □

Theorem 7.1 nicely describes why this decomposition scheme is stable: whether or not $[\ x_i, ..., x_j\ ]$ appears in the decomposition of S depends solely on the levels of the elements $x_i, ..., x_j$ and of x_{j+1}. We now analyze the stability of this decomposition scheme with respect to {*insert, delete, change*} transformations.

Theorem 7.2. Let $S = [\ x_0, ..., x_{i-1}, x_{i+1}, ..., x_{n-1}\ ]$ and $S' = [\ x_0, ..., x_{n-1}\ ]$ (i.e., S' is the same as S except that a new element x_i has been inserted). The only sequences that appear in the decomposition of S' but not in the decomposition of S are those that include either x_{i-1} or x_i.

Proof. Let $[\ x_j, ..., x_k\ ]$ be a sequence not including x_{i-1} or x_i that appears in the decomposition of S' (i.e., $k < i-1$ or $i < j$). The presence of this sequence in the decomposition of S' does not depend on the level of x_i, and therefore it must also appear in the decomposition of S.

Theorem 7.3. Let $S = [\ x_0, ..., x_{n-1}\ ]$ and $S' = [\ x_0, ..., x_{i-1}, x_{i+1}, ..., x_{n-1}\ ]$ (i.e., S' is the same as S except that x_i has been deleted). The only sequences that appear in the decomposition of S' but not in the decomposition of S are those that include x_{i-1}.

Proof. Let $[\ x_j, ..., x_k\ ]$ be a sequence not including x_{i-1} that appears in the decomposition of S' (i.e., $k < i-1$ or $i < j$). The presence of this sequence in the decomposition of S' does not depend on the absence of x_i, and therefore it must also appear in the decomposition of S.

Theorem 7.4. Let $S = [\ x_0, ..., x_{n-1}\ ]$ and $S' = [\ x_0, ..., x_{i-1}, x_i', x_{i+1}, ..., x_{n-1}\ ]$ (i.e., S' is the same as S except that x_i has been replaced by x_i'). The only sequences that appear in the decomposition of S' but not in the decomposition of S are those that include either x_{i-1} or x_i'.

Proof. Let $[\ x_j, ..., x_k\ ]$ be a sequence not including x_{i-1} or x_i' that appears in the decomposition of S' (i.e., $k < i-1$ or $i < j$). The presence of this sequence in the decomposition of S' does not depend on the level of x_i', and therefore it must also appear in the decomposition of S.

Theorems 7.5 and 7.6 show that the chunky decomposition scheme is $O_{prob}(\log n)$ stable for {*insert, delete, change*} transformations.

Theorem 7.5. Let x_d be a designated element in a sequence $S = [\ x_0, ..., x_{n-1}\ ]$. The number of sequences containing x_d that appear in the decomposition of S is $O_{prob}(\log n)$.

Proof. Let $S_0, S_1, ..., S_{k-1}$ be the sequences that appear in the decomposition of S that include x_d. where $S_0 = [\ x_d\ ]$, $S_{k-1} = S$ and for all i, $0 \le i < k-1$, S_i is a either a proper prefix or a proper suffix of S_{i+1}.

Let $Left_i$ and $Right_i$ be defined such that $S_i = [\ x_{Left_i}, x_{Left_i+1}, ..., x_{Right_i-1}\ ]$. For all i, $0 \le i < k-1$, either $Left_{i+1} < Left_i$ and $Right_i = Right_{i+1}$ or $Left_{i+1} = Left_i$ and $Right_i < Right_{i+1}$.

If $Left_{i+1} < Left_i$, then $level(x_{Left_{i+1}}) > level(x_{Left_i})$. The number of distinct values in $[Left_0, Left_1, ..., Left_k] \le 1 + \max(\ level(x_0), level(x_1), ..., level(x_d)\ )$. From Theorem A.2, we derive the result that $\max(\ level(x_0), level(x_1), ..., level(x_d)\ ) \le_{prob} \lg d + NB(1)$.

The number of distinct values in $[Right_0, Right_1, ..., Right_k]$ is the number of non-decreasing elements in $[\ level(x_d), level(x_{d+1}), ..., level(x_{n-1})\ ]$. From Theorem A.3, we derive the result that this value is $\leq_{prob} 1 + NB(1 + \lg(n-d)) + NB(1, 1/3)$.

Therefore, k = (the number of of distinct values in $[Left_0, Left_1, ..., Left_{k-1}]$) + (the number of distinct values in $[Right_0, Right_1, ..., Right_{k-1}]$) – 1. Combining our results, we get

$$k \leq_{prob} 1 + \lg d + NB(1) + NB(1 + \lg(n-d)) + NB(1, 1/3). \square$$

Theorem 7.6. The chunky decomposition scheme is $O_{prob}(\log n)$ stable for {*insert, delete, change*} transformations.

Proof. Let S and S' be sequences such that $tDistance_{\{insert, delete, change\}}(S, S') = k$. By Theorems 7.2, 7.3 and 7.4, there is a set D of no more than $2k$ elements from S' such that a sequence appears in the decomposition of S' but not of S only if it contains an element from D. Based on Theorem 7.5, the number of sequences appearing in the decomposition of S' that contain an element from D is $O_{prob}(|D| \log |S'|)$. $\square$

7.2 Duplicate elements

The reason we have assumed that sequences do not contain any duplicate elements is that we treat the levels of elements as independent random variables. For example, if a sequence simply contained n instances of an element x, all the elements would have the same level, and the decomposition rule would simply remove the rightmost element of a sequence. If no more than a small proportion of the elements in a sequence are duplicates, it should have no significant impact. We are pursuing research on a representation for sequences with duplicate elements, but it is currently an open problem.

7.3 The chunky list representation scheme

In this section, we describe a representation scheme for sequences that provides unique representations and allows a sequence to be split according to the chunky decomposition rule in constant time. We also describe algorithms for appending the representation of two sequences and accessing, deleting, changing or inserting elements in the representation of a sequence; these algorithms all require $O_{prob}(\log n)$ time.

We define the chunky representation scheme by defining a inverse representation function R^{-1} that maps a sequence S to S-expression that represents S. These S-expressions are presented as tagged tuples, as described in Section 6.5. Any S-expression in the range of R^{-1} is termed a *chunky list*.

Definition 7.3 (*chunky lists*). The inverse representation function R^{-1} that maps sequences to chunky lists is

$R^{-1}(S) \equiv$
  if $S = [\,]$ **then** Empty⟨ ⟩
  else if $|S| = 1$ **then** Element⟨ c ⟩
    where c uniquely represents the element contained in S
  else cList⟨ $R^{-1}(S'), R^{-1}(S'')$ ⟩ **where** ⟨ S', S'' ⟩ = $split(S)$.

The chunky list already reflects the chunky decomposition scheme, so no work is needed to decompose the representation of a sequence; the *split* function on chunky lists is constant-time. The implementation of *append* described in Figure 7.3 requires $O_{prob}(\log n + \log m)$ time to append the representations of two sequences of length n and m. This algorithm uses the pattern-matching case statement that is typical of functional programming languages: free variables in the pattern are assigned appropriate values according to the value that the case statement is discriminating on.

We extend *level* so that, when applied to a chunky list representing a sequence S, it is the level of the first element of S. By caching the level of the first element in a sequence S with the data structure representing S, the *level* function remains a constant-time function (i.e., we change the chunky list representation scheme so that $R^{-1}([\, x_0, x_1, ..., x_{n-1} \,]) = \text{cList}\langle R^{-1}(S'), R^{-1}(S''), level(x_0) \rangle$ where $\langle S', S'' \rangle = split([\, x_0, x_1, ..., x_{n-1} \,])$). This change decreases the clarity of the algorithms slightly and is a fairly easy, mechanical change, so it has been omitted from the descriptions given below.

```
append(S, T) ≡
  case ( S, T ) of

      ( Empty( ), * ) ⇒ T

      ( *, Empty( ) ) ⇒ S

      ( Element( * ), Element( * ) ) ⇒
          cList( S, T )

      ( Element( * ), cList( T0, T1 ) ) ⇒
          if level(T0) ≤ level(T1)
              then cList( append(S, T0), T1 )
              else cList( S, T )

      ( cList( S0, S1 ), Element( * ) ) ⇒
          if level(S1 ≤ level(T)
              then cList( S, T )
              else cList( S0, append(S1, T) )

      ( cList( S0, S1 ), cList( T0, T1 ) ) ⇒
          if level(S1) ≤ level(T1) and level(T0) ≤ level(T1)
              then cList( append(S, T0), T1 )
          else if level(S1) ≤ level(T0) and level(T1) < level(T0)
              then cList( S, T )
          else cList( S0, append(S1, T) )
```

FIGURE 7.3 – Description of algorithm to append two sequences represented as chunky lists.

Theorem 7.6. The time required by the algorithm of Figure 7.3 to append the representations of two sequences S and T is $O_{prob}(\log |S| + \log |T|)$.

Proof. Let $S = [\, x_0, x_1, ..., x_{n-1} \,]$ and $T = [\, y_0, y_1, ..., y_{m-1} \,]$. Each recursive step in *append* involves stepping down a right branch of the data structure representing S or stepping down a left branch on the data structure representing T. The number of right steps in the data structure representing S is the number of sequences containing x_{n-1} that appear in the decomposition of S, which is

$O_{prob}(\log |S|)$. The number of left steps in the data structure representing T is the number of sequences containing y_0 that appear in the decomposition of T, which is $O_{prob}(\log |T|)$. □

We define functions *length*, *first* and *last* such that *length(S)* is the number of elements in the sequence represented by S, *first(S, i)* is the chunky list representing the first i elements in the sequence represented by S, and *last(S, i)* is the chunky list representing the last i elements in the sequence represented by S. The function *length* is cached with the data structures, as describe for the level function, so that it is a constant time function. We can implement *first* and *last* as shown in Figure 7.4, and functions to access, change, insert, or delete elements,as shown in Figure 7.5. Each of these operations requires $O_{prob}(\log n)$ time.

```
first(S, i) ≡
  If i = 0 then Empty( )
  else If i = length(S) then S
  else case S of
      cList( S0, S1 ) ⇒
          If i ≤ length(S0) then first(S0, i)
          else cList( S0, first(S1, i – length(S0)) )

last(S, i) ≡
  If i = 0 then Empty( )
  else If i = length(S) then S
  else case S of
      cList( S0, S1 ) ⇒
          If i ≤ length(S1) then last(S1, i)
          else cList( last(S0, i – length(S1)), S1 )
```

FIGURE 7.4 – Description of algorithms to extract the first and last i *elements of a sequence.*

```
access(S, i) ≡
  case last(first(S, i+1), 1) of
      Element( x ) ⇒ x

change(S, x, i) ≡
  append(first(S, i), append(Element( x ), last(S, length(S) – i – 1)))

delete(S, i) ≡ append(first(S, i), last(S, length(S) – i – 1))

insert(S, x, i) ≡
  append(first(S, i), append(Element( x ), last(S, length(S) – i)))
```

FIGURE 7.5 – Description of algorithms to change, delete and insert before element i *of a sequence.*

8 A Decomposition and Representation Scheme for Sets

The algorithms and representations we describe use annotated sets: a set annotated with a string of binary digits (possibly null). For example, $\langle S, L \rangle$ is the set S annotated with the string L. If $\langle S, L \rangle$ is an annotated-set value, then for all elements x in S, L is a suffix of the $hash(x)$. A set S is represented by the annotated set value $\langle S, \varepsilon \rangle$ (where ε is the null string). The predicate $suffix(L, k)$ is true iff L is a suffix of the binary representation of k. The term $hashBits$ denotes the number of bits in a hash key (i.e., $hash(x) \in 0..2^{hashBits} - 1$). Since our set algorithms work with annotated sets, we define our decomposition scheme on annotated sets as shown below.

Definition 8.1 (*decomposition scheme for annotated-sets*). The decomposition scheme for annotated sets is defined as:

$$\begin{aligned}
&atomic(\langle S, L \rangle) \equiv |\{ hash(x) \mid x \in S \}| \leq 1,\\
&split(\langle S, L \rangle) \equiv \langle\ \langle \{ x \in S \mid suffix(0L, hash(x)) \}, 0L \rangle,\\
&\qquad\qquad\qquad\quad \langle \{ x \in S \mid suffix(1L, hash(x)) \}, 1L \rangle \rangle, \text{ and}\\
&bound(\langle S, L \rangle) \equiv \textbf{if } atomic(\langle S, L \rangle) \textbf{ then } 0 \textbf{ else } hashBits - |L|. \square
\end{aligned}$$

The representation scheme we use for annotated sets can be described by defining the inverse representation function R^{-1} as shown below. Let r^{1} be the inverse representation function for a representation scheme in which sets are represented by a sorted list of the elements of the set.

$$\begin{aligned}
&R^{-1}(\langle S, L \rangle) \equiv\\
&\quad \textbf{if } S = \varnothing \textbf{ then } \mathsf{EmptySet}\langle L \rangle\\
&\quad \textbf{else if } atomic(\langle S, L \rangle) \textbf{ then } \mathsf{AtomicSet}\langle r^{1}(S), L \rangle\\
&\quad \textbf{else } \mathsf{Pair}\langle R^{-1}(\langle S', 0L \rangle), R^{-1}(\langle S'', 1L \rangle), L \rangle,\\
&\qquad \textbf{where } \langle \langle S', 0L \rangle, \langle S'', 1L \rangle \rangle = split(\langle S, L \rangle)
\end{aligned}$$

This representation scheme is equivalent to using *binary hash tries*. Binary hash tries use a binary trie [Knu73] data structure based on the hash keys of the elements. Atomic annotated sets that appear in the decomposition of $\langle S, \varepsilon \rangle$ appear as leaf nodes in the trie representing $\langle S, \varepsilon \rangle$ and non-atomic annotated sets appear as interior nodes. Figure 8.1 shows the representation of a sample set, using the hash keys given in Table 8.1.

With this representation, the appropriate divide and conquer algorithms are fairly obvious. If $split(\langle A, L \rangle) = \langle \langle A', 0L \rangle, \langle A'', 1L \rangle \rangle$ and $split(\langle B, L \rangle) = \langle \langle B', 0L \rangle, \langle B'', 1L \rangle \rangle$, we compute $\langle A, L \rangle \cup \langle B, L \rangle = \langle \langle A', 0L \rangle \cup \langle B', 0L \rangle, \langle A'', 1L \rangle \cup \langle B'', 1L \rangle \rangle$. The algorithm for performing set union is described in Figure 8.2. One point to note is that our algorithms make use of unique representations and constant-time equality tests. For example, when computing $\langle A, L \rangle \cup \langle B, L \rangle$, if $A = B$ we can immediately return $\langle A, L \rangle$ as the answer.

Pardo [Par78] suggested an equivalent data structure for representing sets. He did not consider the application of this representation for incremental computation, and his analysis was more complicated than ours and achieved poorer results. Part of the reason for his poorer results is that he did not make use of the ability to perform constant-time equality tests between sets provided by this representation.

If hash keys are chosen to be sufficiently long, atomic sets will very rarely contain more than a single element, and therefore an atomic set S can be efficiently represented by a sorted list of the elements of S.

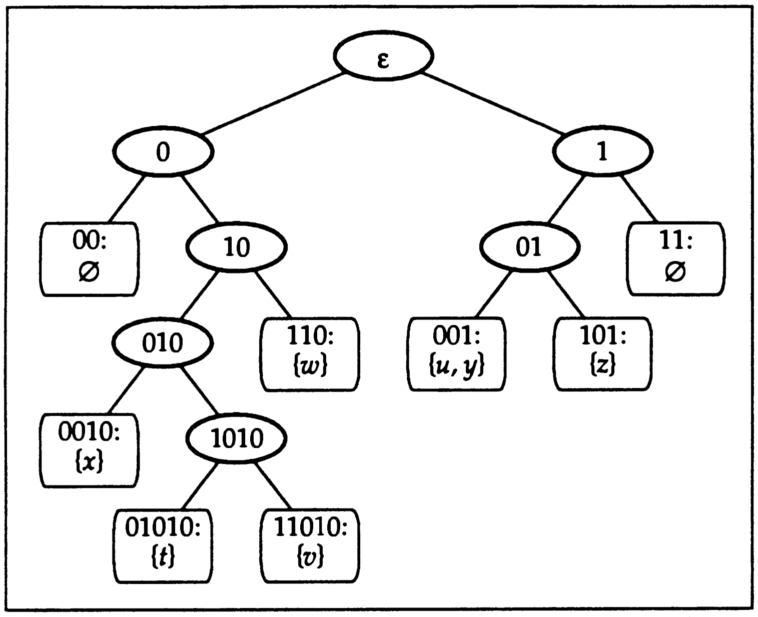

FIGURE 8.1 – The representation of the set {t, u, v, w, x, y, z} as a binary hash trie, using the hash keys shown in Table 8.1.

element	5-bit hash key
t	01010
u	01001
v	11010
w	00110
x	10010
y	01001
z	10101

TABLE 8.1 – Hash keys of the elements in the set in Figure 8.1

8.1 Set operations

The algorithms for performing set union are described in Figure 8.2. Other standard set operations such as intersection, difference and subset tests can be implemented in a method similar to the method used for set union, and the same time analysis holds. For subset tests, the result *false* may be computed substantially faster than the time bounds given in Section 8.2. The algorithm in Figure 8.2 and the analysis in Section 8.2 can be applied to any set operation *op* that satisfies the following properties:

(1) the computations A *op* $\varnothing$, $\varnothing$ *op* A and A *op* A are constant-time computations and either

(2*a*) for all sets C, A *op* B = $((A \cap C)$ *op* $(B \cap C)) \cup ((A - C)$ *op* $(B - C))$ and $((A \cap C)$ *op* $(B \cap C)) \subseteq C$, or

(2*b*) for all sets C, the result of A *op* B can be computed in constant time from the results of $(A \cap C)$ *op* $(B \cap C)$ and $(A - C)$ *op* $(B - C)$.

Set union, intersection, difference, and subset tests all possess these properties. Condition 1 makes sure that we get fast termination when the two arguments are equal. If condition 2 is satisfied, the problem can be solved using a divide and conquer approach. Consider calculating $A \cup B$. Let C be the set of all elements that have even hash keys. The algorithm in Figure 8.2 divides this computation into the problems of computing $((A \cap C) \cup (B \cap C))$ and of computing $((A - C) \cup (B - C))$. The algorithm also expects that $((A \cap C) \cup (B \cap C)) \subseteq C$.

8.2 Analysis

The techniques used in this section give a good example of the techniques described in Chapter 5 for discussing probabilistic upper bounds. In our analysis, we make the assumption that all elements have distinct hash keys. In practice, the small number of hash key collisions typically seen should have no significant impact.

```
union(X, Y) ≡ { set union }
  case ( X, Y ) of
    ( AnnotatedSet( A ), AnnotatedSet( B ) ) ⇒ aUnion(A, B)

aUnion(X, Y) ≡ { annotated-set union }
  if X = Y then X
  else if readyForAtomicUnion(X,Y) then atomicUnion(X, Y)
  else case ( splitAtomic(X), splitAtomic(Y) ) of
    ( *, EmptySet( L ) ) ⇒ X
    ( EmptySet( L ), * ) ⇒ Y
    ( Pair( A0, A1, L ), Pair( B0, B1, L ) ) ⇒
      Pair( aUnion(A0, B0), aUnion(A1, B1), L)

splitAtomic(X) ≡
  case X of
    EmptySet( L ) ⇒ X
    Pair( *, *, * ) ⇒ X
    AtomicSet( B, L ) ⇒
      if suffix(0L, hash(B) )
        then Pair( AtomicSet( B, 0L ), EmptySet( 1L ), L )
        else Pair( EmptySet( 0L ), AtomicSet( B, 1L ), L )

readyForAtomicUnion(X, Y) ≡
  case ( X, Y ) of
    ( AtomicSet( A, L ), AtomicSet( B, L ) ) ⇒ hash(A) = hash(B)
    ( *, * ) ⇒ false

atomicUnion(X, Y) ≡
  case ( X, Y ) of
    ( AtomicSet( A, L ), AtomicSet( B, L ) ) ⇒
      AtomicSet( PrimitiveUnion(A, B), L )
```

FIGURE 8.2 – A description of the algorithm for set union.

The results in this section are presented in an order such that the simplest proof are described first. With the exception of the proof for Theorem 8.7 in Section 8.2.5, these proofs are all fairly understandable. The proof for Theorem 8.7 is very intricate, and can be skipped by the casual reader.

Results obtained

Theorem 8.3 shows that the time required to compute $A \; op \; B$, where $op \in \{ \cup, \cap, -, \subseteq \}$, is bounded by

$$O_{prob}(\, |S_2| \, (1 + \log(|S_3| / |S_2|)) \,)$$

where S_2 and S_3 are the middle and largest of the sets $A - B$, $B - A$, $A \cap B$. If the sets A and B differ in only k elements, the time required to calculate A *op* B is therefore only

$$O_{prob}(\, k\,(1 + \log(\,|A \cap B\,|\,/\,k))\,).$$

Theorem 8.4 shows that the decomposition scheme we have described for annotated sets is $O_{prob}(\log n)$ stable for (*addElement, deleteElement*) transformations. This would give us fairly good time bounds for incremental set operations, but we can do even better. Theorem 8.6 gives the following result. Assume the computations involved in determining if $x \in S$ are stored in the cache. Define ΔS to be $(S' - S) \cup (S - S')$. The time required to determine if $x \in S'$ is bounded by $O_{prob}(\log\,|\Delta S\,|)$ if $x \notin \Delta S$ and by $O_{prob}(\log\,|\,S'\,|)$ otherwise.

Theorem 8.7 gives a bound on more general incremental set operations. Assume that the computations involved in computing A *op* B are stored in the cache and consider computing A' *op* B', where A' is similar to A and B' is similar to B. Let S_1, S_2 and S_3 be the smallest, middle and largest of the sets $A' - B'$, $B' - A'$, $A' \cap B'$. Let $\Delta Both = (A' - A) \cup (A - A') \cup (B' - B) \cup (B - B')$. The time required to compute A' *op* B' is bounded by

$$O_{prob}(\;|\Delta Both - (S_1 \cup S_2)|\;\log(\,|S_2|\,/\,|\Delta Both\,|) + \;|\Delta Both \cap (S_1 \cup S_2)|\;\log\,(\,|S_3|\,/\,|\Delta Both\,|\,)\,).$$

8.2.1 Analysis techniques

In our analysis, we make use of a *hash-space tree*. A hash-space tree is rooted at a node labeled ε. Let n be a node in a hash-space tree labeled L. If the length of L is equal to the length of the hash keys used, n is a leaf node. Otherwise, n has two children, labeled $0L$ and $1L$. We define several terms regarding sets and hash-space trees.

Definition 8.2 (*select(S, L)*). Let S be a set and L be a label. We define *select*(S, L) to be the annotated set value $\langle\, S', L\,\rangle$ that appears in the decomposition of $\langle\, S, \varepsilon\,\rangle$. If no such value appears in the decomposition of $\langle\, S, \varepsilon\,\rangle$, *select*$(S, L)$ is undefined.

More formally, let *tail*(L) denote the label containing all but the first digit of L (i.e., *tail*$(0L)$ = *tail*$(1L) = L$). The term *select*(S, L) is defined iff $L = \varepsilon$ or *select*$(S,$ *tail*$(L))$ is defined and non-atomic. If *select*(S, L) is defined, *select*$(S, L) = \langle\, \{\, x \in S \mid suffix(L, hash(x))\,\}, L\,\rangle$. □

Definition 8.3 (*level*). The *level* (i.e., depth) of a node n in a hash-space tree is the length of the label of n. □

Definition 8.4 (*matching*). An element x *matches* a node n in a hash-space tree iff the label of n is a suffix of the hash key of x. □

8.2.2 The depth of leaf nodes and the size of set representations

We first analyze the depth of these set data structures: for an arbitrary set S and a particular element x in S, what is the depth of the leaf node corresponding to x in the trie representing S?

Start with an uncolored hash-space tree. Dye *red* all the nodes that match an element in $S - \{x\}$. Dye *blue* all the nodes that match x (*red* + *blue* yields purple). This coloring is shown in Figure 8.3 for the set and hash keys shown in Figure 8.1 and Table 8.1 (white nodes are nodes that do not match any element in S). Let n be the highest blue node (i.e., the blue node with the shortest label). Let L be the label of n. Since x is the only element in S that matches n, $\{\, y \in S \mid suffix(L, hash(y))\,\} = \{x\}$. Since the parent of n is purple, $|\,\{\, y \in S \mid suffix(tail(L), hash(x))\,\}\,| > 1$. Therefore, *select*$(S, L) = \langle \{x\}, L \rangle$, which is atomic. In the trie representing S, the leaf node representing x will be $\langle \{x\}, L \rangle$, which is at a depth equal to the length of L. The length of L is equal to the number of purple nodes in the hash-space tree.

Definition 8.5 (*dyes* and *colors*). A node is *dyed* a tint if the node contains that tint dye. A node is *colored* a color if the final color of a node is that color. To make this distinction clearer, tints will be appear in *italics typeface* and colors will be appear in roman typeface. □

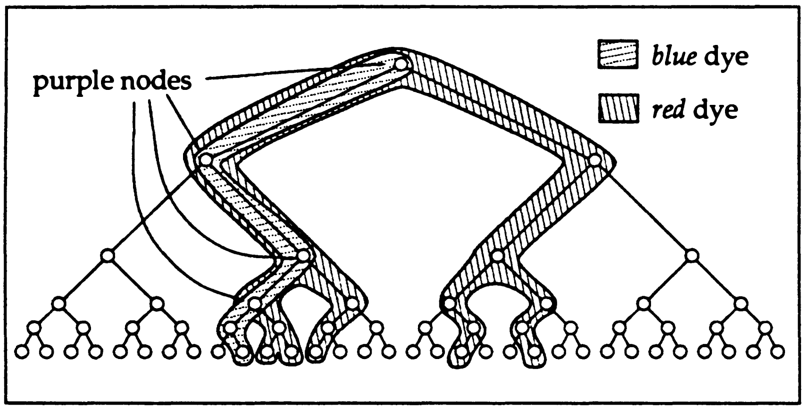

FIGURE 8.3 – Shows the result of dyeing blue the nodes in a hash-space tree that match x *and dyeing red nodes that match elements in* S – {x}*. The set* S *and the hash keys are taken from Table 8.1 and Figure 8.1.*

Lemma 8.1. For an arbitrary set S and an element x, when dyeing *red* all the nodes in a hash-space tree that match an element in $S - \{x\}$ and dyeing *blue* all the nodes that match x, the number of purple nodes generated is $\leq_{prob} (\lceil \lg |S| \rceil + 1 + NB(1))$.

Proof. Let N = the smallest power of 2 at least $|S|$. Let p_i = the probability that the level i node that matches x is colored purple. We know that the node at level i that matches x is either purple or blue, so p_i is the probability that the level i node that matches x contains *red* dye (i.e., is dyed *red*).

Consider level $\lg N$. There are N nodes at this level, and it is possible that they will all be dyed *red*. There are no more than $1 + \lg N$ purple nodes at or above level $\lg N$. Consider level $i + \lg N$ ($i > 0$). There are $2^i N$ nodes at this level and only $|S|$ of them can be dyed *red*. The hash key of x is unrelated to the hash key of any element in $S - \{x\}$. Therefore, $p_{i + \lg N} \leq 1/2^i$. This gives a distribution for the number of purple nodes that is $\leq_{prob} \lg N + 1 + NB(1)$. □

Theorem 8.1. For an element x and a set S that contains x, the level of the leaf node containing x is $\leq_{prob} (\lceil \lg |S| \rceil + 1 + NB(1))$.

Proof. The distribution of the level of a leaf node in a tree is the same as the distribution of the number of purple nodes generated when dyeing the nodes in a hash-space tree as described above. Result follows from Lemma 8.1. □

This result gives an upper bound on the expected level of leaf nodes of $\lceil \lg |S| \rceil + 2$. Using a more complicated analysis, Knuth [Knu73] derives a value of approximately $\lg |S| + 1.33275$.

8.2.3 Performance of set operations on disjoint sets

In performing an annotated-set computation, the computation on values annotated with L is associated with the node of the hash-space tree labeled L.

Question. For two disjoint sets A and B, what is the time to compute A *op* B?

In an uncolored hash-space tree, dye *blue* all the nodes that match elements in A. Dye *red* all the nodes that match elements in B (*red* + *blue* yields purple). Purple nodes correspond to non-trivial computations. White nodes correspond to computing $\varnothing$ *op* $\varnothing$. Blue nodes correspond to computing S *op* $\varnothing$. Red nodes correspond computing $\varnothing$ *op* S. The computation associated with a trivial node might be performed only if the parent of that node is non-trivial. Since we assume all elements have distinct hash keys, all atomic set operations take constant time and the computation associated with each node takes constant time. The time required to compute A *op* B is therefore proportional to the number of nodes colored purple, which are exactly those nodes that match elements from both sets.

Question. For two *disjoint* sets A and B, how many purple nodes are generated when all the nodes that match elements in A are dyed *blue* and all the nodes that match elements in B are dyed *red* (i.e., what is the number of nodes in the hash-space tree that match elements from both sets) ?

Assume w.l.g that $|A| \leq |B|$. For each element x in A do the following: Start with a white hash-space tree and dye *red* all the nodes that match elements in B. Dye *blue* all the nodes that match x. By Lemma 8.1, the number of purple nodes generating by this dyeing operation is $\leq_{prob} (\lceil \lg |B| \rceil + 1 + NB(1))$.

The total number of purple nodes generated is bounded by the sum of the number of purple nodes generated in each of these operations. The hash keys of the elements of A are independent. The upper bound we have obtained is independent of the hash keys of the elements of B. The sum of all $|A|$ of these independent $NB(1)$ distributions is therefore $NB(|A|)$. Therefore, the total number of purple nodes $\leq_{prob} |A| \lceil \lg |B| \rceil + |A| + NB(|A|)$.

However, several nodes are counted multiple times (the root node is counted $|A|$ times). There are fewer than $4|A|$ nodes at or above level $\lceil \lg |A| \rceil$ which will we count exactly once. We define a function *Overlap* to count the number of purple nodes deeper than level $\lceil \lg |A| \rceil$.

Definition 8.5 (*Overlap*(A, B, k)). Let A and B be disjoint sets and let k be a non-negative integer such that $|A| \leq |B|$ and $k \leq \lceil \lg |B| \rceil$. We define *Overlap*$(A, B, k)$ to be the number of nodes deeper than level k that match elements in both A and B.

Lemma 8.2. *Overlap*$(A, B, k) \leq_{prob} |A| (\lceil \lg |B| \rceil - k) + NB(|A|)$

Proof. Similar to arguments given above. For each element x in A, the number of nodes deeper than level k that match both x and an element in B is bounded by$(\lceil \lg |B| \rceil + 1 + NB(1) - (k+1))$. Results follows by summing the results for all the elements of A. □

Lemma 8.3. For two *disjoint* sets A and B, the number of nodes that will match elements in both sets $\leq_{prob} (|A|(4 + \lceil \lg |B| \rceil - \lceil \lg |A| \rceil) + NB(|A|))$ (assuming w.l.g. $|A| \leq |B|$).

Proof. As shown above, the number of nodes that will match elements in both sets $\leq_{prob} 4|A| +$ *Overlap*$(A, B, \lceil \lg |A| \rceil)$. Result follows by substituting upper bound on *Overlap*$(A, B, \lceil \lg |A| \rceil)$ shown in Lemma 8.2. □

Theorem 8.2. For two disjoint sets A and B, the time required to compute A *op* B is $O_{prob}(|A| (1 + \log(|B| / |A|)))$ (assuming w.l.g. $|A| \leq |B|$).

Proof. The time required to compute A *op* B is proportional to the number of double matching nodes. Result follows from Lemma 8.3 and Theorem 5.1. □

8.2.4 Performance of set operations on arbitrary sets

Question. Let A and B be arbitrary sets. What is the time required to compute A *op* B?

Starting with a white hash-space tree, dye *blue* nodes that match elements in $A - B$. Dye *red* nodes that match elements in $B - A$. Dye *yellow* nodes that match elements in $A \cap B$. Nodes whose final colors are red, blue, white or yellow are trivial nodes. A blue node corresponds to computing S *op* $\varnothing$. A red node corresponds to computing $\varnothing$ *op* S. A white node corresponds to computing $\varnothing$ *op* $\varnothing$. A yellow node corresponds to computing S *op* S. Since this set representation provides unique representations, we can perform a constant-time equality check using the methods discussed in Chapter 4 and computations corresponding to yellow nodes take constant time to compute.

Dyes			Color
—	—	—	white
red	—	—	red
—	*blue*	—	blue
—	—	*yellow*	yellow
red	*blue*	—	purple
red	—	*yellow*	orange
—	*blue*	*yellow*	green
red	*blue*	*yellow*	brown

TABLE 8.2 – Results of mixing dyes.

A computation will be associated with a trivial node only if the parent of the trivial node is non-trivial. The cost of the computing A *op* B is therefore proportional to the number of non-trivial nodes, which is equal to the number of nodes dyed with more than one tint.

Reassign the colors to the sets $A - B$, $B - A$, $A \cap B$: Let *yellowSet*, *blueSet* and *redSet* be the smallest, middle and largest of the sets $A - B$, $B - A$, $A \cap B$. For each set, dye the matching nodes in the tree the appropriate colors. The resulting tree looks like Figure 8.4.

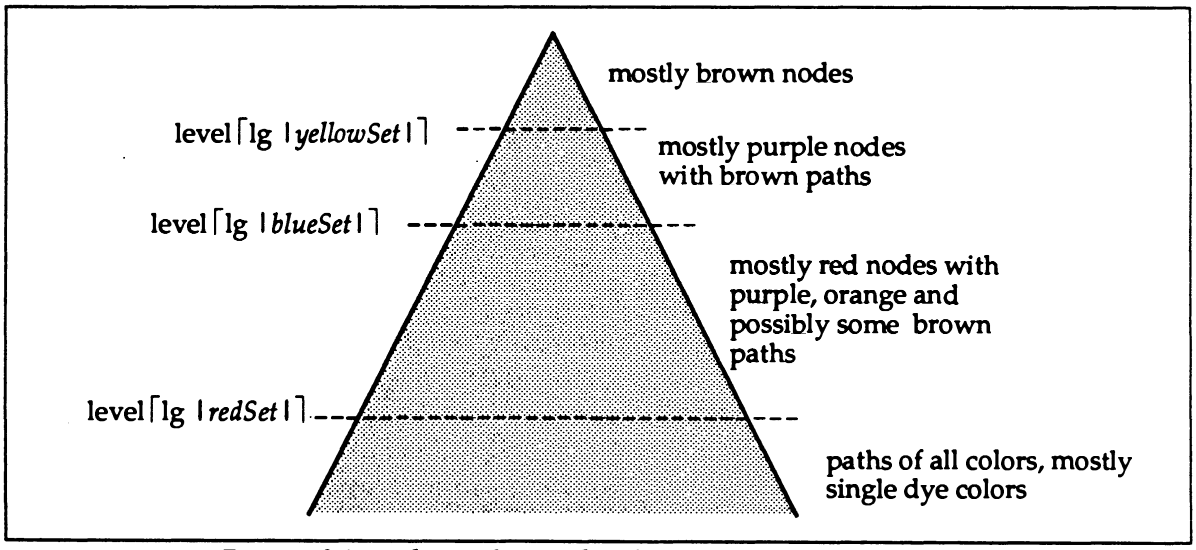

FIGURE 8.4 – Coloring that results when nodes are dyed as described.

We want to count the number of multi-dye nodes.

number of multi-dye nodes
$\le$ number of multi-dye nodes at level $\lceil \lg |blueSet| \rceil$ or higher
    + number of nodes dyed *yellow* also dyed *red* or *blue*
        (i.e., green, orange and brown nodes) at a level deeper than $\lceil \lg |blueSet| \rceil$
    + number of nodes dyed both *red* and *blue*
        (i.e., purple or brown nodes) at a level deeper than $\lceil \lg |blueSet| \rceil$

$\leq$ number of nodes at level $\lceil \lg \ |\mathit{blueSet}| \rceil$ or higher
$+ \mathit{Overlap}(\mathit{yellowSet}, \mathit{blueSet} \cup \mathit{redSet}, \lceil \lg \ |\mathit{blueSet}| \rceil)$
$+ \mathit{Overlap}(\mathit{blueSet}, \mathit{redSet}, \lceil \lg \ |\mathit{blueSet}| \rceil)$

Since the hash keys of the elements in *yellowSet* and of the elements in *blueSet* are independent, the negative binomial distributions can be summed.

number of multi-dye nodes

$$\begin{aligned}
&\leq_{prob} 4\,|\mathit{blueSet}| \\
&\quad + |\mathit{yellowSet}|\,(\lceil \lg\ |\mathit{blueSet} \cup \mathit{redSet}|\rceil - \lceil \lg\ |\mathit{blueSet}|\rceil) \\
&\qquad + NB(|\mathit{yellowSet}|) \\
&\quad + |\mathit{blueSet}|\,(\lceil \lg\ |\mathit{redSet}|\rceil - \lceil \lg\ |\mathit{blueSet}|\rceil) \\
&\qquad + NB(|\mathit{blueSet}|) \\
&\leq_{prob} 4\,|\mathit{blueSet}| + |\mathit{yellowSet}| \\
&\quad + (|\mathit{blueSet}| + |\mathit{yellowSet}|)(\lceil \lg\ |\mathit{redSet}|\rceil - \lceil \lg\ |\mathit{blueSet}|\rceil) \\
&\quad + NB(|\mathit{blueSet}| + |\mathit{yellowSet}|). \square
\end{aligned}$$

Theorem 8.3. Let S_1, S_2, S_3 be the smallest, middle and largest of the sets $A - B$, $B - A$, $A \cap B$. The time required to perform set union is bounded by $O_{prob}(\ |S_2|\ (1 + \log(|S_3| / |S_2|))\ )$.
Proof. The number of non-trivial computations is $\leq_{prob} 4\,|S_2| + |S_1| + (|S_2| + |S_1|)(\lceil \lg |S_3| \rceil - \lceil \lg |S_2| \rceil) + NB(|S_2| + |S_1|)$ (as discussed above). Result follows from Theorem 5.1. □

8.2.5 Decomposition stability

In this section we show the decomposition scheme we have defined is stable.

Theorem 8.4. A binary hash trie representation for sets is $O_{prob}(\log n)$ stable for (*addElement*, *deleteElement*) transformations.

Proof. Let $\Delta S = (S' - S) \cup (S - S')$. Clearly, $tDistance_{\{addElement,\, deleteElement\}}(S, S') = |\Delta S|$. In a hash-space tree, dye nodes matching elements in $S' \cap S$ *red*, $S' - S$ *blue*, and $S - S'$ *yellow*. A node labeled L in the hash-space tree corresponds to a value annotated with L that may appear in the decomposition of an annotated sets. Nodes colored red correspond to values that, if they appear, appear in the decomposition of both $\langle S', \varepsilon \rangle$ and $\langle S, \varepsilon \rangle$. The values that may appear in the decomposition of $\langle S', \varepsilon \rangle$ but not of $\langle S, \varepsilon \rangle$ correspond to either purple, orange or brown nodes or to blue or green nodes that correspond to values appearing in the decomposition of $\langle S', \varepsilon \rangle$.

Using techniques described previous, we obtain the bounds that the number of purple or brown nodes $\leq_{prob}$ $\mathit{Overlap}(S' - S, S' \cap S, 0)$ and the number of orange or brown nodes $\leq_{prob}$ $\mathit{Overlap}(S - S', S' \cap S, 0)$.

Let n be a blue or green node that corresponds to a value that appears in the decomposition of $\langle S', \varepsilon \rangle$. Since the only elements matching n from the set S' are in $S' - S$, n must also be a node in a representation of the set $\langle S' - S, \varepsilon \rangle$. As is shown in Theorem 8.4, below the number of interior or leaf nodes that would appear in a representation of the set $\langle S' - S, \varepsilon \rangle$ is $\leq_{prob} 8\,|S' - S| + 2\,NB(|S' - S|)$. Therefore, the number of blue or green nodes that correspond to interior or leaf nodes of the representation of $\langle S', \varepsilon \rangle \leq_{prob} 1 + 8\,|S' - S| + 2\,NB(|S' - S|)$.

Therefore,

$$\begin{aligned}
&dDistance(\langle S, \varepsilon \rangle, \langle S', \varepsilon \rangle) \\
&\leq_{prob} 1 + |\Delta S|\ \lg |S' \cap S| + 8\,|S' - S| + 3\,NB(|S' - S|) + NB(|S - S'|),
\end{aligned}$$

which is

$O_{prob}(|\Delta S| + |\Delta S| \log |S' \cap S|)$. □

Theorem 8.5. The number of nodes in the trie representing a set S is $\leq_{prob} 1 + 8|S| + 2\,NB(|S|)$.
Proof. The interior nodes of the trie representing S correspond to nodes in the hash-space trie that match more than one element in S. The number of nodes in the trie of S that are at level $\lceil \lg |S| \rceil$ or higher is less than $4|S|$. For each element x in S, the number of nodes in the hash-space tree at level $\lceil \lg |S| \rceil + 1$ or deeper that match both x and another element in S is $\leq_{prob} NB(1)$. These results can be summed, giving the result that the number of interior nodes in the trie representing of S that are deeper than level $\lceil \lg |S| \rceil$ is $\leq_{prob} NB(|S|)$. The number of leaf nodes in a trie is equal to one plus the number of interior nodes. □

8.2.6 Incremental performance of set operations

The simple fact that our decomposition scheme is stable could be used to show that we can use binary hash tries to get fast incremental set operations. It turns out that we can do even better; this section obtains bounds on the incremental performance of set operations that are better than could be obtained using the techniques of Theorem 6.1.

Theorem 8.6. Assume the results of computing $x \in S$ are stored in the cache. Define ΔS to be $(S' - S) \cup (S - S')$. The time required to determine if $x \in S'$ is bounded by $O_{prob}(\log |\Delta S|)$ if $x \notin \Delta S$ and by $O_{prob}(\log |S'|)$ otherwise.
Proof. We can compute $x \in S'$ as $\{x\} \subseteq S'$. Theorem 8.4 gives the result that the time to compute $x \in S'$ is $O_{prob}(\log |S'|)$. Assume $x \notin \Delta S$. Dye the nodes that match x *red* and dye nodes that match elements in ΔS *blue*. The only nodes that might correspond to non-trivial computations are nodes that are colored either red or purple. Any non-trivial computations corresponding to red nodes can be reused from the cache. The number of non-trivial computations required to compute $x \in S$ is equal to the number of purple nodes, which is calculated as *Overlap*$(\{x\}, \Delta S, 0)$. Result follows from Lemma 8.2 and Theorem 5.1. □

Question. Assume the results from the computations involved in computing A *op* B are stored in the function cache. How much time is required to compute A' *op* B' (where A' is similar to A and B' is similar to B)?

Let *yellowSet*, *blueSet* and *redSet* be the smallest, middle and largest of the sets $A' - B'$, $B' - A'$, $A' \cap B'$. Define $\Delta A = (A' - A) \cup (A - A')$, $\Delta B = (B' - B) \cup (B - B')$, and $\Delta Both = \Delta A \cup \Delta B$. Dye *polka-dot* the nodes that match an element in $\Delta Both$. Polka-dot green, orange, purple and brown nodes correspond to non-trivial computations that cannot be reused from the previous computation. The computations corresponding to solid colored nodes can be reused. The time required to compute A' *op* B' is proportional to the number of polka-dot green, orange, purple and brown nodes. However, the requirement that the arguments to the *Overlap* function be disjoint sets means we cannot use the techniques previously discussed to count the number of polka-dot green, orange, purple and brown nodes.

Undo the previous decision to dye all the nodes matching elements in $\Delta Both$ polka-dot. Instead:

Dye *polka-dot* nodes that match an element in $\Delta Both - (yellowSet \cup blueSet)$.
Dye *striped* nodes that match an element in $\Delta Both \cap blueSet$.
Dye *paisley* nodes that match an element in $\Delta Both \cap yellowSet$.
(e.g., *red* dye + *blue* dye + *polka-dot* dye + *paisley* dye yields purple paisley polka-dots)

A node that is dyed *polka-dot, striped* and/or *paisley* is patterned, otherwise it is colored a solid color. A node that has been dyed with more than one of the dyes *red, blue* or *yellow* (i.e., a green, orange, purple or brown node) is a multi-tint node. The time required to compute $A' \cup B'$ is proportional to the number of patterned multi-tint nodes.

A multi-tint node must be dyed with either *blue* or *yellow* dye, so the set of *polka-dot* dyed, multi-tint nodes is a subset of the set of *polka-dot* and (*yellow* and/or *blue*) dyed nodes. Since the set $\Delta Both - (yellowSet \cup blueSet)$ is disjoint from the set $yellowSet \cup blueSet$, we can use the *Overlap* function to count the number of nodes dyed both *polka-dot* and (*yellow* and/or *blue*), giving us:

$$\begin{aligned}&\text{the number of of } \textit{polka-dot} \text{ dyed, multi-tint nodes deeper than level } k\\ &\quad \leq_{prob} Overlap(\Delta Both - (yellowSet \cup blueSet), yellowSet \cup blueSet, k).\end{aligned}$$

Similarly,

$$\begin{aligned}&\text{the number of of } \textit{striped} \text{ dyed, multi-tint nodes deeper than level } k\\ &\quad \leq_{prob} Overlap(\Delta Both \cap blueSet, yellowSet \cup redSet, k)\end{aligned}$$

and

$$\begin{aligned}&\text{the number of of } \textit{paisley} \text{ dyed, multi-tint nodes deeper than level } k\\ &\quad \leq_{prob} Overlap(\Delta Both \cap yellowSet, blueSet \cup redSet, k).\end{aligned}$$

We can now count the total number of patterned, multi-tint nodes:

$$\begin{aligned}&\text{\# of patterned multi-dye nodes}\\ &\leq_{prob} \text{\# of patterned multi-dye nodes at or above level } \lceil \lg |\Delta Both| \rceil\\ &\quad + \text{\# of polka-dot multi-dye nodes below level } \lceil \lg |\Delta Both| \rceil\\ &\quad + \text{\# of striped multi-dye nodes below level } \lceil \lg |\Delta Both| \rceil\\ &\quad + \text{\# of paisley multi-dye nodes below level } \lceil \lg |\Delta Both| \rceil\\ &\leq_{prob} \text{\# of nodes at or above level } \lceil \lg |\Delta Both| \rceil\\ &\quad + Overlap(\Delta Both - (yellowSet \cup blueSet),\\ &\qquad\qquad yellowSet \cup blueSet, \lceil \lg |\Delta Both| \rceil)\\ &\quad + Overlap(\Delta Both \cap blueSet, yellowSet \cup redSet, \lceil \lg |\Delta Both| \rceil)\\ &\quad + Overlap(\Delta Both \cap yellowSet, blueSet \cup redSet, \lceil \lg |\Delta Both| \rceil)\end{aligned}$$

We want to be able to sum the results of these three *Overlap* functions. To do this, we must assume $|\Delta Both - (yellowSet \cup blueSet)| \leq |yellowSet \cup blueSet|$, $|\Delta Both \cap blueSet| \leq |yellowSet \cup redSet|$ and $|\Delta Both \cap yellowSet| \leq |blueSet \cup redSet|$. These assumptions, along with the fact that the sets $\Delta Both - (yellowSet \cup blueSet)$, $\Delta Both \cap blueSet$ and $\Delta Both \cap yellowSet$ are all disjoint, imply that the results of the three *Overlap* functions are independent, which allows us to combine the results.

$$\begin{aligned}&\text{\# of patterned multi-dye nodes}\\ &\leq_{prob} 4\,|\Delta Both|\\ &\quad + |\Delta Both - (yellowSet \cup blueSet)|\\ &\qquad\qquad (\lceil \lg |yellowSet \cup blueSet| \rceil - \lceil \lg |\Delta Both| \rceil)\\ &\qquad + NB(|\Delta Both - (yellowSet \cup blueSet)|)\\ &\quad + |\Delta Both \cap blueSet|\,(\lceil \lg |yellowSet \cup redSet| \rceil - \lceil \lg |\Delta Both| \rceil)\\ &\qquad + NB(|\Delta Both \cap blueSet|)\end{aligned}$$

$$
\begin{aligned}
&+ \ |\Delta Both \cap yellowSet| \ (\lceil \lg \ |blueSet \cup redSet| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&\quad + NB(\ |\Delta Both \cap yellowSet| \ )
\end{aligned}
$$

$$
\begin{aligned}
\leq_{prob} &\ 4 \ |\Delta Both| \\
&+ \ |\Delta Both - (yellowSet \cup blueSet)| \\
&\quad (\lceil \lg \ |yellowSet \cup blueSet| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&+ \ |\Delta Both \cap blueSet| \ (\lceil \lg \ |yellowSet \cup redSet| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&+ \ |\Delta Both \cap yellowSet| \ (\lceil \lg \ |blueSet \cup redSet| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&+ NB(\ |\Delta Both| \ )
\end{aligned}
$$

Since $2\,|blueSet| \geq |blueSet \cup yellowSet|$, $2\,|redSet| \geq |redSet \cup yellowSet|$ and $2\,|redSet| \geq |redSet \cup blueSet|$, we can simplify this to

$$
\begin{aligned}
&\#\text{ of patterned multi-dye nodes} \\
&\leq_{prob} 5\,|\Delta Both| \\
&\quad + \ |\Delta Both - (yellowSet \cup blueSet)| \ (\lceil \lg (\,|blueSet| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&\quad + \ |\Delta Both \cap (yellowSet \cup blueSet)| \ (\lceil \lg (\,|redSet| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&\quad + NB(\ |\Delta Both| \ ). \square
\end{aligned}
$$

Theorem 8.7. Let A, B, A' and B' be sets. Assume the results from the computations involved in computing A *op* B are stored in the function cache. Let S_1, S_2 and S_3 be the smallest, middle and largest of $A' - B'$, $B' - A'$, $A' \cap B'$. Let the name $\Delta Both$ refer to $(A' - A) \cup (A - A') \cup (B' - B) \cup (B - B')$. Assume $|\Delta Both - (S_1 \cup S_2)| \leq |S_1 \cup S_2|$, $|\Delta Both \cap S_2| \leq |S_1 \cup S_3|$ and $|\Delta Both \cap S_1| \leq |S_2 \cup S_3|$. The time required to compute A' *op* B' is

$$
\begin{aligned}
O_{prob}(\ &|\Delta Both - (S_1 \cup S_2)| \ \log(|S_2| \,/\, |\Delta Both|) \\
&+ \ |\Delta Both \cap (S_1 \cup S_2)| \ \log (|S_3| \,/\, |\Delta Both|)).
\end{aligned}
$$

Proof. The time required to compute A' *op* B' is proportional to the number of non-trivial computations not available from cache. This is shown above to be at most

$$
\begin{aligned}
&5\,|\Delta Both| \\
&\quad + \ |\Delta Both - (S_1 \cup S_2)| \ (\lceil \lg (\,|S_2| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&\quad + \ |\Delta Both \cap (S_1 \cup S_2)| \ (\lceil \lg (\,|S_3| \rceil - \lceil \lg |\Delta Both| \rceil) \\
&\quad + NB(\ |\Delta Both| \ )
\end{aligned}
$$

Result follows from Theorem 5.1. $\square$

The sets $\Delta Both - (S_1 \cup S_2)$, $\Delta Both \cap S_2$ and $\Delta Both \cap S_1$ are required to be small so that the results of the three overlap functions are independent. Even if these sets are not small, the *expected* time required to compute $A' \cup B'$ is still bounded by

$$
\begin{aligned}
O(\ &|\Delta Both - (S_1 \cup S_2)| \ \log(|S_2| \,/\, |\Delta Both|) \\
&+ \ |\Delta Both \cap (S_1 \cup S_2)| \ \log (|S_3| \,/\, |\Delta Both|)).
\end{aligned}
$$

8.3 Relationship to previous work

This set representation has many interesting properties and improves on the time bounds for set operations on similar sets. For two sets A and B satisfying $|A| \leq |B|$, all previous algorithms for non-disjoint sets requires at least $O(|A|)$ to compute A *op* B. If A and B differ in only k elements, the algorithms presented here require only $O(k + k \log(|A \cap B|/k))$ time. This set representation also supports constant-time equality tests. The only previous set representation that supported constant-time equality tests [SG76] had inefficient set construction operations; any operation that computed a new set S required at least $O(|S|)$ time to perform.

Pardo [Pardo 78] suggested an equivalent data structure for representing sets. However, his analysis was much more complicated and achieved poorer results. The bounds he obtains on the expected number of operations required to compute $A \cap B$ are

$$|A| \lg(1 + \frac{|B|}{|A|}) + |B| \lg(1 + \frac{|A|}{|B|})$$
$$+ |A \cap B| (\frac{2}{\ln 2} + \lg \frac{|A||B|}{|A \cap B|^2}).$$

Our algorithms gives better results primarily because our algorithms take advantage of the fact that (A *op* A) is a trivial computation. Pardo failed to recognize that this set representation provides unique representations and allows constant-time equality tests and did not analyze the probability distribution of the time performance of the set algorithms.

Our analysis was simpler primarily because we started from the premise that we were going to calculate a probabilistic upper bound and our analysis involved doing an exact analysis of an upper bound. Pardo analyzed set operations by solving a recurrence relation, the standard technique for analyzing divide and conquer algorithms. However, this method is of dubious wisdom for probabilistic algorithms. In order to compare our analysis techniques with those of Pardo, we briefly present the technique he used. Define $T(a, b)$ to be the average time required to compute $A \cup B$, where A and B are disjoint sets represented as binary hash tries such that $a = |A|$ and $b = |B|$. The algorithm can terminate if either set is empty, so $T(a, 0) = T(0, b) = c_1$. Otherwise, the algorithm uses a divide and conquer step. Define $P(a, a')$ to be the probability that a set of size a is divided into a set a size a' and a set of size a-a' by this divide and conquer step. We can calculate

$$P(a, a') = \binom{a}{a'} 2^{-a}.$$

We therefore get the recurrence relation

$$\begin{aligned} T(a, b) &= c_2 + \sum_{a'=0}^{a} \sum_{b'=0}^{b} P(a, a')\, P(b, b')\, (T(a', b') + T(a - a', b - b')) \\ &= c_2 + \frac{1}{2^{a+b}} \sum_{a'=0}^{a} \sum_{b'=0}^{b} \binom{a}{a'} \binom{b}{b'} (T(a', b') + T(a - a', b - b')). \end{aligned}$$

Since $P(a, a') = P(a, a - a')$, this can be simplified to

$$T(a, b) = c_2 + \frac{2}{2^{a+b}} \sum_{a'=0}^{a} \sum_{b'=0}^{b} \binom{a}{a'} \binom{b}{b'} T(a', b').$$

Deriving an exact closed form for this function appears impossible. Pardo obtains an asymptotic upper bound, but doing so is a very complex process, one that we feel is much more complicated than the analysis used to obtain the results of Theorem 8.2 of this chapter. It is unclear if it would be feasible to extend Pardo's analysis method to other problems we examined in this chapter such as the probability distribution of running times, the stability of binary hash tries and the incremental performance of set operations using binary hash tries.

8.4 Using binary hash tries to implement other abstract data types

Binary hash tries can be used to implement a number of other abstract data types. This section will briefly describes these data types and the performance of operation as implemented using binary hash tries. The algorithms and analysis for these other implementations can easily be extrapolated from the described for sets.

8.4.1 Bags

Bags are similar to sets except that they can contain more than one instance of a single element. Incremental data structures for bags can be built similarly to those for sets, simply by using a set of atomic routines that handle bags rather than sets. The performance and stability of bags is the same as that of sets.

8.4.2 Finite functions

Finite functions are functions that contain a finite number of mappings from elements from the domain of the function to the range of the function. Typically, finite functions are used for symbol tables or similar data structures that map from a name or an identifier to a value. When applied to an element that does not have a specific mapping, a predefined bottom element ($\perp$) is returned. The operations supported on finite functions are listed in Table 8.3.

operation	*notes*
empty	The function that maps all values to $\perp$
$f(x)$	The value to which f maps x
$(f; i{:}v)$	The finite function g such that $g(x) \equiv$ *if* $x = i$ *then* v *else* $f(x)$
update(f, g)	The finite function h such that $h(x) \equiv$ *if* $g(x) \neq \perp$ *then* $g(x)$ *else* $f(x)$
$f - i$	The finite function g such that $g(x) \equiv$ *if* $x = i$ *then* $\perp$ *else* $f(x)$
project(f, s)	The finite function h such that $h(x) \equiv$ *if* $x \in s$ *then* $f(x)$ *else* $\perp$

TABLE 8.3 – Operations that can be supported on finite functions.

Implementation of finite function operations

Finite functions are implemented as tagged sets. The finite function is represented as a set of those elements that have non-bottom mappings and each element is tagged with the value to which it is mapped. Table 8.4 shows the correspondence from finite function operations to set operations.

operation	*correspondence*	*notes*
empty	$\varnothing$	
$f(x)$	$x \in f$	Return tag of x if found, otherwise return $\perp$
$f[i \to v]$	$f \cup \{i\}$	If $i \in f$, change its tag to v otherwise add i tagged with v to f
update(f, g)	$f \cup g$	If an element is in both f and g, use tag from g
$f - i$	$f - \{i\}$	
project(f, s)	$f \cap s$	

TABLE 8.4 – Implementation of finite function operations

8.4.3 Priority queues

Priority queues are a method of maintaining an ordered set. One way of handling an ordered set is to use a sorting algorithm to return a sorted list, using the methods suggested in Chapter 7 for representing sequences. This is the preferred method when the sorted list will be decomposed by a divide and conquer algorithm. However, there are some situations in which the sorted elements will be consumed in an inherently sequential manner. The overhead of the methods discussed for chunky lists are not severe, but in this situation there are more appropriate data structures. Several different interfaces can be described for priority queues. The simplest and most general technique is to say that priority queues are similar to finite functions; if the element x is in a priority queue q, $q(x)$ returns the priority of x. We implement an additional operation $max(q)$ that returns the element in q with the highest priority.

9 Other Approaches to Incremental Computation

9.1 Incremental Constraint Solving

Incremental constraint solving, while in some ways similar to incremental evaluation, involves a basic philosophical difference. In a constraint system, a set of constraints are given that must be maintained. For example, two values F and C might be constrained so that $F = (9C/5) + 32$ — if C represents a temperature in Celsius, F represents that same temperature in Fahrenheit. In a particular situation, F might currently be 32 and C would be 0. If the user changed F to be 50, C would be updated by the system to be 10.

Constraint systems can involve tens or hundreds of constraints. When a variable is changed, an incremental constraint solver attempts to update the values of other variables so as to re-satisfy the constraints. Some systems attempt to find a solution that minimizes the "distance" between the previous solution and the new solution. However, this can require looking at the entire constraint system and in fact is an NP-complete problem [Van88]. Other work has attempted to balance the desire for a solution that is similar to the previous solution and the desire to quickly find the new solution [Van88].

The major philosophical difference between incremental evaluation and incremental constraint solving is that constraints systems are under-defined - that is, there are many possible solutions to the constraints, and the order in which changes are made will make a difference in the final result. Making a change x followed by a change y will not produce the same result as making change y followed by change x, while it is guaranteed to produce the result in an incremental evaluator.

The difference is largely due to a different concept of "input" and "output". In a incremental evaluation system, there is an input which can be changed by the user and an output which is the result of applying a computation to that input. In a constraint system, there is a set of variables which both the user and the constraint system are allowed to manipulate. Thus, it does not make sense to talk of a constraint system maintaining a specific relation between the "input" and the "output", since "input" and "output" are the same thing to a constraint system.

9.2 Supply-driven evaluation

In Bengelloun's thesis [Ben82], he describes an incremental evaluator for functional programs. However, the approach he pursues is substantially different from the ideas presented in this thesis. His approach is essentially a supply-driven version of lazy evaluation. In lazy evaluation, the amount of input demanded is based on what is needed to compute the output. With supply-driven evaluation, the output that appears is dependent on the amount of input that has been made available.

Values can be either standard values or they can be mutable values. A mutable value has a current value, but is subject to change. Data structures are maintained so that when a mutable value changes, the outputs can be updated correctly.

Although, in theory, Bengelloun's evaluator could be applied to any problem, it is only intended and efficient for situations where the input is monotonically increasing. The typical kind of change supported by Bengelloun's evaluator is to replace a *"not known yet"* value with a new value, which may include other values that are *"not known yet"*. The only example Bengelloun gives of a program that works well with his evaluator is a primal sieve; given a list of the integers two through n, it returns a list of the primes two through n. For this situation, the only incremental change that is supported is to add an additional element to the end of the input.

9.3 Incremental Attribute Grammar and Dependency Graph Evaluation

Chapter 2 discusses some reasons why incremental dependency graph evaluation schemes are limited to certain applications; function caching does not suffer from the same limitations and there are many applications for which function caching is much more suitable. Can we compare function caching against incremental dependency graph evaluation on those applications for which incremental dependency graph evaluation does work well?

We would like to have a formal evaluation of these two approaches. However, graph evaluation and functional programs are sufficiently different that it is difficult to make any sort of direct comparison. For the case of attribute grammars however, we can make a direct comparison. Katayma has described a method for translating an attribute grammar G into a set of recursive functions $F(G)$ [Kat84]. We can then compare the incremental performance of using Reps' OPTIMAL attribute grammar propagation scheme on G with the incremental performance of using function caching on $F(G)$. The material in this section assumes the reader is somewhat familiar with attribute grammars and with Reps' OPTIMAL attribute grammar evaluation scheme [Reps82].

The process of converting an attribute grammar into a set of recursive functions is fairly straight forward. Let N be a nonterminal in a grammar G. For each synthesized attribute s of N, we define a corresponding function s_N. Let r be a node of type N in a tree T. Let T_r be the subtree rooted at r. The function s_N is defined such that in a consistently attributed tree, $r.s = s_N(T_r, r.i_1, r.i_2, r.i_3 ..., r.i_m)$, where the set $\{i_1, i_2, i_3 ... i_m\}$ is the set of inherited attributes of r that s might possibly depend on (*i.e.*, $\{i_1, i_2, i_3 ... i_m\} = \{ i \in I(N) \mid (i, s) \text{ is an edge of } IO[N] \}$).

The process of translating an attribute grammar is fairly easy and can be mechanized [Kat84], although we will not go into the details of the translation process here. Because we plan to make use of function caching, we can avoid some of the complexities introduced by Katayama. Figure 9.1 shows the standard example of using an attribute grammar to express the semantics of binary integers and the corresponding recursive functions. The recursive functions are shown with a syntax that is similar to that of PROLOG in order to make the correspondence between the attribution rules and the functions more obvious. Some additional mechanical manipulation can combine the multiple definitions given here for each s_N into a single function with a case statement. Efficient evaluation of the function program produced relies on the use of function caching.

Comparing function caching with incremental attribute grammar evaluation

Theorem 9.1. Let T be a tree consistently attributed according to an attribute grammar G. Define the set *Affected* to be the set of attributes that receive a new value as a result of a subtree replacement at a node *new* (as in Reps's discussion). Define *path_to_root* to be the set of nodes in T that are an ancestor of *new* (define *path_to_root* so that it include *new*). Let *Affected_Applications* be the set of functions applications that need to be computed and will not be found in the cache when using function caching and $F(G)$. Then,

$$|Affected\_Applications| \text{ is } O(|Affected| + |path\_to\_root|).$$

Proof. Define the set *Affected_I_Nodes* to be the set of nodes r in T such that an inherited attribute of r receives a new value as a result of a change. Since each attribute in an inherited attribute of only one node, $|Affected\_I\_Nodes|$ *is* $O(|Affected|)$.

Define *Needed(T)* to be the set of function calls needed to evaluate the function calls from $F(G)$ corresponding to synthesized attributes of the root of T. For a function call f, let $root(f)$ denote be the root of the subtree that is the first argument of f (e.g., if f is a function call whose first argument is T_r, $root(f) = r$). Since the number of synthesized attributes at a node is bounded by a constant based on the size of the grammar, for all nodes r in T, $|\{f \mid f \in Needed(T) \wedge root(f) = r\}|$ is bounded by a constant.

In a function caching implementation, we cannot perform destructive editing operations. To perform an edit operation on a tree T, we create a new tree T' and evaluate the functions corresponding to the synthesized attributes of the root of T'. The nodes that must be copied are those nodes in *path_to_root*.

The only functions that need to be computed are those that were not computed previously. Therefore, $Affected\_Applications \subseteq \{f \mid f \in Needed(T') \wedge root(f) \in Affected\_I\_Nodes \cup path\_to\_root\}$. Therefore, $|Affected\_Applications|$ *is* $O(|Affected\_I\_Nodes| + |path\_to\_root|)$ which is $O(|Affected| + |path\_to\_root|)$. □

---

p_0: *Number* → *Integer*

Number.value = Integer.value

Integer.scale = 0

| $\text{value}_{\text{Number}}(p_0[\text{Integer}]) = \text{value}_{\text{Integer}}(\text{Integer}, 0)$

p_1: Integer → Integer Bit

$Integer_1$.value = $Integer_2$.value + Bit.value

$Integer_2$.scale = $Integer_1$.scale + 1

Bit.scale = $Integer_1$.scale

| $\text{value}_{\text{Integer}}(p_1[\text{Integer Bit}], \text{scale})$
| $= \text{value}_{\text{Integer}}(\text{Integer}, \text{scale}+1) + \text{value}_{\text{Bit}}(\text{Bit}, \text{scale})$

p_2: Integer → Bit

Integer.value = Bit.value

Bit.scale = Integer.scale

| $\text{value}_{\text{Integer}}(p_2[\text{Bit}], \text{scale}) = \text{value}_{\text{Bit}}(\text{Bit}, \text{scale})$

p_3: Bit → **0**

Bit.value = 0

| $\text{value}_{\text{Bit}}(p_3[\mathbf{0}], \text{scale}) = 0$

p_4: Bit → **1**

Bit.value = $2^{Bit.scale}$

| $\text{value}_{\text{Bit}}(\text{right}p_4[\mathbf{1}], \text{scale}) = 2^{\text{scale}}$

FIGURE 9.1 - An attribute grammar to convert a bit stream to an integer and a set of recursive functions derived from that attribute grammar

How well have we done? For very narrow and stringy trees, this would be very bad. In some situations, a change in a tree of size n would involve a length n path to root. Several points come to mind however. First, we feel that this is an argument against narrow stringy trees, not against function caching. Typically program editors have used a right recursive or left recursive format for sequences such as statement lists. The data structures we have presented in this thesis, or many other data structures, allow lists of length n to be manipulated as trees of depth $\log n$. Also, the overhead of $|path\_to\_root|$ is partially associated with the fact that the conversion process is only appropriate for attribute grammars that only produce information at the synthesized attributes of the root of the syntax tree; for such attribute grammars, function caching has the same asymptotic time bounds as attribute grammars whenever a change to the syntax tree causes a change in the output of the algorithm.

We should also note that new research in incremental dependency graph evaluation has improved and extended Reps' algorithms in new directions [HT86b] [Hoo87] [ACRSZ87] [YS88]. Even if we wanted to claim that we could do as well as Reps' original algorithm, this would not say anything about function caching verses attribute grammar or dependency graph techniques, since many new results have improved on Reps' original results.

10 Conclusions

10.1 Future research

This section outlines some of the future research suggested by this thesis.

Further explorations of stable decompositions

This thesis has provided two examples of data structures that have unique representations and stable decompositions. What other data structures can we devise that have these properties?

Caching of human decisions

One of the more active research areas in software engineering is that of software reuse: when designing or coding a software system, how can we take advantage of work that was done for previous, similar problems. If we think of the human decisions that are made in this process as results that can be cached, then the ideas presented here on stable decompositions may be applicable to the problem of software reuse.

Compiling recursive functions into attribute grammars

As we noted in Chapter 9, for those algorithms which can be efficiently implemented as attribute grammars, Reps' algorithms or other incremental attribute grammar schemes appear to be perfectly suitable methods for performing incremental evaluation and may involve less overhead. Within a single application, some computations may be expressible as attribute grammars and others may not. For example, in specifications for the Synthesizer Generator, some algorithms are written as attribute grammars and some as recursive functions. Sometimes, after an algorithm has been encoded as an attribute grammar, a situation will arise in which that same algorithm needs to be applied in a situation where an attribute grammar cannot be used, forcing the algorithm to be coded again, this time as a set of recursive functions.

What we would like would be a uniform way of expressing algorithms for incremental use. Recursive functions are more powerful than attribute grammars, so it makes sense to write all algorithms as a set of recursive functions. If a function has an equivalent attribute grammar, it would seem possible to translate those functions into a set of attribution rules. Clearly, not all functions can be so translated and the attribute grammar version of an algorithm may not be usable in all situations. However, for those situations where an attribute grammar can be used, we can make use of the research that has been done on incremental attribute grammar and dependency graph evaluation. We would have a system with a single, uniform method of expressing algorithms (recursive functions) that could be used for any problem solvable using function caching and yet would have performance as good as the best incremental attribute grammar evaluator if the algorithm was expressible as an attribute grammar.

Equivalence Classes for Function Calls

In our discussion of function caching, we have assumed that we could only reuse cache results that exactly matched the pending function call. There are several ways in which we may be able to use a weaker form of equivalence than strict equality, and thus improve the performance of the function cache. This section discusses two such equivalence classes.

Commutative functions

Commutative functions are a simple example of an equivalence class for function calls. A function f of two arguments is commutative if and only if:

$$\forall\ x, y \text{ s.t. } f(x,y) \text{ is defined, } f(x, y) = f(y, x)$$

A function of more than two arguments may be commutative in certain of its arguments. For example, a function g of three arguments is commutative in its first two arguments iff:

$$\forall\ x, y, z \text{ s.t. } g(x, y, z) \text{ is defined, } g(x, y, z) = g(y, x, z)$$

Assuming f is a commutative function, we could check the cache for both $f(x, y)$ and $f(y, x)$ each time a function call was made to $f(x, y)$. A better way would be to always store the call to a commutative function in the cache with the argument having the lower hash key first.

This optimization might not improve the incremental performance of the evaluator much, because there are few algorithms in which an incremental change would cause the order of the arguments to a function to permute. This optimization could improve the non-incremental performance of the cache.

Non-strict functions

Some functions do not require or care about all of their arguments. If a function doesn't care about all or part of an argument, it is said to be non-strict. For example, the function $f(x, y) = x$ is non-strict in y. The function $f(x, y) =$ ***if*** $x \geq 0$ ***then*** x^x ***else*** x^y ***fi*** is non-strict in y if x at least zero. The function $f(x, y) = head(x)+head(y)$ is non-strict in the tails of both x and y.

Consider the example $f(x, y) =$ ***if*** $x \geq 0$ ***then*** x^x ***else*** x^y ***fi***. Assume we compute $f(5, 25)$. We could store this in the cache as $f(5, 25) = 3125$. But what we would really like to do is store *f(5, <wildcard>)* = 3125, so that, for example, we could reuse this result when computing $f(5, 30)$.

Adding wildcards to the cache entries complicates the problem of checking for cache hits. If we use wildcards, we need to do pattern matching rather than equality checking when comparing a cache entry with a pending call.

An even more subtle problem involves generating the hash index for a pending function call. Assume a function call $f(x, y)$ is pending. How can we generate a hash index? If we combine the hash keys for f, x and y, we will only find exact matches. For the function described above, we could examine the hash bucket indexed by both *f(x, <wildcard>)* and $f(x, y)$. We could be smart and check under *f(x, <wildcard>)* *if* $x \geq 0$ and under $f(x, y)$ if $x < 0$.

What about the function $f(x, y) = head(x) + head(y)$? This is even more tricky because we can't use the hash keys of either x or y in computing a hash index.

The obvious answer seems to be to write functions so that non-strict functions don't have to be cached. For example, rather than defining $f(x, y) =$ ***if*** $x \geq 0$ ***then*** x^x ***else*** x^y ***fi*, we** should define $f(x, y) =$ ***if*** $x \geq 0$ ***then*** $h(x, x)$ ***else*** $h(x, y)$ ***fi*, *where*** $h(x, y) = x^y$ and cache the function h but not f.

What sort of situations arise in which it is not easy to rewrite the functions? One situation that often arises in programing environments involves the use of finite functions (e.g., symbol tables). Consider the function *typecheck(stmtList, env)*, which checks to see that the statement list typechecks using the type definitions provided in *env*. Since any part of the environment might be referenced in any part of the statement list, the entire environment must be passed down as the statement list is decomposed. Typically, *env* would contain all type definitions visible to the statement list. For example, *env* might contain the definition that x is of type *integer*, even though no reference to x occurred in the statement list. Assume the environment *env'* is the same as *env*, except that x is defined to be of type *real*. Unless we find some method to take into account the fact that the typecheck function is non-strict in its environment argument, we will not be able to share work between the invocations of *typecheck(stmtList, env)* and *typecheck(stmtList, env')*.

Possible solutions. It is possible to perform fast, incremental pattern matching. The function that checks if a value x matches a pattern p can be executed and cached as a recursive function (requiring only strict equality tests for cache hits). If a value x was previously matched against a pattern p and x' is k-similar to x (*i.e., decompositionDistance*$(x', x) = O(k)$), then x' can be checked against pattern p in $O(k)$ time. This still leaves the problem of how to compute the hash index. It may be possible to decide, based on some analysis of the function, to only base the cache lookup on the hash keys of a subset of the arguments. For example, for the typechecking function, if we knew that any particular statement list were unlikely to be checked against more than one environment at a time, we could base the hash index for the function call solely on the statement list.

Another possibility is to perform some kind of optimizations to eliminate expensive, non-strict functions. For example, a very smart compiler might transform the typecheck function as *typecheck(stmtList, env) = typecheck'(stmtList, project(env, referenced(stmtList)))*. In other words, first collect a list of all the variables that are referenced within the statement list and pass only the definitions of those variables down.

10.2 Summary

Although it was obvious that function caching could be used to speed execution when problems identical to previous problems were seen, previous researchers had not made systematic attempts to analyze how function caching could be used when problems that were only similar to previous problems were being solved. Our discussion in Chapter 3 clarifies how function caching can be used to solve a problem that is similar, but not identical, to a problem that was seen before.

Most previous implementations of function caching have not been tools of general use, but have only been useful for functions with highly specialized behavior such as the Fibonacci function. This has been mainly due to either the inefficiency of previous implementations or limitations put on the use of function caching. The results we describe in Chapter 4 considerably improve the performance of function caching. Benchmarks reported in that chapter show that even for complex programs not intended for function caching, function caching can provide significant real-time speed-ups.

Many of the data structures and algorithms presented in this thesis are probabilistic. In Chapters 5, 7 and 8 we present some new techniques that drastically simplify the analysis of probabilistic algorithms.

Function caching alone does not provide incremental evaluation. Function caching must be used with suitable algorithms and data structures. Chapter 6 describes and quantifies the properties that are important for incremental evaluation and discusses their importance. Chapter 7 presents an efficient representation for sequences that also provides constant-time equality tests and stable decompositions. Chapter 8 presents an efficient representation and algorithms for sets that offer constant-time equality tests and stable decompositions and are faster any previous algorithms for set operations on two similar sets. Both of these results are significant improvements over previous data structures that supported constant-time equality tests, and the techniques used to analyze these data are substantially simpler than previous techniques used for probabilistic algorithms and data structures.

The techniques used in Chapters 7 and 8 suggest a methodology for developing appropriate data structures. To allow data structures to be decomposed, they must be balanced. Data structures that are balanced by balance or positional information tend to be too sensitive to changes to be stable. Therefore, we must use data structures that are balanced probabilistically. But since the data structures must have unique representations, we must use a deterministic source of randomness: the hash keys of elements.

The methods presented in this thesis can provide incremental evaluation for a wide variety of problems. Function-caching based incremental evaluation reuses the results of any similar problem solved previously; incremental dependency graph evaluation updates the results of a specific previous problem. Because of this fundamental difference, these two paradigms are applicable in different degrees to different problems. These two ideas can be profitably combined by using whichever method is most appropriate for a particular part of a computation.

Appendix – Negative Binomial Distributions

This appendix includes a number of results related to negative binomial distributions. We first introduce some lemmas and a theorem so that we can obtain a simple, closed-form upper bound on cumulative negative binomial distributions.

Lemma A.1: Let p be a probability, s be a positive integer and M be at least 1.

$$\text{Prob}\{\,NB(s,p) > (M-1)\,s\,\} < \frac{1-p}{p}\frac{Mp}{Mp-1}\,\text{Prob}\{\,NB(s,p) = (M-1)\,s\,\}$$

Proof. Let P_k stand for Prob{ $NB(s,p) = k$ }. Define M such that $Ms = k{+}s$, or $M{=}k/s{+}1$. By definition,

$$P_k = \binom{k+s-1}{s-1}\,p^s\,(1-p)^k.$$

We can easily derive

$$P_{k+1} = \frac{k+s}{k+1}\,(1-p)\,P_k.$$

Since for all i, $i > k$, $(i{+}s)/i < (k{+}s)/k$,

$$\forall\, i \text{ s.t. } i > k,\ P_{i+1} < \frac{k+s}{k}\,(1-p)\,P_i.$$

We can therefore get a bound on Prob{ $NB(s,p) > k$ }:

$$\text{Prob}\{\,NB(s,p) > k\,\} = \sum_{i=k+1}^{\infty} P_i$$
$$< P_k\,(r + r^2 + r^3 + \dots\dots), \text{ where } r = \frac{k+s}{k}\,(1-p).$$

This simplifies to

$$Prob\{\,NB(s,p) > k\,\} < P_k\left(\frac{1}{1-\frac{k+s}{k}(1-p)} - 1\right)$$

Define M such that $Ms = k{+}s$, or $M = k/s{+}1$. We can then simplify as follows.

$$\text{Prob}\{\,NB(s,p) > (M-1)\,s\,\} < P_{(M-1)\,s}\left(\frac{1}{1-\frac{Ms}{Ms-s}(1-p)} - 1\right)$$
$$< P_{(M-1)\,s}\,\frac{1-p}{p}\,\frac{Mp}{Mp-1}$$
$$< \frac{1-p}{p}\,\frac{Mp}{Mp-1}\,\text{Prob}\{\,NB(s,p) = (M-1)\,s\,\}.\ \square$$

Lemma A.2. Let p be a probability, s a positive integer and M at least 1.

$$\text{Prob}\{\,NB(s,p) = (M-1)\,s\,\} \le (e\,M\,max(p,1-p)^M)^s$$

Proof. Let P_k stand for Prob{ $NB(s,p) = k$ }.

$$P_{(M-1)s} = \binom{Ms-1}{s-1} p^s (1-p)^{Ms-s}$$

$$P_{(M-1)s} \le \binom{Ms}{s} p^s (1-p)^{Ms-s}$$

$$< \frac{(Ms)^s}{(s)!} p^s (1-p)^{Ms-s}$$

$$< \frac{(Ms)^s}{e^{-s} s^s \sqrt{2\pi s}} p^s (1-p)^{Ms-s} \qquad \textit{(by Stirling's formula)}$$

$$< (e\,M)^s p^s (1-p)^{Ms-s}$$

$$< (e\,M\,max(p, 1-p)^M)^s. \square$$

Theorem A.1. Let p be a probability, s a non-negative integer and M at least 1.

$$\text{Prob}\{\, NB(s,p) > (M-1)\,s \,\} < \frac{1-p}{p} \frac{Mp}{Mp-1} (e\,M\,max(p, 1-p)^M)^s$$

This means that Prob{ $NB(s,p) > (M-1)\,s$ } decreases almost exponentially as M is increased. If $(e\,M\,max(p, 1-p)^M) < 1$, then Prob{ $NB(s,p) > (M-1)\,s$ } decreases exponentially as s is increased.

Proof. Follows directly from Lemma A.1 and Lemma A.2. □

We now prove two theorems about sequences of random variables, where each variable is bounded by a negative binomial distribution.

Theorem A.2. Let p be a probability, and $X_0, X_1, \ldots, X_{n-1}$ be independent random variables, each bounded by $NB(1,p)$.

$$\max(X_0, X_1, \ldots, X_{n-1}) \le_{prob} \log_{1/(1-p)} n + NB(1,p)$$

Proof. For each X_i, Prob{ $X_i \ge k$ } = $(1-p)^k$. Combining these probabilities, we get:

$$\text{Prob}\{\max(X_1, X_2, \ldots, X_n) \ge k\} = 1 - (1 - (1-p)^k)^n.$$

Expanding this gives

$$\text{Prob}\{\max(X_0, X_1, \ldots, X_{n-1}) \ge k\} = 1 - (1 - n(1-p)^k + n(n-1)(1-p)^{2k}/2 - \ldots)$$
$$= n(1-p)^k - n(n-1)(1-p)^{2k}/2 + \ldots$$
$$\le n(1-p)^k.$$

Let $j = k - \log_{1/(1-p)} n$. Substituting for k, we get

$$\text{Prob}\{\max(X_1, X_2, \ldots, X_n) - \log_{1/(1-p)} n \ge j\} \le (1-p)^j.$$

Since $(1-p)^j = \text{Prob}\{NB(1,p) \ge j\}$, we have

$$\text{Prob}\{\max(X_1, X_2, \ldots, X_n) - \log_{1/(1-p)} n \ge j\} \le \text{Prob}\{NB(1,p) \ge j\}. \square$$

Lemma A.3. Let p be a probability, L be at least 0, and [$X_0, X_1, \ldots$] be an infinite sequence of independent random variables, each bounded by $NB(1,p)$. Let n be the smallest integer such that $X_n \ge L$. Let ND be the set of non-decreasing elements in [$X_0, X_1, \ldots$]:

$$ND = \{\, X_i \mid \forall j \text{ s.t. } 0 \le j < i, X_j < X_i \,\}.$$

Then

$$| \{ X_i \in ND \mid i \leq n \} | =_{prob} 1 + NB(L, p).$$

Proof. Define $P_{i,k,c}$ informally to be the probability that in the sequence $[X_i, \ldots]$, we will encounter at least k elements of ND up until the point where we have climbed c additional levels. Formally,

$$P_{i,k,c} = \text{Prob}\{ \ | \{ X_j \in ND \mid i \leq j \leq m \} | \geq k \},$$
where m is the smallest integer such that $X_m \geq c + \max(0, X_0, X_1, \ldots, X_{i-1})$.

Using this definition,

$$\text{Prob}\{ \ | \{ X_i \in ND \mid i \leq n \} | \geq k \} = P_{0,k,L}.$$

Setting up $P_{i,k,c}$ as a recurrence relation, we get the result shown below.

$P_{i,k,c} =$
- if $k = 0$ then 1
- else if $c = 0$ then 0
- else if $k = 1$ then 1
- else if $X_i \geq \max(0, X_0, X_1, \ldots, X_{i-1})$
  - then $P_{i+1,k-1,c'}$ where $c' = c - (X_i - \max(0, X_0, X_1, \ldots, X_{i-1}))$
- else $P_{i+1,k,c}$

If $X_i \geq \max(X_0, X_1, \ldots, X_{i-1})$ then $X_i - \max(X_0, X_1, \ldots, X_{i-1}) =_{prob} NB(1, p)$. Solving this recurrence relationship, if $k > 0$ then $P_{0,k,c}$ is equal to the probability that we will see at least $k - 1$ heads before c tails are seen in a series of random, independent coin flips, where the probability of seeing a head is p. If $k = 0$, $P_{0,k,c}$ is equal to 1. Therefore,

$$P_{0,k,c} = \text{Prob}\{ NB(c, p) \geq k - 1 \}.$$

Combining our results,

$$\text{Prob}\{ \text{The number of non-decreasing elements in } [X_0, X_1, \ldots, X_n] \geq k \}$$
$$= \text{Prob}\{ 1 + NB(L, p) \geq k \}. \square$$

Lemma A.4. Let p be a probability, and $X_0, X_1, \ldots, X_{n-1}$ be independent random variables, each bounded by $NB(1, p)$.

$$| \{ X_i \mid X_i = \max(X_0, X_1, \ldots, X_{n-1}) \} | \leq_{prob} 1 + NB(1, 1 - p).$$

Proof. Define $P_{i,k} = \text{Prob}\{ \ | \{ X_j \mid j < i \wedge X_j = \max(X_0, X_1, \ldots, X_{i-1}) \} | \geq k \}$. The definition of $P_{i,k}$ as a recurrence relation is shown below.

$P_{i,k} =$
- if $i \geq n$ then $P_{n-1,k}$
- else if $k = 1$ then 1
- else if $X_i > \max(X_0, X_1, \ldots, X_{i-1})$ then 0
- else if $X_i = \max(X_0, X_1, \ldots, X_{i-1})$ then $P_{i-1,k-1}$
- else $P_{i-1,k}$

If $X_i \geq \max(X_0, X_1, \ldots, X_{i-1})$, $\text{Prob}\{ X_i > \max(X_0, X_1, \ldots, X_{i-1}) \} = 1 - p$. Solving this recurrence relationship, we get the result that the limit, as n goes to infinity, of $P_{n,k}$ is equal to Prob{ $NB(1, 1 -$

$p) \geq k-1$ }. Since $P_{n,k}$ approaches Prob{ $NB(1, 1-p) \geq k-1$ } from below, $P_{n,k}$ is at most Prob{ $NB(1, 1-p) \geq k-1$ }. Therefore,

$$\text{Prob}\{\ |\ \{X_j \mid X_j = \max(X_0, X_1, ..., X_{n-1})\}\ |\ \geq k\}$$
$$\leq \text{Prob}\{\ 1 + NB(1, 1-p) \geq k\}. \square$$

Lemma A.5. Let p_1 and p_2 be probabilities, and $s \geq 0$.

$$NB(NB(s, p_1), p_2) \leq_{prob} NB(s, \frac{p_1 p_2}{1 - p_2 + p_1 p_2})$$

Proof. Let $X_0, X_1, ..., X_{s-1}$ be independent random variables, each $=_{prob} NB(1, p_1)$. Let $Y_0, Y_1, ..., Y_{s-1}$ be random variables, independent of each other, such that $Y_i =_{prob} NB(X_i, p_2)$. $Y_0 + Y_1 + ... + Y_{s-1} =_{prob} NB(NB(s, p_1), p_2)$. We can generate values for $Y_0, Y_1, ..., Y_{s-1}$ using the procedure *r*() below, which makes use of an procedure *random*() that returns a random value uniformly distributed in the range [0..1).

```
r() ≡
    Y := 0
    while random() > p1 do
        while random() > p2 do
            Y := Y+1
    return Y
```

A small amount of program transformation produces:

```
r() ≡
    if random() ≤ p1
        then return 0
        else return r′()

r′() ≡
    Y := 0
    repeat
        while random() > p2 do
            Y := Y+1
    until random() ≤ p1
    return Y
```

Define $P_{i,k}$ to be the probability that the value returned by r' is at least k, assuming the while loop in r' is executed no more than i times. Setting up $P_{i,k}$ as a recurrence relation, we get:

$P_{i,0} = 1$,
$P_{0,k}\ (k > 0) = 0$, and
$P_{i+1,k} = (1 - p_2)P_{i,k-1} + p_2(1 - p_1)P_{i+1,k}$.

Solving this recurrence relation, we get

$$\lim_{i \to \infty} P_{i,k} = (1 - \frac{p_1 p_2}{1 - p_2 + p_1 p_2})\, P_{i,k-1}.$$

This gives the result that value returned by r' is $=_{prob}$ $NB(1, p_1 p_2/(1 - p_2 + p_1 p_2))$. Since the value returned by r is $\leq_{prob}$ the value returned by r', the value returned by r is $\leq_{prob}$ $NB(1, p_1 p_2/(1 - p_2 + p_1 p_2))$. Since each of the Y_i's are independent, $NB(NB(s, p_1), p_2) \leq_{prob} NB(s, p_1 p_2/(1 - p_2 + p_1 p_2))$. □

Theorem A.3. Let p be a probability, and $X_0, X_1, ..., X_{n-1}$ be independent random variables, each bounded by $NB(1, p)$. Let ND be the set of non–decreasing elements in the sequence $[X_0, X_1, ..., X_{n-1}]$, which is equal to $\{ X_i \mid 0 \leq i < n \wedge \forall j \text{ s.t. } 0 \leq j < i, X_j < X_i \}$.

$$| ND | \leq_{prob} 1 + NB(\log_{1/(1-p)} n, p) + NB(1, p^2/(1 - p + p^2)) + NB(1, 1 - p)$$

Proof. Let $L = \max(X_0, X_1, ..., X_{n-1})$. By Theorem A.2, $L \leq_{prob} \log_{1/(1-p)} n + NB(1, p)$. Let m be the smallest integer such that $X_m \geq L$.

By Lemma A.3, the number of non–decreasing elements in $[X_0, ..., X_m]$ $=_{prob}$ $1 + NB(L, p)$. By Lemma A.4, the number elements in $[X_m, ..., X_{n-1}]$ that are equal to L is $=_{prob}$ $1 + NB(1, 1 - p)$. By Lemma A.5, $NB(L, p) \leq_{prob} NB(\log_{1/(1-p)} n, p) + NB(1, p^2/(1 - p + p^2))$.

$$\begin{aligned} | ND | \leq_{prob} & (1 + NB(\log_{1/(1-p)} n, p) + NB(1, p^2/(1 - p + p^2))) \\ & + (1 + NB(1, 1 - p)) \\ & - 1. \square \end{aligned}$$

Bibliography

[ACRSZ87] B. Alpern, A. Carle, B. Rosen, P. Sweeny and K. Zadeck. *Incremental evaluation of attributed graphs*, Technical Report RC 13205, IBM, Thomas J. Watson Research Center, Yorktown Heights, New York 10598, October 1987.

[All78] John Allen. *Anatomy of LISP*, McGraw Hill Book Company, NY, 1978.

[Ben82] Safwan A. Bengelloun. *Aspects of Incremental Computing*, Ph.D. Thesis, Yale University, 1982

[BPR85] A.M. Berman, M.C. Paull and B.G. Ryder. *Proving relative lower bounds for incremental algorithms*, Technical Report DCS–TR–154, Department of Computer Science, Rutgers University, 1985.

[Coh79] Norman Cohen. Characterization and Elimination of Redundancy in Recursive Programming, *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[CR88] Martin Carroll and Barbra Ryder. *Incremental Data Flow Analysis via Dominator and Attribute Update*, PROC of Fifteenth POPL, pp 260–273, 1988.

[DRT81] Alan Demers, Thomas Reps and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax–directed editors. *PROC of the Eighth POPL*, pp 105–116, 1981.

[FWW76] Friedman, D. Wise and M. Wand. Recursive Programming through Table Look–up, *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pp 85-89.

[GK76] Eiichi Goto and Yasumasa Kanada. Hashing Lemmas on Time Complexities with Applications to Formula Manipulation, *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pp 154-158.

[Hen86] Peter Henderson. Data–Oriented Incremental Programming Environments, *Advanced Programming Environments*, Lecture Notes in Computer Science 244 (13-25), Springer–Verlag (Jun 1986).

[Hil76] J. Hilden. Elimination of recursive calls using a small table of 'randomly' selected function values, *BIT* **16**(1):60–73 (1976).

[Hoo86] Roger Hoover. Dynamically bypassing copy rules in attribute grammar, *PROC of Thirteenth POPL*, pp 14–25, 1986.

[Hoo87] Roger Hoover, *Incremental Graph Evaluation*, Ph.D. Thesis, Cornell University, 1987.

[HT86a] Roger Hoover and Tim Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars, *PROC of SIGPLAN 86 Symposium on Compiler Construction*, pp 39–50, 1986.

[HT86b] Susan Horowitz and Tim Teitelbaum. Generating editing environments based on relations and attributes, *TOPLAS*, pp 577–608, 1986.

[Hug85] John Hughes. Lazy Memo-functions, *Functional Programming Languages and Computer Architecture*, 1985

[HW85] Peter Henderson and Mark Weiser. Continuous Execution: The VisiProg Environment, *Eighth International Conference on Software Engineering*, 1985.

[JF82] Greg Johnson and Charles Fischer. Non–syntactic Attribute Flow in Language Based Editors, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1982, pp 185–195.

[JF85] Greg Johnson and Charles Fischer. A Meta–Language and System for Nonlocal Incremental Attribute Grammar Evaluation in Language–Based Editors, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp 141–151 (1985).

[JW88] Greg Johnson and Janet Waltz. *Incremental evaluation of a general class of circular attribute grammars*, SIGPLAN 88

[Kat84] Takuya Katayama. *Translation of attribute grammars into procedures*, ACM Trans. of Programming Languages and Systems, 6(3):345–369, July 1984

[Knu73] Donald Knuth. Sorting and Searching, *The Art of Computer Programming*, Vol. 3, 1973.

[KS86] Robert M. Keller and M. Ronan Sleep. Applicative caching, *ACM Transactions on Programming Languages and Systems* 8(1):88–106 (Jan 1986).

[MC85] Jack Mostow and Donald Cohen. Automating Program Speedup by Deciding What to Cache, *Proc. of the Ninth International Joint Conference on Artificial Intelligence* 1:165–172 (Aug 1985).

[McC65] J. McCarthy et al., *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, 1965.

[MF81] Raul Medina–Mora and Peter H. Feiler. An Incremental Programming Environment, *IEEE Transactions on Software Engineering* 7(5) (Sep 1981).

[Mic68] Donald Michie. "Memo" Functions and Machine Learning, *Nature* 218:19–22 (Apr 1968).

[Nis88] Naomi Nishimura. *Complexity Issues in Tree–Based Version Control*, Technical Report 212/88, University of Toronto, June, 1988.

[Par78] Luis Isidoro Trabb Pardo. *Set Representation and Set Intersection*, Ph.D. thesis, Stanford University, 1978

[RA84] Thomas Reps and Bowen Alpern. Interactive proof checking, *PROC of Eleventh POPL*, pp 36–45, 1984.

[Rep82] Thomas Reps. Optimal–time incremental semantic analysis for syntax–directed editors, *PROC of Ninth POPL*, pp 169–176, 1982.

[Rep84] Thomas Reps. *Generating Language-Based Editors*, The MIT Press, Cambridge, 1984.

[Rob87] Arch Robison. The Illinois Functional Programming Interpreter, *Proceedings of the ' AN 87 Symposium on Interpreters and Interpretive Techniques.*

[Ros81] Barry Rosen. Linear Cost is Sometimes Quadratic, *PROC of the Eighth POPL*, pp 117–124, 1981.

[RT84] Tom Reps and Tim Teitelbaum. The synthesizer generator, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp 42-48 (Apr 84).

[RT87] Thomas Reps and Tim Teitelbaum. Language processing in program editors, *IEEE Computer*, **20**(11):29–40 (Nov 1987).

[SG76] M. Sassa and E. Goto. A hashing method for fast set operations. *Inf. Proc. Let.* 5(2):31–34, June 1976.

[SL78] J.M. Spitzen and K.N. Levitt. An example of hierarchical design and proof, *Communications of the ACM*, **21**(12):1064–1075, December 1978.

[TK84] H. Taylor and S. Karlin. *An Introduction to Stochastic Modeling*, Academic Press, Orlando, Florida, 1984.

[Van88] Bradley Vander Zanden. *Incremental Constraint Satisfaction and it's Application to Graphical Interfaces*, Ph.D. thesis, Cornell University, 1988.

[YS87] Daniel Yellin and Robert Strom. *INC: A language for incremental computation*, Technical Report RC 13327, IBM, Thomas J. Watson Research Center, Yorktown Heights, New York 10598, December 1987.