INCREMENTAL COMPUTATION: A SEMANTICS-BASED SYSTEMATIC TRANSFORMATIONAL APPROACH

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yanhong Annie Liu

January 1996

INCREMENTAL COMPUTATION: A SEMANTICS-BASED SYSTEMATIC TRANSFORMATIONAL APPROACH

Yanhong Annie Liu, Ph.D.
Cornell University 1996

Incremental computation takes advantage of repeated computations on inputs that differ slightly from one another, computing each new output incrementally by making use of the previous output rather than from scratch.

This thesis concerns the theory, design, and implementation of a general approach to incremental computation. It also elucidates the essence of improving the efficiency of computations by relating it to incremental computation. Our general approach allows incremental computation to be obtained systematically from non-incremental computation and program efficiency to be systematically improved.

This research focuses on identifying the fundamentals of efficient incremental computation out of domain-specific properties and language-specific features, devising a general framework that accommodates these fundamentals, and developing a systematic approach based on the framework that exploits program semantics.

Three fundamental aspects of incremental computation are identified: avoiding repeated identical computations, caching useful intermediate results, and discovering appropriate auxiliary information. Given a program f and an operation $\oplus$, an incremental program is developed to compute $f(x \oplus y)$ efficiently by using $f(x)$, the intermediate results computed in computing $f(x)$, and auxiliary information about x that can be inexpensively maintained.

The approach in this thesis is formalized for a simple functional language, but the underlying principles also apply to other programming languages. It exploits program semantics to discover incrementality that is not directly embedded in primitive operators and takes into consideration properties of application domains as well. It is composed of step-wise program analysis and transformation modules that can, for the most part, be mechanized.

Since every non-trivial computation proceeds by iteration (or recursion), the approach is used straightforwardly for achieving efficient computation in general, by computing each iteration incrementally using an appropriate incremental program. This method is applied to problems in interactive systems, optimizing compilers, transformational program development, *etc.* The design and implementation of a prototype system, CACHET, for obtaining incremental programs is also described.

Biographical Sketch

Yanhong Liu was born on May 20, 1965, in Beijing, China. After living a few years in Beijing, she went to Hefei with her parents and spent several colorful years of childhood on the campus of the University of Science and Technology of China. On the day before the disastrous Tangshan earthquake, she was sent to her grandparents back in Beijing and, two years later, she had the luck of attending Huiwen High School, which her father had attended before and her sister attended later. At the wish of her parents in 1983, she went to Peking University to study Computer Science but luckily met so many wonderful people and learned so many wonderful things that she has loved Peking University and Computer Science since then. She received a Bachelor of Science degree in 1987. To further her studies, she went to Tshinghua University and received a Master of Engineering degree in 1988. In 1990, she decided to attend graduate school at Cornell University in America. There, she gave herself the name Annie and had the luck of meeting beautiful people and learning beautiful ideas in Computer Science. She received a Master of Science degree in 1992. She has accepted a position as a Post-Doctoral Associate with Cornell's Department of Computer Science.

To all my loving teachers,
especially my parents.

Acknowledgements

First of all, I would like to thank my thesis advisor, Tim Teitelbaum, for his generous guidance, patience, and knowledge, which have been invaluable to the research in this thesis and beyond. I feel very lucky that I have been able to work closely with him for the past four years.

I also want to thank Bob Constable and Anil Nerode for serving on my thesis committee and showing great interest in my research. They have been particularly helpful in bringing related work to my attention and broadening my view. David Gries deserves special thanks for discussions and for help in polishing this thesis.

I also feel deeply indebted to Dexter Kozen, Bard Bloom, Keith Marzullo, and Anil Nerode for great encouragement during my course studies at Cornell. Dexter has also helped me sort out some strange questions that arose during my research.

I would also like to thank Scott Stoller for his love and encouragement, his understanding, consideration, and patience. He has usually been the first person to hear what I have been working on; he has given me immense help in making my ideas more precise and my writing more succinct.

I benefited a great deal from stimulating discussions with Tom Reps, Bob Paige, and Doug Smith through conferences, visits, and emails. They helped me understand important issues related to this research. In particular, Tom's comments, received directly or indirectly through Tim, have always helped push this work forward.

Thanks also go to John Reppy, John Field, Roger Hoover, Bill Pugh, Tom Marlowe, Allen Brown, Daniel Weise, Keshav Pingali, Richard Johnson, Devika Subramanian, Adam Webber, Tom Henzinger, Aswin van den Berg, Thomas Yan, Wilfred Chen, and Monika Rauch Henzinger for discussions about related research.

The Office of Naval Research has provided financial support under grants N00014-92-J-1973.

Many friends at Cornell and old friends from Peking University and Tsinghua University have helped make my years in Ithaca enjoyable. I would not be myself without their friendship.

Finally, my parents have given me everything with love and prepared me for both high points and low points in my endeavors. I thank you, Baba and Mama.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Software development and maintenance

Computations in all kinds of computing systems are pervasive activities in today's information age. Even though hardware is getting faster and faster, complicated applications that require intensive computations are increasingly desired, which requires software to be more and more efficient. This straightforwardly explains the pervasive symptom in software: even the simplest tasks are obscured by optimizations in designs and programs.

Optimizations are hard to find and code, and they make it even harder to understand and maintain the resulting software. These lead to the high cost of software development and the even higher cost of software maintenance. Methodologies for program improvement, such as step-wise refinement, correctness-preserving transformations, and object-oriented methods, provide guidelines for the development and maintenance process. Programming tools, such as structured editors, compilers for very-high-level languages, and application generators, furnish programmers with a modest amount of automated assistance.

Despite these developments, most people still find the software development and maintenance process frustrating. Existing approaches and techniques are not generally applicable for improving program efficiency. Software development and maintenance costs keep rising. This bogs down the development and maintenance of programming tools as well: often, these tools are themselves not fast enough.

A conceptual breakthrough that solves all the problems involved in software development at once is unlikely; however, there has always been a need to explore general principles illuminating the essence of efficient computation and to develop a systematic approach based on these principles that can be used to guide efficiency improvement for all applications.

1.2 Incremental computation—the essence of improving the efficiency of computations

This thesis concerns the theory, design, and implementation of a general approach to improving the efficiency of computations. Given a program f and an input change operation $\oplus$, the approach aims to obtain an incremental program that computes $f(x \oplus y)$ efficiently by making use of $f(x)$, the intermediate results computed in computing $f(x)$, and auxiliary information about x that can be inexpensively maintained. Since every non-trivial computation proceeds by iteration (or recursion), the approach is used straightforwardly for achieving efficient computation in general by computing each iteration incrementally using an appropriate incremental program.

This approach states explicitly that *incremental computation* is the essence of improving the efficiency of computations. The principles of the approach are essentially the same as those underlying the work by Allen, Cocke, Kennedy, and others on strength reduction [All69,CS70,Gri71,CK77,ACK81], by Earley on high-level iterators [Ear76], by Fong and Ullman on inductive variables [FU76,Fon77,Fon79], by Paige, Schwartz, and Koenig on finite differencing [PS77,Pai81,PK82], by Dijkstra, Gries, and Reynolds [Dij76,Gri81,Rey81,Gri84] on maintaining and strengthening loop invariants, by Boyle, Moore, Manna, and Waldinger on induction, generalization, and deductive synthesis [BM79,MW80,MW93], by Dershowitz on extension techniques [Der83], by Bird on promotion and accumulation [Bir84,Bir85], by Broy, Bauer, Partsch, *etc.* on transforming recursive functional programs in CIP [Bro84,BMPP89, Par90], by Smith on finite differencing of functional programs in KIDS [Smi90,Smi91], as well as the work pioneered by Michie on memoization [Mic68,Bir80,Coh83,Web95]. The most basic idea can be traced back to the Difference Engine of Charles Babbage in the 19th century [Gol72].

The approach in this thesis is formalized for a simple functional language with all the most basic program constructs: variables, data constructors, primitive functions, user-defined recursive functions, conditionals, and binding expressions. The underlying principles also apply to other programming languages. The approach exploits program semantics, using analysis of program states at various program points, to discover incrementality that is not directly embedded in primitive operators. It is composed of step-wise program analysis and transformation modules that can, for the most part, be mechanized. Therefore, compared to work by Allen, Cocke, Kennedy, Earley, Fong, Paige, *etc.*, where a set of rules is developed to transform expensive primitive operations syntactically into efficient incremental operations, the approach in this thesis is more semantics-based and more general; compared to work by Dijkstra, Gries, Boyle, Manna, Dershowitz, Bird, Broy, Smith, *etc.*, where only general strategies are suggested, the approach in this thesis is more systematic and more automatable.

1.3 An overview of the work

This work began with the study of *incremental computation*, efficient computation that takes advantage of repeated computations on inputs that differ slightly from one another, making use of the old output in computing a new output rather than computing from scratch. The large number of works on incremental computation in recent years and their many applications [RR93], demonstrated by various incremental algorithms such as [GM79,JG82,RTD83,Yeh83,RP88,Van88,FMB90,AHR+90,RR94] and general incremental computation approaches such as [Ear76,Pai81,PK82,Pai84,HT86, CP89,PT89,FT90,Smi90,Smi91,YS91,SH91,Sun91,Hoo92,van92,Fie93], motivated us to look for the fundamentals of incremental computation and their role in efficient computation. The goal has been a general approach and tools that allow incremental computation to be obtained systematically from non-incremental computation and program efficiency to be systematically improved.

The most challenging aspect of such a project is the design of a general framework that captures the fundamentals of incremental computation. Program efficiency depends on properties of domains for which programs are designed and on features of languages in which programs are written. Therefore, this research has focused on identifying the fundamentals of efficient incremental computation out of domain-specific properties and language-specific features, devising a general framework and developing a systematic approach based on this framework that exploits program semantics.

A second aspect of this work has been relating the fundamentals of incremental computation with efficient computation and program improvement in general. For applications that manipulate direct input changes, such as interactive systems, efficient incremental computation is directly adopted. For general program efficiency improvement, the iterative (or recursive) computation that is essential to all non-trivial programs is cast as incremental computation in each iteration. This establishes incremental computation as the essence of improving the efficiency of computations.

A third aspect of this work has been investigating the applicability of this general approach for efficient computation. This resulted in re-development or re-discovery of many existing incremental techniques and incremental programs. In some cases, efficient programs were obtained by systematically following the approach where no previous systematic approach was known. This investigation was further assisted by the implementation of our prototype system, CACHET.

As discussed further in Chapter 2, much effort has been devoted to incremental computation, but existing techniques depend to a large extent on domain-specific properties, language-specific features, or both. Each technique has its own functionality, none subsumes the others, relationships among them are scattered, and applicability to various application domains are not always clear.

In contrast, the approach in this thesis is general and systematic. It sorts out fundamental aspects of incremental computation approached by existing techniques and beyond. These aspects include avoiding repeated identical computations, identifying useful intermediate results, determining appropriate auxiliary information, *etc.* The general principles underlying the approach apply to different programming languages and data structure organizations and take into consideration properties of application domains as well. The approach is straightforwardly applied for systematic program efficiency improvement. The prototype implementation helped demonstrate that effective tools can be developed based on the systematic approach.

1.4 An outline of the dissertation

This dissertation explores fundamental aspects of incremental computation step by step, together with their applications to improving program efficiency in general.

As mentioned above, Chapter 2 discusses the literature in incremental computation. The desire to capture the fundamental issues out of the literature has inspired us with a new outlook on various techniques. The new outlook has led to a semantics-based, systematic, transformational approach for incremental computation and program improvement, as discussed in Chapters 3, 4, and 5.

The first and most important task was to devise a general approach for using the result of a previous computation for an efficient new computation on changed input. Such an approach to deriving *incremental programs*, also called *incrementalization*, is described in Chapter 3. The basic idea is to symbolically expand the computation on the new input and replace subcomputations whose values can be efficiently retrieved from the cached result of the previous computation by the corresponding retrievals. This approach exploits program semantics in a way that is more general and systematic than previous approaches.

The efficiency of a new computation may often be improved by using not only the result of a previous computation but also some *intermediate results*. These results need to be identified and maintained. Chapter 4 presents a systematic approach for symbolically caching intermediate results to improve efficiency of the derived incremental programs. It first symbolically caches *all* intermediate results, then incrementalizes the resulting program, and finally prunes the programs to retain only intermediate results that are useful for the incremental computation. The approach is applied for improving program efficiency using caching, also called *memoization*.

Furthermore, the efficiency of a new computation may be improved by using information other than the intermediate results computed by the original computation; such *auxiliary information* must be discovered, initialized, and efficiently maintained. Chapter 5 proposes transforming the computation on the new input to expose subcomputations that depend only on the old input but are not in the old computation as candidate auxiliary information. A comprehensive approach is given that addresses using auxiliary information and intermediate results at the same time.

A prototype system, CACHET, has been developed to facilitate application of the systematic approach for deriving incremental programs written in our simple functional language. It allows interactive program transformation by direct tree manipulation and, as program trees are transformed, performs incremental program analysis by incremental attribute evaluation. The design and implementation of CACHET are described in Chapter 6.

Chapter 7 summarizes of the major ideas discussed in this thesis, draws some conclusions, and suggests areas for future research.

As an aid to the reader, an index of symbols, terms, and names are included at the end of the thesis.

Much of Chapters 3, 4, 5, and 6 appears in separate papers (specifically, [LT95b], [LT95a], [LST96], and [Liu95], respectively). Each of them is largely self-contained.

Chapter 2

Providing a general systematic approach to efficient computation

Incremental computation takes advantage of repeated computations on inputs that differ slightly from one another, computing each new output incrementally by making use of the previous output rather than from scratch. Methods of incremental computation have widespread applications, e.g., loop optimizations in optimizing compilers [Ear76,CK77,MJ81,ASU86] and transformational programming [Pai83,Par90, Smi90], interactive systems like programming environments [MF81,Rep84] and editors [RTD83,RT88,BGV92], and dynamic systems like distributed databases [LS92, CHKS93,ZGMHW94] and real-time systems [VC92].

The premise of this work is that methods of incremental computation can be generalized and systematized by finding a conceptual model that captures all the fundamentals. A comprehensive guide to methods of incremental computation has appeared in [RR93]. Despite the relatively diverse categories discussed in [RR93], we divide most of the work into three classes: incremental algorithms, incremental execution frameworks, and incremental-program derivation approaches. This classification provided a new prospective on the literature in incremental computation, which then motivated the approach in this thesis.

2.1 Incremental algorithm

Our first class includes particular incremental algorithms designed for particular problems dealing with particular input changes. Examples include incremental parsing [GM79,JG82], incremental attribute evaluation [RTD83,Yeh83,YK88,LMOW88, Jon90], incremental data-flow analysis [Zad84,RP88,Bur90,MR90,RR94], incremental circuit evaluation [AHR$^+$90], and incremental constraint solving [Van88,FMB90]. The study of dynamic graph algorithms, such as transitive closure algorithms [Yel93], can be viewed as falling into this class. These incremental algorithms are called *explicit incremental algorithms* by Pugh [Pug88b] and *ad hoc incremental algorithms* by Field [Fie91].

Although efforts in this class are directed towards particular incremental algorithms, they apply to a broad class of problems, e.g., any attribute grammar and any circuit. However, designing these particular algorithms has been the realm of the few experts in the application domain that also have good knowledge about algorithms. As the word *ad hoc* suggests, these algorithms themselves do not provide a general systematic approach for obtaining them.

Complexity-theoretic issues of incremental algorithms have received much attention in the past few years [BPR90,Mil91,RR91,Ber92,SVT93,Ram93,MSVT94]. Although these works do not provide methods for obtaining incremental algorithms, they can assist the study of general approaches to incremental computation by establishing certain theoretical bounds. Research in this area is yet in an early stage.

2.2 Incremental execution framework

In our second class, rather than manually developing particular incremental algorithms as in the first class, application programs are run in a general incremental execution framework so that incremental computation is achieved automatically. Attempts to provide general incremental mechanisms have become more active in the past several years and they mostly fall into this class, e.g., incremental attribute evaluation frameworks [RT88], incremental computation via function caching [PT89], incremental lambda reduction [FT90], formal program manipulation using traditional partial evaluation [SH91,Sun91], the change detailing network of INC [YS91], incremental computation as program abstraction [Hoo92], and incremental term rewriting [van92,Fie93]. Such frameworks are called *incremental evaluators* by Pugh [Pug88b] and *general approaches* by Field [Fie91].

Note that an incremental execution framework always employs a particular incremental algorithm, where the algorithm is designed for executing a particular class of application programs on their inputs and deals with a particular class of changes in an application program and/or its input.

In these frameworks, often no explicit incremental version of an application program is derived and run autonomously by a standard evaluator. Even if we could specialize an incremental execution framework with respect to an application program to get a stand-alone incremental application program, any input change to the application program is mapped to whatever the framework can handle, which is fixed for each framework. Therefore, these solutions to the incremental computation problem for particular applications are not readily comparable with explicitly derived incremental algorithms such as those in the first class.

2.3 Incremental-program derivation approach

In our third class, systematic approaches are studied to derive explicit incremental programs from non-incremental programs using program transformation techniques like finite differencing [Pai81,PK82]. These approaches aim to be general, as do those in the second class; they also aim to derive explicit incremental programs, like those manually derived in the first class. Examples are high-level iterators [Ear76], finite differencing of set expressions in SETL [PK82], optimizing relational database operations [Pai84,HT86], fixed-point computation and recomputation [CP89], differentiation of functional programs in KIDS [Smi90,Smi91], *etc.*

The pioneering work on finite differencing by Paige [PS77,Pai81,PK82,CP89] has been one of the most successful contributions in this class. It uses set-theoretic languages, such as SETL, which provide convenient mathematical notations. Fixed rules are offered for transforming aggregate primitive operations on sets into more efficient incremental operations, and these transformations can be applied automatically. A number of new algorithms have been derived following this approach [PTB85,PT87, CPT92,BP92,CP92].

Unfortunately, such very-high-level languages pose hard questions concerning efficient implementation [PH87,Pai89]. But the fundamental drawback of this approach is that it does not sufficiently exploit program semantics. Thus, no incremental programs can be derived when the given rules do not apply. Such approaches are characterized as *syntactic*, using only *formal notations*, whereas a *semantic* solution that explores the *mathematical content* of the subject has not been developed [Ner92].

Another highly successful contribution for deriving efficient programs is by Smith [Smi90, Smi91]. It advocates a more flexible approach to transforming programs written in a functional language. In this work, a high-level strategy is given and a general simplifier is used for finite differencing of functional programs that takes advantage mainly of distributivity laws. This approach has been implemented in a semi-automatic system KIDS and has helped to derive a number of efficient programs [Smi91,SP93].

Approaches based on such high-level strategies do not provide systematic steps to follow. How to automate such an approach is one of the most challenging issues to be studied.

In summary, in most works in the third class, programs are written in very-high-level languages with aggregate data structures, such as sets and bags; in most other works, only high-level strategies are proposed. What is not provided is a systematic procedure for deriving incremental programs from non-incremental programs written in a standard language.

2.4 A general systematic transformational approach

Providing a systematic approach to deriving incremental programs for a standard language is not simply a matter of treating yet a different language from SETL. In such standard languages, there are primitive operations that can not be mapped syntactically into incremental operations. A general systematic approach to incremental computation must exploit more program semantics to incrementalize programs written using such primitive operations. Such a semantics-based approach can be used to derive and justify finite differencing rules, as well as to obtain efficient incremental computations where no given rules apply.

This thesis discusses such a general systematic approach for discovering incrementality for programs written in a standard functional programming language. Given a program f and an input change operation $\oplus$, it aims to obtain an incremental program that computes $f(x \oplus y)$ efficiently by making use of the value of $f(x)$, the intermediate results computed in computing $f(x)$, and auxiliary information about x that can be inexpensively maintained, as will be discussed in Chapters 3, 4, and 5. The approach exploits a number of program analysis and transformation techniques and domain-specific knowledge, centered around effective caching and its utilization, in order to provide a degree of incrementality not otherwise achievable by a generic incremental evaluator. A prototype implementation of the general approach is described in Chapter 6.

Language. For simplicity of exposition, we use a simple first-order functional programming language. The expressions of our language are given by the following grammar:

$$
\begin{array}{rcll}
e & ::= & v & \text{variable} \\
 & | & c(e_1, ..., e_n) & \text{constructor application} \\
 & | & p(e_1, ..., e_n) & \text{primitive function application} \\
 & | & f(e_1, ..., e_n) & \text{function application} \\
 & | & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & \text{conditional expression} \\
 & | & \textbf{let } v = e_1 \textbf{ in } e_2 & \text{binding expression}
\end{array}
$$

A program is a finite set F of mutually recursive function definitions of the form

$$f(v_1, ..., v_n) = e \tag{2.1}$$

and a function f_0 that is to be evaluated with some input $x = \langle x_1, ..., x_n \rangle$. Note, for simplicity, we will just use f_0 to refer to the program F, and f to refer to an arbitrary function defined in F. Figure 2.1 gives some example definitions.

$$
\begin{array}{lcl}
out(C, R) & : & \text{compute the outer product of lists } C \text{ and } R \\
out(C, R) & = & \textbf{if } null(C) \textbf{ then } nil \\
 & & \textbf{else } cons(row(car(C), R),\ out(cdr(C), R)) \\
row(c, R) & = & \textbf{if } null(R) \textbf{ then } nil \\
 & & \textbf{else } cons(c * car(R),\ row(c, cdr(R))) \\
ins(i, a, R) & : & \text{insert } a \text{ in list } R \text{ at position } i \\
ins(i, a, R) & = & \textbf{if } i \leq 1 \textbf{ then } cons(a, R) \\
 & & \textbf{else if } null(R) \textbf{ then } cons(a, nil) \\
 & & \textbf{else } cons(car(R),\ ins(i-1, a, cdr(R)))
\end{array}
$$

Figure 2.1: Example function definitions of *out*, *row*, and *ins*

Each constructor c, primitive function p, and user-defined function f has a fixed arity. In general, $\bar{c_i}^1$ denotes the i-th selector corresponding to the constructor c. The semantics of the language is strict.

An input change operation $\oplus$ to a program f_0 combines an old input $x = \langle x_1, ..., x_n \rangle$ and a change $y = \langle y_1, ..., y_m \rangle$ to form a new input $x' = \langle x'_1, ..., x'_n \rangle = x \oplus y$, where each x'_i is some function of x_j's and y_k's. For example, an input change operation $\oplus$ to the function *out* of Figure 2.1 can be defined by

$$\langle C', R' \rangle = \langle C, R \rangle \oplus \langle i, a \rangle = \langle C, ins(i, a, R) \rangle. \tag{2.2}$$

For typographical convenience, we shall always use x to denote the previous input to f_0, y to denote the change parameter to the input x, and x' to denote the new input $x \oplus y$. Parameter y can be regarded as a change δx to the input x.

Chapter 3

Deriving incremental programs

Given a program f_0 and an input change operation $\oplus$, a program f_0' that computes $f_0(x \oplus y)$ efficiently by making use of the value of $f_0(x)$ is called an *incremental version* of f_0 under $\oplus$. For example, if *sort* is a sorting program and *add* adds a new element to an old input list, then a program that inserts the new element into the old sorted output list at the appropriate place is an incremental version of *sort* under *add*.

This chapter discusses a general systematic approach for deriving an incremental version f_0' from a program f_0 and an input change operation $\oplus$ written in a standard functional programming language. The basic idea is to identify in the computation of $f_0(x \oplus y)$ subcomputations that are also performed in the computation of $f_0(x)$ and whose values can be retrieved from the cached result r of $f_0(x)$. The computation of $f_0(x \oplus y)$ is symbolically transformed to avoid re-performing these subcomputations by replacing them with corresponding retrievals. This efficient way of computing $f_0(x \oplus y)$ is captured in the definition of $f_0'(x, y, r)$.

Defining the problem. Discussing efficient computation needs a cost model, accordingly, a time model $\mathcal{T}$ such that $\mathcal{T}(e)$ describes the time needed to compute expression e. Function $\mathcal{T}$ can be obtained from standard constructions [Weg75,Ros89]. In general, given two expressions e_1 and e_2, it is not decidable whether e_2 computes faster than e_1 for given values of their variables. Therefore, we say $\mathcal{T}(e_2) \leq \mathcal{T}(e_1)$ if we can *effectively* confirm the inequality. Suppose $v_1, ..., v_k$ are all the variables in e_1 and e_2 and P is some predicate on these variables. We write

$$t(e_2) \leq_P t(e_1) \tag{3.1}$$

to denote that we can effectively decide that there is a constant k such that, for any values of $v_1, ..., v_k$, if P holds then $\mathcal{T}(e_2) \leq k\mathcal{T}(e_1)$, and we say that e_2 is *asymptotically at least as fast* as e_1 under P. During our derivation, P always represents the equations that hold at the occurrence of the expression currently under consideration; therefore, it will be omitted for simplicity.

Given a program f_0 and an input change operation $\oplus$, we aim to derive an incremental version f_0' of f_0 under $\oplus$, such that, if $f_0(x) = r$, then whenever $f_0(x \oplus y)$ returns a value, $f_0'(x, y, r)$ returns the same value and is asymptotically at least as fast.[1] Instead of trivially defining $f_0'(x,y,r)$ to be $f_0(x \oplus y)$, we attempt to make f_0' as efficient as possible by having it use the cached result r of $f_0(x)$ as much as possible.

For typographical convenience, we shall always use r to denote the cached result of $f_0(x)$, and f_0' to denote an incremental version of f_0 under $\oplus$. When r is used as a parameter to f_0', we call it a *cache parameter*.

We use function *out* of Figure 2.1 and input change operation $\oplus$ in (2.2) as a running example. At the end, we will obtain the incremental version out' shown in Figure 3.1.

If $out(C,R) = r$, then
$out'(C,i,a,r) = out(C, ins(i,a,R))$.

For C of length m and R of length n,
$out'(C,i,a,r)$ takes time $O(m * \min(i,n))$;
$out(C, ins(i,a,R))$ takes time $O(m*n)$.

$$
\begin{aligned}
out'(C,i,a,r) &= \textbf{if } null(r) \textbf{ then } nil \\
& \textbf{ else } cons(row'(car(C),i,a,car(r)), \\
&\phantom{= \textbf{ else } cons(} out'(cdr(C),i,a,cdr(r))) \\
row'(c,i,a,r_1) &= \textbf{if } i \le 1 \textbf{ then } cons(c*a,\ r_1) \\
& \textbf{ else if } null(r_1) \textbf{ then } cons(c*a,\ nil) \\
& \textbf{ else } cons(car(r_1), \\
&\phantom{= \textbf{ else } cons(} row'(c,i-1,a,cdr(r_1)))
\end{aligned}
$$

Figure 3.1: Resulting function definitions of out' and row'

3.1 Outlining the derivation procedure

The derivation procedure recursively follows function applications in the computation of $f_0(x \oplus y)$ and aims to replace these applications by uses of new functions introduced to compute the applications incrementally.

To introduce a new function f' to compute a function application $f(e_1,...,e_n)$ incrementally, we collect an information set I_f that describes the context of the application and a cache set C_f that indicates how the values of certain relevant computations can be retrieved from a cached result under certain conditions. Then, we obtain a definition of f' by the following three steps. First, we unfold [BD77] (also called expand [Weg76]) the application. Second, we incrementalize the unfolded application. Basically, we consider each subexpression e of the unfolded application in applicative order and (a) collect an information set $I_{[e]}$ from e's context based on I_f and extend the cache set C_f under the condition that the facts in $I_{[e]}$ are valid, (b) recursively apply this procedure if e is a function application, (c) apply simplification using $I_{[e]}$ and replacement by efficient retrieval using the extended C_f. Third, we eliminate dead code mainly related to dead parameters of f'. If the function f' so obtained is suitably fast, then $f(e_1,...,e_n)$ can be replaced by an application of f'. Other applications of f that are subsequently analyzed may also be replaced by applications of this f', if appropriate.

[1]While $f_0(x)$ abbreviates $f_0(x_1,...,x_n)$, and $f_0(x \oplus y)$ abbreviates $f_0(\langle x_1,...,x_n\rangle \oplus \langle y_1,...,y_m\rangle)$, $f_0'(x,y,r)$ abbreviates $f_0'(x_1,...,x_n,y_1,...,y_m,r)$. Note that some of the parameters of f_0' may be dead and eliminated, as discussed in Section 3.3.

The derivation procedure starts by considering the function application $f_0(x \oplus y)$, with an empty information set and a cache set containing only $f_0(x) = r$. We maintain a global data structure for the set D of functions introduced during the derivation procedure. We take special care of recursive function applications to help the derivation procedure terminate naturally and, at the same time, discover as much incrementality as possible. When finished, we have the original set of functions F and the set D of functions introduced during the derivation procedure, including f_0'. We then eliminate dead functions in F and D not needed in computing f_0'.

A function $\mathcal{I}nc\mathcal{A}pply$ implements the recursive procedure on a function application $f(e_1, ..., e_n)$ with information set I, cache set C, and global definition set D:

$$\mathcal{I}nc\mathcal{A}pply[\![f(e_1, ..., e_n)]\!]\; I\; C\; D = \langle f'(e_1', ..., e_m'),\; D' \rangle \tag{3.2}$$

f' is an introduced function such that $f'(e_1', ..., e_m')$ computes $f(e_1, ..., e_n)$ incrementally under I and C. Global set D may be extended to D' as a side effect.

Two Main Issues. The derivation procedure has two main tasks. First, incrementalizing an unfolded function application, i.e., discovering and replacing subcomputations whose values can be efficiently retrieved from cached results. Second, analyzing recursive function applications and introducing incremental versions that are used to replace these applications.

The first task corresponds to maintaining cache sets under collected information sets at subexpressions of an unfolded application and applying simplification and replacement to these subexpressions using these sets. The second task corresponds to maintaining a global set of functions introduced to compute function applications incrementally and replacing function applications with appropriate applications of the introduced functions.

The two main issues are addressed in Sections 3.2 and 3.3, respectively. Section 3.4 summarizes the derivation procedure and addresses a number of important issues, including correctness, termination, and mechanization. Section 3.5 gives some examples. Section 3.6 discusses related work.

3.2 Incrementalization

We define two notions, *information set* and *cache set*. Given an information set I and an initial cache set C relevant to a function application, we describe how to use them in incrementalizing the unfolded application, i.e., collecting information sets at subexpressions, extending the cache set with respect to the collected information sets, and using them to simplify subexpressions and replace subexpressions whose values can be efficiently retrieved from cached results.

3.2.1 Information set and simplification

An information set $I_{[e]}$ at the occurrence of an expression e is a collection of equations that hold in the context of e. It represents symbolically the program state right before the execution of expression e. We write $e_1 \leftrightarrow e_2$ to denote that two expressions e_1 and e_2 are equal. For example, if $f_0(x \oplus y)$ is unfolded to an expression e as follows:

$$\textbf{let } v = e_1 \textbf{ in if } v = 0 \textbf{ then } 0 \textbf{ else } e_2$$

then $I_{[e]} = \emptyset$, $I_{[e_1]} = \emptyset$, $I_{[v=0]} = \{v \leftrightarrow e_1\}$, and $I_{[e_2]} = \{v \leftrightarrow e_1, v=0 \leftrightarrow F\}$. Note that we use T to denote the Boolean value *true*, and F to denote the Boolean value *false*.

Given an information set at a top-level expression, it is simple to compute information sets at occurrences of subexpressions:

$$\begin{array}{ll} \text{if } e \text{ is } c(e_1,...,e_n),\ p(e_1,...,e_n), \text{ or } f(e_1,...,e_n), & \\ & \text{then } I_{[e_i]} = I_{[e]} \text{ for } i = 1..n; \\ \text{if } e \text{ is } \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, & \text{then } I_{[e_1]} = I_{[e]},\ I_{[e_2]} = I_{[e]} \cup \{e_1 \leftrightarrow T\}, \\ & \qquad\qquad\quad I_{[e_3]} = I_{[e]} \cup \{e_1 \leftrightarrow F\}; \\ \text{if } e \text{ is } \textbf{let } v = e_1 \textbf{ in } e_2, & \text{then } I_{[e_1]} = I_{[e]},\ I_{[e_2]} = I_{[e]} \cup \{v \leftrightarrow e_1\}. \end{array}$$

An underlying logic L_0 is used to make inferences based on the facts in an information set. We require that L_0 be *compatible* with the semantics of the programming language we are using [FN88], i.e., if two expressions are proved to be equal under L_0, then they compute the same value. In this work, we assume that a theorem prover based on L_0 is available, and we write $e_1 \leftrightarrow_I^* e_2$ to denote that a finite proof that e_1 equals e_2 can be found by the theorem prover using equations in set I.

Simplification. We can simplify expressions using information sets and the underlying logic, as summarized in Figure 3.2. Given an expression e and an information set I, we say that *e can be simplified under I to e'* if the corresponding condition $cond(I)$ holds.

Basically, the simplification is as conventional, except with the identity relation generalized everywhere to the equality under I relation. Simplification of a function application $f(e_1,...,e_n)$ can unfold the application if the resulting expression (or its context) can be computed as least as fast (through appropriate simplification corresponding to the context). To automate this in practice, heuristic conditions such as the following are used:

1) f is not recursively defined and unfolding $f(e_1,...,e_n)$ does not duplicate non-trivial computations, i.e., for each i, either $t(e_i) \leq t(v_i)$ or v_i occurs at most once on every (syntactic) execution path in e_f.

2) $f(e_1,...,e_n)$ is an argument to a primitive function p and this application of p can be simplified after unfolding f using properties of p.

Simplification of a binding expression **let** $v = e_1$ **in** e_2 unfolds it to $e_2[v_1/v]$ if e_1 equals a variable v_1 under I. Otherwise, it treats the expression as $f(e_1)$, with f defined by $f(v) = e_2$.

Given an expression e and an information set I, we define $\mathcal{S}imp[\![e]\!]I$ to be e' if e can be simplified under I to e', and e otherwise.

expression e	expression e'	condition $cond(I)$
v	c	$v \leftrightarrow_I^* c$
$c(e_1, ..., e_n)$	e_c	$e \leftrightarrow_I^* c(\bar{c}_1^{-1}(e_c), ..., \bar{c}_n^{-1}(e_c))$ and $t(e_c) \leq t(e)$
$p(e_1, ..., e_n)$	e_p	e can be simplified to e_p under I using properties of p
$f(e_1, ..., e_n)$	$e_f[e_1/v_1, \ldots, e_n/v_n]$	e can be unfolded under I and f is defined as $f(v_1,...,v_n) = e_f$
if e_1 **then** e_2 **else** e_3	e_2	$e_1 \leftrightarrow_I^* T$
	e_3	$e_1 \leftrightarrow_I^* F$
	e_2 (or e_3)	$e_2 \leftrightarrow_I^* e_3$ and $t(e_2) \leq t(e_3)$ (or $t(e_3) \leq t(e_2)$)
let $v = e_1$ **in** e_2	$e_2[v_1/v]$	$e_1 \leftrightarrow_I^* v_1$
	$e_2[e_1/v]$	e can be unfolded under I

Figure 3.2: Simplification

3.2.2 Cache set and replacement

A cache set C for an unfolded application is a set of tuples $\langle e_1, e_2, I \rangle$ such that

1) expression e_1 depends only on x, expression e_2 depends only on r, and
2) if the equations in information set I hold, then e_1 and e_2 are equal.

For example, if $f_0(x \oplus y)$ is unfolded to be e, then the initial cache set for e is $\{\langle f_0(x), r, \emptyset \rangle\}$.

Intuitively, an element $\langle e_1, e_2, I \rangle$ in a cache set C says that if the equations in I hold, then the value of e_1 can be retrieved from a cached result by computing e_2. Given a cache set and an occurrence of an expression e with information set $I_{[e]}$, we can extend the cache set at e under $I_{[e]}$. This extension requires techniques for discovering more expressions whose values can be retrieved from cached results, i.e., discovering incrementality, as described below.

Discovering incrementality. The schematic diagrams of Figure 3.3 help explain the basic ideas. The leftmost rectangle depicts the expanded computation of $f_0(x \oplus y)$. Clearly, if $f_0(x)$ occurs anywhere as a subcomputation, then its value can be straightforwardly retrieved from r. However, we seek to discover other subcomputations whose values can also be retrieved from r. Suppose $g(e_1)$ occurs somewhere as a subcomputation and it is not $f_0(x)$. If we collect the context information I_1 at the occurrence of $g(e_1)$ and find that $f_0(x)$ can be specialized to $g(e_1)$ under I_1, as depicted in the middle rectangle, then the value of $g(e_1)$ at the occurrence can also be retrieved from r. Moreover, if g is a function with an inverse $\bar{g}^{-1}$, then the value of e_1 can be retrieved from $\bar{g}^{-1}(r)$, wherever I_1 holds. In a special situation, suppose $h(e_2)$ occurs as a subcomputation but neither $h(e_2)$ nor e_2 is $f_0(x)$, $g(e_1)$, or e_1. If h is

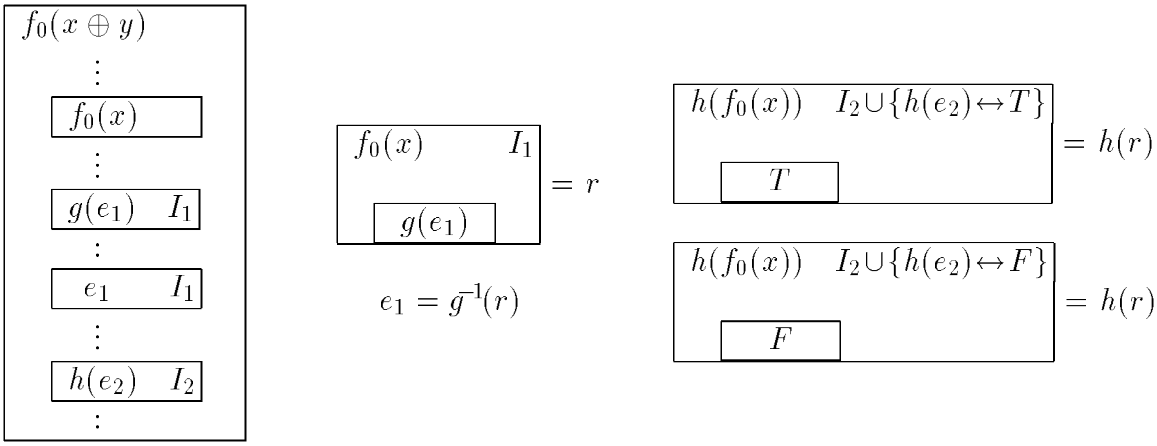

Figure 3.3: Discovering incrementality

a Boolean valued function defined on all inputs, and $h(f_0(x))$ can be specialized to true (false) when $h(e_2)$ equals true (false), as depicted in the right rectangles, then the value of $h(e_2)$ can be retrieved from $h(r)$.

The specializations shown in the middle and right rectangles in Figure 3.3 employ an auxiliary specializer $\mathcal{G}$. Given an expression e and an information set I, $\mathcal{G}[\![e]\!]I$ specializes e under I, and whenever e computes a value, $\mathcal{G}[\![e]\!]I$ computes the same value. The specialization, achieved by unfolding and simplification, will be defined at the end of this section. Here, we use $\mathcal{G}$ as a subroutine to help discover more subcomputations whose values can be retrieved from cached results.

We formalize the basic ideas as follows. Given a cache set C and an occurrence of an expression e with information set $I_{[e]}$, we can extend C at e under $I_{[e]}$ to be $\mathcal{C}(C, e, I_{[e]})$, where $\mathcal{C}(C, e, I)$, called the *closure of C at e under I*, is defined as follows. Given C, e, and I, let

$$\begin{array}{l} F_1(C) = \{\langle e_1', \quad e_2, \quad I\rangle \mid \langle e_1, e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![e_1]\!]I = e_1'\}, \\ F_2(C) = \{\langle e_1', g^{-1}(e_2), I\rangle \mid \langle e_1, e_2, I'\rangle \in C,\ I \Rightarrow I',\ e_1 = g(e_1')\},\ \text{and} \\ F_3(C) = \{\langle e, \ h(e_2),\ I\rangle \mid \langle e_1, e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![h(e_1)]\!](I \cup \{e \leftrightarrow {}^{T}_{F}\}) = {}^{T}_{F}\}, \end{array} \tag{3.3}$$

where g is a constructor or primitive function that has a known inverse g^{-1}, and h is a total Boolean valued function. Note that g and h can be straightforwardly generalized to include functions with two or more parameters. As an example for F_2, if $e_1 = c(e_1', ..., e_n')$, then $\langle e_i', c_i^{-1}(e_2), I\rangle \in F_2(C)$ for $i = 1..n$. Just as $F_2(C)$ extends C when e_1 is a constructor or a special primitive function application, the following sets extend C when e_1 is a binding expression or a special conditional expression:

$$\begin{array}{l} F_{21}(C) = \{\langle e_2'', e_2, I\rangle \mid \langle \textbf{let}\ v = e_1'\ \textbf{in}\ e_2', \quad e_2, I'\rangle \in C,\ I \Rightarrow I',\ e_2'[e_1'/v] = e_2''\}, \\ F_{22}(C) = \{\langle e_3', e_2, I\rangle \mid \langle \textbf{if}\ e_1'\ \textbf{then}\ e_2'\ \textbf{else}\ e_3', e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![e_3']\!](I \cup \{e_1' \leftrightarrow T\}) = e_2'\}, \\ F_{23}(C) = \{\langle e_2', e_2, I\rangle \mid \langle \textbf{if}\ e_1'\ \textbf{then}\ e_2'\ \textbf{else}\ e_3', e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![e_2']\!](I \cup \{e_1' \leftrightarrow F\}) = e_3'\}. \end{array} \tag{3.4}$$

We let $F_2(C)$ also include the elements in these sets, and define $\mathcal{C}(C, e, I)$ to be $C \cup C' \cup F_3(C \cup C')$, where C' is the least set such that $F_1(C) \subseteq C'$ and $F_2(C') \subseteq C'$.

Set C' can be computed using a worklist algorithm. First, initialize C' to be $\emptyset$ and worklist L to be $F_1(C)$. Then, repeatedly move any element $\langle e_1, e_2, I' \rangle$ from L to C' and, if e_1 is $g(e_1')$, add $\langle e_1', g^{-1}(e_2), I' \rangle$ to L, if e_1 is **let** $v = e_1'$ **in** e_2', add $\langle e_2'[e_1'/v], e_2, I' \rangle$ to L, *etc.* This stops when L is empty, at which time we have obtained the final set C'. Optimizations to the computation of the closure are possible. For example, we can group elements that have the same information sets into units, and maintain a tree of these units so that, if $I_1 \Rightarrow I_2$, then the unit with I_1 is a descendent of the one with I_2. Every time we compute the closure, we only need to look at elements in the units that are closest to the leaves and whose information sets are implied by the current information set.

Example. Using the example in Figure 2.1, let e be the unfolded application of $out(C, ins(i, a, R))$

$$\begin{array}{l} \textbf{if } null(C) \textbf{ then } nil \\ \textbf{else } cons(row(car(C), ins(i,a,R)),\ out(cdr(C), ins(i,a,R))) \end{array} \tag{3.5}$$

with information set $I_{[e]} = \emptyset$ and initial cache set $C_{out} = \{\langle out(C,R), r, \emptyset \rangle\}$.

Let e_1 be the false branch of e. Given $C_{out} = \{\langle out(C,R), r, \emptyset \rangle\}$, consider extending C_{out} at e_1 with $I_{[e_1]} = \{null(C) \leftrightarrow F\}$. Specializing $out(C, R)$ under $I_{[e_1]}$, we get the expression e_2 below:

$$cons(row(car(C), R),\ out(cdr(C), R)) \tag{3.6}$$

Thus,

$$\begin{aligned} \mathcal{C}(C_{out}, e_1, I_{[e_1]}) = C_{out} \cup \{ & \langle e_2,\ r,\ I_{[e_1]} \rangle, \\ & \langle row(car(C), R),\ car(r),\ I_{[e_1]} \rangle, \\ & \langle out(cdr(C), R),\ cdr(r),\ I_{[e_1]} \rangle \}. \end{aligned} \tag{3.7}$$

Given $C_{out} = \{\langle out(C,R), r, \emptyset \rangle\}$, consider extending C_{out} at the Boolean subexpression $null(C)$ in e with $I_{[null(C)]} = \emptyset$. When $null(C) \leftrightarrow T$ holds, $out(C, R)$ is specialized to nil and thus $null(out(C, R))$ equals T; when $null(C) \leftrightarrow F$ holds, $out(C, R)$ is specialized to e_2 and thus $null(out(C, R))$ equals F. Thus

$$\mathcal{C}(C_{out}, null(C), \emptyset) = C_{out} \cup \{\langle null(C), null(r), \emptyset \rangle\}. \tag{3.8}$$

Replacement. We say that expression e *can be replaced by* e' *under* I *and* C, denoted as $e \rightarrow^*_{IC} e'$, if

$$(\ \exists \langle e_1, e', I_1 \rangle \in C\ )\ [\, e \leftrightarrow^*_I e_1\ \wedge\ I \Rightarrow I_1\ \wedge\ t(e') \leq t(e) \,].$$

Given a non-conditional and non-binding expression e, an information set I, and a cache set C, if e can be replaced by e' under I and C, then we do so. Otherwise, we extend the cache set to be $\mathcal{C}(C, e, I)$, and, if e can be replaced by e'' under I and the extended cached set $\mathcal{C}(C, e, I)$, then we do so. As a result, the cache set may be extended as a side effect of a replacement. We define a function $\mathcal{R}epl$ for replacement as follows:

$$\mathcal{R}epl[\![e]\!]IC = \begin{cases} \langle e, C\rangle & \text{if } e \text{ is a conditional or binding expression} \\ \langle e', C\rangle & \text{else if } e \rightarrow^*_{IC} e' \\ \langle e'', C'\rangle & \text{else if } e \rightarrow^*_{IC'} e'', \ \text{ where } C' = \mathcal{C}(C, e, I) \\ \langle e, C'\rangle & \text{otherwise, } \ \text{where } C' \text{ is as above} \end{cases} \tag{3.9}$$

Another use of a cache set for replacement is as follows. Suppose an expression e can not be replaced by any expression under I and C, but

$$\exists\, \langle \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3,\ e_4,\ I_1\rangle \in C, \quad I \Rightarrow I_1$$

such that e can be replaced by e_T(respectively, e_F) under $I \cup \{e_1 \leftrightarrow T\}$ (respectively, $I \cup \{e_1 \leftrightarrow F\}$) and the correspondingly extended cache set. Then we can replace e by **if** e_1**then** e_T **else** e_F provided $t(\textbf{if } e_1 \textbf{then } e_T \textbf{ else } e_F) \leq t(e)$. For example, if e is e_3, and e_1 takes unit time, then we can replace e by **if** e_1**then** e **else** e_4. We extend function $\mathcal{R}epl$ for replacement to include this case, i.e., we replace the last case of (3.9) by the following two cases:

$$\begin{cases} \langle \textbf{if } e'_1 \textbf{ then } e_T \textbf{ else } e_F,\ C'''\rangle \\ \quad \text{if } \exists\, \langle \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3,\ e_4,\ I_1\rangle \in C',\ I \Rightarrow I_1,\ t(\textbf{if } e'_1 \textbf{ then } e_T \textbf{ else } e_F) \leq t(e) \\ \qquad\qquad \text{where } e'_1 = \begin{cases} e''_1 & \text{if } e_1 \rightarrow^*_{IC''} e''_1 \\ e_1 & \text{otherwise} \end{cases} \text{ where } C'' = \mathcal{C}(C', e_1, I) \\ \qquad\qquad\qquad e_B = \begin{cases} e'_B & \text{if } e \rightarrow^*_{I_B C'''} e'_B \\ e & \text{otherwise} \end{cases} \text{where } I_B = I \cup \{e'_1 \leftrightarrow B\},\ \text{for } B = T, F \\ \qquad\qquad\qquad C''' = \mathcal{C}(C'', e, I_T) \cup \mathcal{C}(C'', e, I_F) \\ \langle e, C'''\rangle \\ \quad \text{otherwise, where } C''' \text{ is as above} \end{cases} \tag{3.10}$$

3.2.3 Incrementalization by simplification and replacement

To incrementalize an unfolded application, a function $\mathcal{I}nc$ applies simplification and replacement on subexpressions in applicative order. The cache set for the current unfolded application may be extended as a side effect of replacement using $\mathcal{R}epl$. In particular, $\mathcal{I}nc$ calls $\mathcal{I}nc\mathcal{A}pply$ to consider subexpressions that are function applications. The global set of introduced functions may be extended as a side effect of using $\mathcal{I}nc\mathcal{A}pply$.

We refer to the application of simplification and replacement by $\mathcal{I}nc$ as *reduction*. Thus, $\mathcal{I}nc$ does innermost leftmost reduction. If a subexpression is reduced to a conditional expression, then the condition is lifted out of the enclosing expression. Similarly, if a subexpression is reduced to a binding expression, then the binding is lifted. A function $\mathcal{S}ubl$ is used by $\mathcal{I}nc$ to recursively reduce subexpressions and perform necessary lifting, as defined in Figure 3.4.

The presentation of $\mathcal{S}ubl$ is simplified by omitting detailed control structures that sequence $\mathcal{S}ubl$ through its subexpressions. We just present the case of $\mathcal{S}ubl$ working on the ith subexpression of the top-level construct under the assumption that the subexpressions 1 through $i-1$ have been *reduced*. Operationally, we say that a subexpression is reduced if it is the result of having already applied $\mathcal{I}nc$ for the subexpression at that position; otherwise, it is not reduced. For a conditional expression **if** e_1**then** e_2 **else** e_3, $\mathcal{I}nc$ reduces e_2 (respectively, e_3) with the assumption that e_1 equals true (respectively, false) added to the information set. Similarly, for a binding expression **let** $v = e_1$ **in** e_2, $\mathcal{I}nc$ reduces e_2 with the assumption that v equals e_2 added to the information set. At the end, all subexpressions are reduced with necessary lifting performed.

Name	Transformation	Condition
(g)	$\mathcal{S}ubl[\![g(e_1,...,e_n)]\!]\ I\ C\ D$ where g is c, p, or f	
(g_i)	$= \mathcal{S}ubl[\![g(e_1,...,e_{i-1},e'_i,e_{i+1},...,e_n)]\!]\ I\ C'\ D'$ where $\langle e'_i, C', D'\rangle = \mathcal{I}nc[\![e_i]\!]\ I\ C\ D$	if $e_1,...,e_{i-1}$ are reduced, not **if** or **let**, but e_i is not reduced
$(g_{i-\mathit{if}})$	$= \mathcal{S}ubl[\![$**if** e'_1 **then** $g(e_1,...,e_{i-1},e'_2,e_{i+1},...,e_n)$ **else** $g(e_1,...,e_{i-1},e'_3,e_{i+1},...,e_n)]\!]\ I\ C\ D$ where e'_1 is reduced and is not **if** or **let**	if $e_1,...,e_{i-1}$ are reduced, not **if** or **let**, e_i is reduced, but e_i is **if** e'_1 **then** e'_2 **else** e'_3
$(g_{i-\mathit{let}})$	$= \mathcal{S}ubl[\![$**let** $v = e'_1$ **in** $g(e_1,...,e_{i-1},e'_2,e_{i+1},...,e_n)]\!]\ I\ C\ D$ where e'_1 is reduced and is not **if** or **let**	if $e_1,...,e_{i-1}$ are reduced, not **if** or **let**, e_i is reduced, but e_i is **let** $v = e'_1$ **in** e'_2
(g_n)	$= \langle g(e_1,...,e_n),\ C,\ D\rangle$	otherwise
(if)	$\mathcal{S}ubl[\![$**if** e_1 **then** e_2 **else** $e_3]\!]\ I\ C\ D$	
(if_1)	$= \mathcal{S}ubl[\![$**if** e'_1 **then** e_2 **else** $e_3]\!]\ I\ C'\ D'$ where $\langle e'_1, C', D'\rangle = \mathcal{I}nc[\![e_1]\!]\ I\ C\ D$	if e_1 is not reduced
$(\mathit{if}_{1-\mathit{if}})$	$= \mathcal{S}ubl[\![$**if** e'_1 **then** (**if** e'_2 **then** e_2 **else** e_3) **else** (**if** e'_3 **then** e_2 **else** e_3)$]\!]\ I\ C\ D$ where e'_1 is reduced and is not **if** or **let**	if e_1 is reduced, and e_1 is **if** e'_1 **then** e'_2 **else** e'_3
$(\mathit{if}_{1-\mathit{let}})$	$= \mathcal{S}ubl[\![$**let** $v = e'_1$ **in** (**if** e'_2 **then** e_2 **else** e_3)$]\!]\ I\ C\ D$ where e'_1 is reduced and is not **if** or **let**	if e_1 is reduced, and e_1 is **let** $v = e'_1$ **in** e'_2
(if_n)	$= \langle$**if** e_1 **then** e'_2 **else** $e'_3,\ C'',\ D''\rangle$ where $\langle e'_2, C', D'\rangle = \begin{cases} \langle e_2,\ C,\ D\rangle & \text{if } e_1 \leftrightarrow^*_I F \\ \mathcal{I}nc[\![e_2]\!]\ (I \cup \{e_1 \leftrightarrow T\})\ C\ D & \text{otherwise} \end{cases}$ $\langle e'_3, C'', D''\rangle = \begin{cases} \langle e_3,\ C',\ D'\rangle & \text{if } e_1 \leftrightarrow^*_I T \\ \mathcal{I}nc[\![e_3]\!]\ (I \cup \{e_1 \leftrightarrow F\})\ C'\ D' & \text{otherwise} \end{cases}$	otherwise
(let)	$\mathcal{S}ubl[\![$**let** $v = e_1$ **in** $e_2]\!]\ I\ C\ D$	
(let_1)	$= \mathcal{S}ubl[\![$**let** $v = e'_1$ **in** $e_2]\!]\ I\ C'\ D'$ where $\langle e'_1, C', D'\rangle = \mathcal{I}nc[\![e_1]\!]\ I\ C\ D$	if e_1 is not reduced
$(\mathit{let}_{1-\mathit{if}})$	$= \mathcal{S}ubl[\![$**if** e'_1 **then** (**let** $v = e'_2$ **in** e_2) **else** (**let** $v = e'_3$ **in** e_2)$]\!]\ I\ C\ D$ where e'_1 is reduced and is not **if** or **let**	if e_1 is reduced, and e_1 is **if** e'_1 **then** e'_2 **else** e'_3
$(\mathit{let}_{1-\mathit{let}})$	$= \mathcal{S}ubl[\![$**let** $v' = e'_1$ **in** (**let** $v = e'_2$ **in** e_2)$]\!] I\ C\ D$ where e'_1 is reduced and is not **if** or **let**	if e_1 is reduced, and e_1 is **let** $v' = e'_1$ **in** e'_2
(let_n)	$= \langle$**let** $v = e_1$ **in** $e'_2,\ C',\ D'\rangle$ where $\langle e'_2, C', D'\rangle = \mathcal{I}nc[\![e_2]\!]\ (I \cup \{v \leftrightarrow e_1\})\ C\ D$	otherwise

Figure 3.4: Definition of $\mathcal{S}ubl$

Finally, we define function $\mathcal{I}nc$ as in (3.11), where I^C denotes the set $I \cup \{e_1 \leftrightarrow$

$e_2 \mid \langle e_1, e_2, I' \rangle \in C, I \Rightarrow I' \}$. We need I^C instead of I because $\mathcal{I}nc$ does applicative order reduction, during which some subexpressions may be replaced by retrievals, and thus the equations in I may involve the cache parameter. As a result, the underlying logic needs to know the equality relation involving the cache parameter to make inferences. For example, to reduce the expression e in (3.5), first $null(C)$ is reduced to $null(r)$ according to (3.8), and thus $I_{[e_1]} = \{null(r) \leftrightarrow F\}$ for the expression e_1 in the false branch of e. Now to specialize $out(C, R)$ at e_1, we use the information set $\{null(r) \leftrightarrow F\} \cup \{null(x) \leftrightarrow null(r)\}$, and we obtain the same expression as in (3.6).

$$
\begin{aligned}
\mathcal{I}nc[\![e]\!]ICD = \langle e''', C'', D'' \rangle & \\
\text{where } \langle e''', C'' \rangle &= \mathcal{R}epl[\![\mathcal{S}imp[\![e'']\!]I^{C'}]\!]I^{C'}C' \\
\langle e'', D'' \rangle &= \begin{cases} \mathcal{I}nc\mathcal{A}pply[\![e']\!]I^{C'}C'D' & \text{if } e' \text{ is } f(e_1, ..., e_n) \\ \langle e', D' \rangle & \text{otherwise} \end{cases} \\
\langle e', C', D' \rangle &= \begin{cases} \mathcal{S}ubl[\![e]\!]ICD & \text{if } e \text{ is not } v \\ \langle e, C, D \rangle & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.11}
$$

Function $\mathcal{I}nc$ proceeds as follows. First, if an expression e has subexpressions, then $\mathcal{I}nc$ calls $\mathcal{S}ubl$ to recursively reduce the subexpressions in turn. The cache set and definition set may be changed while reducing subexpressions. Then if the resulting expression is a function application, $\mathcal{I}nc$ calls $\mathcal{I}nc\mathcal{A}pply$ and aims to replace the application with an application of an introduced function that computes incrementally. The definition set may be changed by $\mathcal{I}nc\mathcal{A}pply$. Finally, $\mathcal{I}nc$ uses $\mathcal{S}imp$ to simplify the top-level expression, and then calls $\mathcal{R}epl$ to replace the resulting expression by a retrieval from a cached result, if possible. The cache set may be changed by $\mathcal{R}epl$.

Auxiliary specializer. The auxiliary specializer $\mathcal{G}$ is defined in a way similar to $\mathcal{I}nc$, but it is much simpler. It simplifies subexpressions in applicative order and lifts conditions and bindings as $\mathcal{I}nc$ does, but there are no cache sets or definitions sets involved. If simplification of a function application unfolds the application, then $\mathcal{G}$ is applied to the unfolded application. Let $\mathcal{S}ubl'$ be $\mathcal{S}ubl$ except that $\mathcal{S}ubl'$ takes only an expression and an information set as arguments, returns only an expression, and calls $\mathcal{G}$ instead of $\mathcal{I}nc$. Then $\mathcal{G}$ is defined as follows:

$$
\begin{aligned}
\mathcal{G}[\![e]\!]I &= \begin{cases} \mathcal{G}[\![e'']\!]I & \text{if } e' \text{ is } f(e_1, ..., e_n) \text{ and } e'' \neq e' \\ e'' & \text{otherwise} \end{cases} \\
\text{where } e'' &= \mathcal{S}imp[\![e']\!]I \\
e' &= \begin{cases} \mathcal{S}ubl'[\![e]\!]I & \text{if } e \text{ is not } v \\ e & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.12}
$$

3.3 Manipulating recursive function applications

We define the *definition set*, which is a global set of functions introduced during the derivation procedure to compute function applications incrementally. We describe how to maintain the definition set when introducing functions and how to use the introduced functions to replace appropriate function applications.

3.3.1 Definition set

Definition set D is a set of tuples $\langle f(e_1,...,e_n),\ f'(v_1,...,v_k),\ C\rangle$, where C is a single-element cache set $\{\langle e_c, e_r, I\rangle\}$, such that

1) f is a function in the original set F, expressions $e_1,...,e_n$ depend on x and possibly on y, f' is a new function introduced in the set D, and $v_1,...,v_k$ are variables in $e_1,...,e_n,e_c$, and e_r,
2) if the cache set C is valid, i.e., the equations in the information set I hold, and $e_c = e_r$, then whenever $f(e_1,...,e_n)$ terminates with a value, $f'(v_1,...,v_k)$ terminates with the same value, and
3) a definition of f' is obtained by incrementalizing the unfolded $f(e_1,...,e_n)$ using I and C, and some of the parameters of f' may be dead and eliminated after the incrementalization.

For example, given $out(C,R) = r$ with empty information set at the initial application $out(C, ins(i,a,R))$, we introduce a new function out', and we get the initial definition set

$$\{\langle out(C, ins(i,a,R)),\ out'(C,i,a,R,r),\ \{\langle out(C,R), r, \emptyset\rangle\}\rangle\} \tag{3.13}$$

where a definition of out' is to be obtained by incrementalizing the unfolded $out(C, ins(i,a,R))$ using $out(C,R) = r$.

Intuitively, an element in the definition set D says that a new function f' is introduced such that, if the equations in the information set I hold, and the value of e_c can be retrieved from a cached result by computing e_r, then $f'(v_1,...,v_k)$ computes $f(e_1,...,e_n)$ incrementally. To obtain a definition of f', we unfold $f(e_1,...,e_n)$, incrementalize the unfolded application using the sets I and C, and then eliminate dead parameters. While we incrementalize the unfolded $f(e_1,...,e_n)$, we may encounter other function applications before we obtain a final definition of f'. We say f' is *fully defined* if, for every introduced function g' in D that f' (transitively) depends on, a final definition of g' has been obtained.

Note the restriction that cache set C contains only *one* element, which reflects our main heuristic for introducing new functions. In general, a function application has its context information set and a current cache set. Any element in these sets might be used in incrementalizing the unfolded application. But we do not know, before examining the unfolded application, what elements are used and how. Therefore, any dynamic decision must be an approximation. Our one-cache-element heuristic is based on the observation that, in a well-structured program, a function application is expected to be computed incrementally based on the cached result of a corresponding previous computation. As a consequence of our way of choosing the single cache element, as described below, there is only one variable in the expression e_r. This variable depends on r and is introduced as a parameter of f'. We call it the *current cache parameter* during the process of incrementalizing the unfolded $f(e_1,...,e_n)$.

A function f may correspond to multiple introduced functions, since there may be multiple occurrences of applications of f during the derivation, and different applications may correspond to different introduced functions.

3.3.2 Generalization for function introduction

Given a function application $f(e_1, ..., e_n)$, let I be the information set at $f(e_1, ..., e_n)$, and C the current cache set for the unfolded application that contains $f(e_1, ..., e_n)$. To introduce a function f' to compute $f(e_1, ..., e_n)$ incrementally, the main task is to decide, based on I and C, a valid and relevant cache element that is to be used to incrementalize the computation of $f(e_1, ..., e_n)$. An interaction with this comes from using a version of *generalization* [Tur86] that enables f' to be used in more general settings and, at the same time, does not impede the discovery of incrementality.

Considerations. Our use of generalization ignores substructures of expressions to introduce functions for more general uses. For example, consider the function application

$$row(car(C), ins(i, a, R)) \quad \text{with} \quad \langle row(car(C), R),\ car(r),\ \{null(C) \leftrightarrow F\}\rangle \in C_{out} \tag{3.14}$$

and $I = \{null(C) \leftrightarrow F\}$ in the false branch of (3.5). Instead of introducing

$$\begin{aligned}\langle row(car(C), ins(i, a, R)),\ & row'(C, i, a, R, r), \\ & \{\langle row(car(C), R),\ car(r),\ \{null(C) \leftrightarrow F\}\rangle\}\rangle\end{aligned} \tag{3.15}$$

and replacing the application by $row'(C, i, a, R, r)$, we introduce

$$\langle row(c, ins(i, a, R)),\ row'(c, i, a, R, r_1),\ \{\langle row(c, R),\ r_1,\ I'\rangle\}\rangle \tag{3.16}$$

where $I' = \{null(C) \leftrightarrow F, car(C) \leftrightarrow c\}$, and replace the application by $row'(car(C), i, a, R, car(r))$. We say that c generalizes $car(C)$, and r_1 generalizes $car(r)$. Obviously, the latter row' is more general than the former and can be used in more general settings.

Basically, the largest common super-expression of all occurrences of a variable is generalized by a single (new) variable. However, there are two considerations. First, generalization should not impede the discovery of incrementality. For example, if we consider $row(car(C), ins(i, a, R))$ in (3.14), then $ins(i, a, R)$ is not generalized by a variable, since we want to separate subcomputations depending only on x from the rest so that the former can possibly be replaced by retrievals. Therefore, one guideline is to generalize as much as possible without crossing the boundary between subexpressions depending only on x and the rest.

The second consideration is associated with the main task of deciding a valid and most relevant cache element to be used to incrementalize the computation of $f(e_1, ..., e_n)$. For example, among the valid cache elements in (3.7), the element in (3.14) is used to incrementalize $row(car(C), ins(i, a, R))$. To arrive at this choice, consider $row(car(C),\ ins(i, a, R))$ together with the two expressions in a cache element. With the element in (3.14), we can generalize more than with any other element in (3.7). Also, the information set becomes I', as in (3.16), since it relates I with the new variable c. Therefore, the guideline is to generalize the function application together with the two expressions in each valid cache element, choose the element that allows most generalization, and relate the information set with the new variables.

To summarize, our use of generalization does not impede the discovery of incrementality and helps obtain the most relevant cache element. We should note that these are online techniques for the generalization.

Generalization. We present the above ideas formally as follows. Given expressions $e_1, ..., e_m$, let U be the set of variables in them. Let $\{u\} \cup U_1 \subseteq U$ where $u \notin U_1$. An expression e is the *largest common* $u \backslash U_1$*-cover* of $e_1, ..., e_m$ if e is the largest common super-expression of all occurrences of u in the e_i's such that the elements of U_1 do not appear in e.

Given $f(e_1, ..., e_n)$ with I and C, suppose u^r is the current cache parameter. Let $\{u_1^x, ..., u_p^x, u_1^y, ..., u_q^y\}$ be the set of variables other than u^r in $e_1, ..., e_n$ where u_j^x's depend only on x and u_k^y's depend possibly on y. Let $\langle e_c, e_r, I_1 \rangle$ be any element in C such that $I \Rightarrow I_1$ and all the variables in e_c are in $\{u_1^x, ..., u_p^x\}$, and thus the element is valid and relevant.

Let E be a set of non-overlapping expressions e such that e is the largest common $u^r \backslash \{u_1^x, ..., u_p^x, u_1^y, ..., u_q^y\}$-cover, $u_j^x \backslash \{u^r, u_1^y, ..., u_q^y\}$-cover, or $u_k^y \backslash \{u^r, u_1^x, ..., u_p^x\}$-cover of $e_1, ..., e_n, e_c$, and e_r for some u_j^x or u_k^y. Let

$$\theta = \{e/v \mid e \in E\} \tag{3.17}$$

where v's are distinct new variable names,[2] then θ is a substitution corresponding to these non-overlapping largest common covers. Using the inverse substitution $\theta^{-1} = \{v/e \mid e/v \in \theta\}$, we obtain expressions $e'_1, ..., e'_n, e'_c$, and e'_r such that $e'_i = e_i\theta^{-1}$ for $i = 1, ..., n, c, r$, and we obtain an information set I' such that I' is $I\theta^{-1}$ extended with equations induced by θ that are relevant to $I\theta^{-1}$, i.e.,

$$I' = I\theta^{-1} \cup \{e \leftrightarrow v \mid e/v \in \theta, \text{ a variable in } e \text{ occurs in } I\theta^{-1}\}.$$

We say that $\langle e'_1, ..., e'_n, e'_c, e'_r, I' \rangle$ is a *generalization* for $\langle f(e_1, ..., e_n), I, C \rangle$ with *substitution* θ. For any e/v in θ, we say variable v *generalizes* expression e. It is clear that such a generalization obeys the first consideration.

A tuple $\langle f(e_1, ..., e_n), I, C \rangle$ may have more than one generalization if there is more than one element in C or there is more than one set of non-overlapping largest common covers for a cache element. Suppose A_1 and A_2 are two generalizations with substitutions θ_1 and θ_2, respectively. We say A_1 is *more general* than A_2 if every expression in $\{e \mid e/v \in \theta_2\}$ is a subexpression of some expression in $\{e \mid e/v \in \theta_1\}$. We say A_1 is *most general* if no other generalization is more general. This incorporates the second consideration.

3.3.3 Function introduction and replacement

Given a function application $f(e_1, ..., e_n)$, let I be the information set at $f(e_1, ..., e_n)$, C the current cache set for the unfolded application that contains $f(e_1, ..., e_n)$, and

[2] If an expression e in set E is a variable u, then the corresponding variable v can be u, i.e., v does not have to be a new variable in this case.

D the current definition set. If we can use a previously introduced function f' in D to compute $f(e_1, ..., e_n)$ incrementally, then $f(e_1, ..., e_n)$ is replaced by an application of f'. Otherwise, we introduce a new function f' into D to compute $f(e_1, ..., e_n)$ incrementally and, if f' computes fast, replace $f(e_1, ..., e_n)$ by an application of this f', otherwise, leave $f(e_1, ..., e_n)$ unchanged; as a result, the definition set is changed as a side effect. This process is achieved by $\mathcal{I}nc\mathcal{A}pply$, first introduced in Section 3.1. It is defined in Figure 3.5 and explained below.

$\mathcal{I}nc\mathcal{A}pply[\![f(e_1, ..., e_n)]\!]\ I\ C\ D$

$= \langle f'(v_{i_1}\theta, ..., v_{i_j}\theta),\ D\rangle$
  if $\exists\,\langle f(e'_1, ..., e'_n), f'(v_{i_1}, ..., v_{i_j}), \{\langle e'_c, e'_r, I'\rangle\}\rangle \in D$ with a substitution θ s.t.
    $f(e_1, ..., e_n) \leftrightarrow^*_I f(e'_1\theta, ..., e'_n\theta),\ \ I \Rightarrow I'\theta,\ \ e'_c\theta \rightarrow^*_{IC} e'_r\theta,$ and
    if f' is fully defined, $t(f'(v_{i_1}\theta, ..., v_{i_j}\theta)) \leq t(f(e_1, ..., e_n))$;

$= \langle f'(v_{i_1}\theta, ..., v_{i_j}\theta), D'''\rangle$
  else if $f(e_1, ..., e_n)$ depends on x but can not be replaced by a retrieval, and,
    after obtaining a definition of f' as follows,
    if f' is fully defined, $t(f'(v_{i_1}\theta, ..., v_{i_j}\theta)) \leq t(f(e_1, ..., e_n))$;

    introduce $d = \langle f(e'_1, ..., e'_n), f'(v_1, ..., v_k), C'\rangle$ with θ, where $C' = \{\langle e'_c, e'_r, I'\rangle\}$,
    and obtain $f'(v_{i_1}, ..., v_{i_j})$, to be defined as some e''', by the following steps:
      1) $e' = e[e'_1/v_1, ..., e'_n/v_n]$, where f is defined by $f(v_1, ..., v_n) = e$
      2) $\langle e'', C'', D''\rangle = \mathcal{I}nc[\![e']\!]\ I'\ C'\ D'$, where $D' = D \cup \{d\}$
      3) $\langle f'(v_{i_1}, ..., v_{i_j}), D'''\rangle = \mathcal{E}lim[\![f'(v_1, ..., v_k)]\!]\ D''$, where $f'(v_1, .., v_k)$ is
        defined as e''

$= \langle f(e_1, ..., e_n),\ D''''\rangle$
  otherwise, where D'''' is D''' as above if it is computed and D otherwise

Figure 3.5: Definition of $\mathcal{I}nc\mathcal{A}pply$

Function replacement. Since an introduced function f' is associated with a cache element as an invariant, to use f', we need to justify the corresponding invariant. We say a function application $f(e_1, ..., e_n)$ with I, C, and D can be replaced by an application of a previously introduced function f' if there is a tuple $\langle f(e'_1,..., e'_n), f'(v_{i_1},...,v_{i_j}),$ $\{\langle e'_c, e'_r, I'\rangle\}\rangle$ in D and there is a substitution θ over the variables in $e'_1, ..., e'_n, e'_c, e'_r$, and I' such that

1) $f(e_1, ..., e_n)$ equals $f(e'_1\theta, ..., e'_n\theta)$, invariant $I'\theta$ holds, $e'_c\theta$ can be replaced by $e'_r\theta$, and

2) if f' is fully defined, $f'(v_{i_1}\theta,...,v_{i_j}\theta)$ is asymptotically at least as fast as $f(e_1,...,e_n)$.

In this case, $f(e_1, ..., e_n)$ can be replaced by $f'(v_{i_1}\theta, ..., v_{i_j}\theta)$, and definition set D remains unchanged.

For example, given the definition set (3.13), the application

$$out(cdr(C), ins(i, a, R)) \quad \text{with} \quad \langle out(cdr(C), R),\ cdr(r),\ \{null(C) \leftrightarrow F\}\rangle \in C_{out} \tag{3.18}$$

in the false branch of (3.5) can be replaced by $out'(cdr(C), i, a, R, cdr(r))$.

Function introduction. If $f(e_1, ..., e_n)$ with I, C, and D can not be replaced by an application of a previously introduced function in D, then we introduce a new function f' into D to compute $f(e_1, ..., e_n)$ incrementally. Following the basic derivation idea, we introduce f' only if $f(e_1, ..., e_n)$ depends on x but can not be replaced by a retrieval from a cached result.

Given $f(e_1, ..., e_n)$ with I and C, let $\langle e'_1, ..., e'_n, e'_c, e'_r, I' \rangle$ be a most general generalization with substitution θ. Let $v_1, ..., v_k$ be all the variables in $e'_1, ..., e'_n, e'_c$, and e'_r. We introduce $\langle f(e'_1, ..., e'_n), f'(v_1, ..., v_k), C' \rangle$, where $C' = \{\langle e'_c, e'_r, I' \rangle\}$, into D to get D', and we obtain a definition of f' by the following three steps:

1) unfold the application $f(e'_1, ..., e'_n)$ to obtain e';

2) incrementalize e' with information set I', cache set C', and definition set D' to get e'';

3) for f' defined by $f'(v_1, ..., v_k) = e''$, eliminate dead parameters of f'.

Note that the second step uses the function $\mathcal{I}nc$, which may use $\mathcal{I}nc\mathcal{A}pply$ recursively for function applications. After the third step, if we obtain $f'(v_{i_1}, ..., v_{i_j})$ and, if f' is fully defined, $t(f'(v_{i_1}\theta, ..., v_{i_j}\theta)) \leq t(f(e_1, ..., e_n))$, then we replace $f(e_1, ..., e_n)$ by $f'(v_{i_1}\theta, ..., v_{i_j}\theta)$. The set D is changed as a side effect.

Dead parameter elimination. After the second step above, $f'(v_1, ..., v_k)$ computes $f(e'_1, ..., e'_n)$ and is defined as e''. Since e'' is obtained by replacing some subcomputations of $f(e'_1, ..., e'_n)$ depending on x by computations depending on the current cache parameter, those parameters of f' on which the replaced computations depend may become dead.

Dead code elimination is a traditional optimization [MJ81,ASU86]. We assume a subroutine $\mathcal{E}lim$ is given so that $\mathcal{E}lim[\![f'(v_1, ..., v_k)]\!]D''$, where $f'(v_1, .., v_k)$ is defined as e'' in D'', returns the pair $\langle f'(v_{i_1}, ..., v_{i_j}), D''' \rangle$, where $1 \leq i_1 < ... < i_j \leq k$ and $f'(v_{i_1}, ..., v_{i_j})$ is defined as some e''' in D''' after dead parameter elimination, and $f'(v_{i_1}, ..., v_{i_j})$ returns a value if and only if $f'(v_1, ..., v_k)$ returns the same value.

Example. Consider the running example. For application $row(car(C), ins(i, a, R))$ in the false branch of (3.5), we introduce a new function row' as in (3.16). To obtain a definition of row', we first unfold $row(c, ins(i, a, R))$ to get

$$\begin{array}{l} \textbf{if } null(ins(i,a,R)) \textbf{ then } nil \\ \textbf{else } cons(c * car(ins(i,a,R)),\ row(c, cdr(ins(i,a,R)))) \end{array} \tag{3.19}$$

Then, we incrementalize (3.19) using $row(c, R) = r_1$ as given by the cache set in (3.16). The incrementalization is sketched as follows. It is easy to see that $ins(i, a, R)$ in the condition can be unfolded and the condition simplified to true, and thus (3.19) is reduced to

$$cons(c * car(ins(i,a,R)),\ row(c, cdr(ins(i,a,R)))) \tag{3.20}$$

The first occurrence of $ins(i, a, R)$ in (3.20) can be unfolded, conditions in the unfolded application lifted, and *car* of *cons* applications simplified. Thus, (3.20) becomes

$$\begin{array}{l}\textbf{if } i \leq 1 \textbf{ then } cons(c * a,\ row(c, cdr(ins(i, a, R)))) \\ \textbf{else if } null(R) \textbf{ then } cons(c * a,\ row(c, cdr(ins(i, a, R)))) \\ \textbf{else } cons(c * car(R),\ row(c, cdr(ins(i, a, R))))\end{array} \tag{3.21}$$

The three occurrences of $ins(i, a, R)$ in (3.21) can be specialized under their corresponding contexts, unfolded, and then *cdr* of *cons* applications simplified. Thus, (3.21) becomes

$$\begin{array}{l}\textbf{if } i \leq 1 \textbf{ then } cons(c * a,\ row(c, R)) \\ \textbf{else if } null(R) \textbf{ then } cons(c * a,\ row(c, nil)) \\ \textbf{else } cons(c * car(R),\ row(c, ins(i-1, a, cdr(R))))\end{array} \tag{3.22}$$

In the first branch of (3.22), $row(c, R)$ can be directly replaced by r_1. In the second branch, $row(c, nil)$ can be specialized and unfolded to nil. For the third branch, we have $null(R) \leftrightarrow F$; thus $row(c, R)$ is specialized to $cons(c * car(R),\ row(c, cdr(R)))$ and the cache set is extended so that

$$c * car(R) = car(r_1) \quad \text{and} \quad row(c, cdr(R)) = cdr(r_1).$$

Thus, $c * car(R)$ can be replaced by $car(r_1)$, and the application $row(c, ins(i-1, a, cdr(R)))$ can be replaced by $row'(c, i-1, a, cdr(R), cdr(r_1))$. Additionally, in a situation similar to (3.8), $null(R)$ can be replaced by $null(r_1)$. Thus, (3.22) is reduced to

$$\begin{array}{l}\textbf{if } i \leq 1 \textbf{ then } cons(c * a,\ r_1) \\ \textbf{else if } null(r_1) \textbf{ then } cons(c * a,\ nil) \\ \textbf{else } cons(car(r_1),\ row'(c, i-1, a, cdr(R), cdr(r_1)))\end{array} \tag{3.23}$$

Finally, for $row'(c, i, a, R, r_1)$ defined as (3.23), it is clear that the parameter R is dead and can be eliminated. We obtain the final definition of row' as given in Figure 3.1. The application $row(car(C), ins(i, a, R))$ can be replaced by $row'(car(C), i, a, car(r))$, since the latter is asymptotically at least as fast as the former.

To complete our example, for the initial application $out(C, ins(i, a, R))$, we introduce a new function out' as in (3.13). In incrementalizing the unfolded application of *out* as in (3.5), the Boolean expression $null(C)$ can be replaced by $null(r)$ due to (3.8), the application of *row* can be replaced by $row'(car(C), i, a, car(r))$ as just given above, and the recursive application of *out* can be replaced by $out'(cdr(C), i, a, R, cdr(r))$ as followed from (3.18). Therefore, the unfolded application (3.5) is reduced to

$$\begin{array}{l}\textbf{if } null(r) \textbf{ then } nil \\ \textbf{else } cons(row'(car(C), i, a, car(r)),\ out'(cdr(C), i, a, R, cdr(r)))\end{array} \tag{3.24}$$

For $out'(C, i, a, R, r)$ defined as (3.24), it is clear that the parameter R is dead and can be eliminated. We obtain the final definition of out' as given in Figure 3.1. Finally, the application $out(C, ins(i, a, R))$ can be replaced by $out'(C, i, a, r)$, i.e., given $out(C, R) = r$, $out'(C, i, a, r)$ computes $out(C, ins(i, a, R))$ and is at least as fast.

3.4 Summarizing the derivation procedure

The derivation procedure can be summarized as follows. Function $\mathcal{I}nc\mathcal{A}pply$ maintains global set D, introduces new functions to compute function applications incrementally, and replaces these applications by appropriate applications of introduced functions. $\mathcal{I}nc\mathcal{A}pply$ calls function $\mathcal{I}nc$, which maintains a cache set C, discovers subcomputations whose values can be retrieved from cached results, and incrementalizes the computation of an unfolded function by simplification and replacement using retrievals. $\mathcal{I}nc$ recursively calls $\mathcal{I}nc\mathcal{A}pply$ if a subcomputation is a function application. The derivation procedure starts with

$$\mathcal{I}nc\mathcal{A}pply[\![f_0(x \oplus y)]\!]\ \emptyset\ \{\langle f_0(x), r, \emptyset\rangle\}\ \emptyset \tag{3.25}$$

and, if it terminates, returns $\langle f_0'(x, y, r),\ D\rangle$, where D is the set of functions introduced during the derivation. We can eliminate dead functions in F (the set of functions in the original program) and D that are not reachable from f_0' in the call-graph.

The derivation procedure preserves the semantics of programs and achieves at least as fast computations, i.e., if $f_0(x) = r$, then (a) whenever $f_0(x \oplus y)$ returns a value, $f_0'(x, y, r)$ returns the same value; and (b) $f_0'(x, y, r)$ is asymptotically at least as fast as $f_0(x \oplus y)$. To see this, notice that semantics are preserved and fast computations are achieved by all the transformations in the derivation procedure — simplification by $\mathcal{S}imp$, computation of cache sets and replacement by $\mathcal{R}epl$, lifting of conditions and bindings by $\mathcal{S}ubl$, and function replacement and introduction with generalization by $\mathcal{I}nc\mathcal{A}pply$. Note that unfolding may result in computations that terminate more often than the original computations.

Transformation techniques. The transformation is a deterministic transformational procedure. It starts with $f_0(x \oplus y)$, so that f_0' is computable, and aims to improve the efficiency by replacing subcomputations whose values can be retrieved from cached result r of $f_0(x)$ by corresponding retrievals. This starting point is similar to that of partial evaluation, which starts with a trivial specialized program given by Kleene's s-m-n theorem [Kle52] and attempts improvements by symbolic reductions or similar techniques. We summarize the major transformation techniques used and emphasize how they are combined to achieve the goal.

First, context information is collected for each subcomputation and used to simplify the computation, which mimics the main techniques of generalized partial evaluation [FN88], where program states are represented symbolically and programs are specialized with the help of a theorem prover. In addition to simplification, context information has another important role in our work, i.e., it serves as keys to cached results and introduced functions for valid replacement to happen.

Second, a cache set is maintained for each unfolded application and used to incrementalize it, i.e., to replace certain subcomputations, under certain context information, by retrievals from a cached result of a previous computation. A cache set is augmented, finitely and in a disciplined way, with the help of an auxiliary specializer so that the cached result is utilized effectively under valid context information. The use of a cached result often suggests memoization [Mic68,Bir80]. However, the real power of our approach comes from the effective exploitation of a memoized value under valid context information. The approach to be proposed in Chapter 4 for increasing incrementality by caching intermediate results can be regarded as a form of smart memoization.

Third, consistent with the strict semantics of our language, we apply simplification and replacement on subcomputations in applicative order and, moreover, lift conditions and bindings out of subcomputations. This lifting technique is similar in spirit to the driving transformation by supercompilation [Tur86]. It causes relatively drastic reorganization of program structures that helps expose incrementality that is otherwise hidden.

Fourth, a global definition set is maintained and used to replace function applications, with corresponding relevant cache elements and valid context information, by applications of introduced functions. Function introduction with generalization and function replacement use the unfold/define/fold scheme [BD77] in a regulated manner so that the transformations are deterministic and the derived programs converge if the originals do. Moreover, relevant cache elements with valid context information are chosen to be passed into introduced functions, so that they can be effectively used to incrementalize the computation of corresponding function applications.

Last, after the replacements described above, we apply dead code elimination, a traditional optimization technique [MJ81,ASU86]. It is particularly useful here, since replacement changes dependencies between computations, and computations on which no other computations depend are then dead and can be eliminated.

Analysis techniques. To implement the above transformations for achieving the ambitious goal of deriving incremental programs, a number of program analysis techniques are needed.

First, time analysis [Weg75,Ros89] is used when replacing subcomputations by retrievals or replacing function applications by applications of introduced functions.[3] It is a must if we want to guarantee the efficiency of the derived programs.

Then, several analyses [JGS93] are used to assist transforming function applications. *Dependency analysis* enables us to recognize subcomputations that are possibly computed incrementally, i.e., subcomputations depending on x, and thus avoid introducing functions for function applications that depend only on y, which then helps the derivation procedure terminate. *Call-graph analysis* tells us whether a function is recursively defined and also whether an introduced function is fully defined. *Occurrence counting analysis* helps us decide whether an unfolding duplicates computations.

Finally, dead code analysis recognizes dead code to be eliminated. In particular, dead parameters of functions can be recognized with the help of dependency analysis, and dead functions can be identified with the help of call-graph analysis.

Additionally, other analysis techniques, although not mentioned in our transformations so far, would also benefit the derivation procedure. For example, type

[3]The minor use of time analysis in assisting some simplifications can be easily avoided.

analysis would be helpful for simplifying overloaded functions. Also, static analysis could provide annotations that guide the derivation and help it terminate, mimicking binding-time analysis in partial evaluation, as discussed below.

Last but not least, it should be noted that, even with these analysis techniques, the quality of a derived incremental program depends on the corresponding non-incremental program. One should not expect "genuine creativity" without discoveries and proofs of some "substantial" theorems. On another hand, with the power of our combined techniques, a very simple theorem prover can already help us derive efficient incremental programs. Examples can be found in Section 3.5.

Improving the derivation procedure. A number of optimizations can be made to the derivation procedure. An implementer would naturally realize most of them. As an example, assume the replacement guarantees

$$\text{if } \mathcal{R}epl[\![e]\!]IC = \langle e', C' \rangle, \text{ then } \mathcal{R}epl[\![e']\!]IC' = \langle e', C' \rangle,$$

then one can make the optimization:

$$\mathcal{I}nc[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]ICD = \mathcal{I}nc[\![e_3]\!]IC'D', \quad \text{if } \mathcal{I}nc[\![e_1]\!]ICD = \langle F, C', D' \rangle. \tag{3.26}$$

A relatively important improvement is with the function introduction for a function application $f(e_1, ..., e_n)$, as in the second case in Figure 3.5. While we incrementalize the unfolded application, its cache set is extended from C' to C'', but C'' is discarded after this, even if $C''\theta$ might be used in incrementalizing the rest of the unfolded application that contains $f(e_1, ..., e_n)$. To make use of C'' for this purpose, we can let $\mathcal{I}nc\mathcal{A}pply$ also return the set $C''\theta$ and merge it with the cache set of the unfolded application that contains $f(e_1, ..., e_n)$.

Termination. The derivation procedure follows function applications and introduces new functions to compute these applications incrementally. Therefore, if functions are recursively defined, the derivation procedure may not terminate due to introducing infinitely many functions following infinite unfolding. Non-termination is a traditional problem in transformational approaches, and it is well-known that there is a trade-off between termination of the transformation and efficiency of the transformed programs.

In our derivation, we introduce new functions only for function applications that depend on x, which may affect the efficiency of other function applications but makes the derivation terminate more often without impeding the discovery of incrementality. It is also clear that function replacement and the notion of generalization for function introduction help the derivation terminate in a natural way. However, our heuristic of one cache element per introduced function might impede achieving incrementality, since this element may not be *sufficient*, i.e., it may not enable all of the simplifications and replacements that are possible when using more cache elements. We could overcome this by using as many cache elements as possible when introducing a function and eliminating useless ones later. But this may cause a too complicated treatment of recursive functions and may make the derivation terminate less often. On the other hand, this is a place where separate passes of static analysis could help, imitating binding-time analysis in partial evaluation. This suggests a direction for future work.

Although in general, any attempt to limit function introductions could affect achieving incrementality for certain programs, it does not hurt to try a few good heuristics with more reasonable termination behavior. For example, we may introduce a new function at a function application only if we can effectively decide that, in incrementalizing the unfolded application, some subcomputations can be simplified. Thus, assuming we have a complete equality reasoning mechanism and a sufficient cache element when introducing a function, if the derivation procedure does not terminate, there must be simplification possible along an infinite path, and thus there must an execution of the original program that does not terminate. In other words, if the original program terminates on all inputs, then the derivation procedure terminates, and the derived program terminates on all inputs at least as fast with the right values. Note, however, that the complexity of the derivation procedure may not be bounded by the size of a given program, since it may loop on ground values. The rationale is that computations done at transformation time need not be done in the transformed programs.

Other concerns. Two other weaknesses result from unfolding as done by the derivation procedure. First, only partial correctness is preserved, i.e., a derived program may terminate more often than the original program. Second, subcomputations may be duplicated in a derived program.

Both drawbacks can be overcome by inserting **let** bindings to compute the arguments when unfolding function applications, i.e., instead of unfolding a function application to $e_f[e_1/v_1, ..., e_n/v_n]$, we unfold it to

$$\textbf{let } v_1 = e_1 \textbf{ in } ... \textbf{ let } v_n = e_n \textbf{ in } e_f$$

Then we modify the condition of unfolding **let** expressions in $\mathcal{S}imp$, namely, **let** $v = e_1$ **in** e_2 can be unfolded only if $e_2[e_1/v]$ neither duplicates non-trivial computations nor discards non-terminating computations, where the latter means either e_1 can be effectively decided to terminate or v occurs at least once on every (syntactic) execution path in e_2. As occurrence counting analysis helps decide whether an unfolding duplicates computations, it can also help decide whether an unfolding discards computations.

Similar solutions are proposed in partial evaluation [Mog88,BD91]. Note that, even without this technique, the efficiency of our derived programs is guaranteed with the help of time analysis. But in partial evaluation where no time analysis is employed, a transformed program could take exponential time while the original program takes only polynomial time [Mog88]. As a matter of fact, even with this technique, time analysis is still needed in our derivation, since we replace subcomputations by retrievals from a cache result only when we can save time by doing so. This is inherent in incremental computation and is a complication over partial evaluation.

Mechanization. With the oracle of a theorem prover, time analysis techniques, and heuristics for function introductions, the derivation can be fully automated. In practice, the derivation can be made semi-automatic when some of these oracles are only semi-automatically provided.

Although we see the derivation as certainly no more automatable than partial evaluation, it is desirable to at least use the computer as a sophisticated editor, suggesting and carrying out detailed transformations. It is also nice that the derived programs are in the same language as the original programs, and therefore they are executable and one can check solutions and try out alternatives.

3.5 Examples

To see the power and some interesting behavior of the derivation procedure, we consider incrementalizing several different sorting programs. Let *sort* be a function that takes a list of numbers x and returns the sorted list $sort(x)$. Let the change to the input of *sort* be that an extra number is added at the beginning of the list, i.e., $x' = cons(y, x)$.

3.5.1 Insertion sort

Suppose the program is an insertion sort that inserts the first element of the list into the recursively sorted list of the rest, as in Figure 3.6. To compute $sort(cons(y, x))$

$$
\begin{array}{lcl}
sort(x) & : & \text{sort a list } x \text{ using insertion sort} \\[4pt]
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else } insert(car(x), sort(cdr(x))) \\[4pt]
insert(i, x) & = & \textbf{if } null(x) \textbf{ then } cons(i, nil) \\
 & & \textbf{else if } i \leq car(x) \textbf{ then } cons(i, x) \\
 & & \quad\quad \textbf{else } cons(car(x), insert(i, cdr(x)))
\end{array}
$$

Figure 3.6: Example function definitions of insertion *sort* and *insert*

incrementally using $sort(x) = r$, all we need to do is a function introduction, followed by an unfolding, a few simplifications, a replacement, and a dead parameter elimination, and finally a use of the introduced function, as sketched below:

$$
\begin{array}{lll}
sort'(y_1, x_1, r_1) & = sort(cons(y_1, x_1)), \quad \text{with } sort(x_1) = r_1 & \\
 & & \text{function introduction} \\
 & = \textbf{if } null(cons(y_1, x_1)) \textbf{ then } nil & \\
 & \quad \textbf{else } insert(car(cons(y_1, x_1)), sort(cdr(cons(y_1, x_1)))) & \\
 & & \text{unfolding} \\
 & = insert(y_1, sort(x_1)) & \text{simplifications} \\
 & = insert(y_1, r_1) & \text{replacement} \\
sort'(y_1, r_1) & = insert(y_1, r_1) & \text{dead parameter elimination} \\
sort(cons(y, x)) & = sort'(y, r), \quad \text{for } sort(x) = r & \text{using introduced function}
\end{array}
$$

The derived incremental program simply uses *insert* to insert the newly added number into the previously sorted list.

A more formal derivation following the derivation procedure is given below. We start with

$$\mathcal{I}nc\mathcal{A}pply[\![sort(cons(y, x))]\!] \ \emptyset \ \{\langle sort(x), r, \emptyset\rangle\} \ \emptyset \tag{3.27}$$

where we introduce $sort'$ to get a tuple d_1:

$$\langle sort(cons(y_1, x_1)), sort'(y_1, x_1, r_1), C_1\rangle, \quad \text{where } C_1 = \{\langle sort(x_1), r_1, \emptyset\rangle\}$$

and we obtain a definition of $sort'$ as follows:

1. We unfold $sort(cons(y_1, x_1))$ and get an expression e_1:

$$
\begin{array}{l}
\textbf{if } null(cons(y_1, x_1)) \textbf{ then } nil \\
\textbf{else } insert(car(cons(y_1, x_1)), sort(cdr(cons(y_1, x_1))))
\end{array}
$$

2. We incrementalize e_1:

$$\mathcal{I}nc[\![e_1]\!] \ \emptyset \ C_1 D_1, \quad \text{where } D_1 = \{d_1\}$$

which first calls $\mathcal{S}ubl$ to reduce subexpressions:

$$
\begin{array}{l}
\mathcal{S}ubl[\![e_1]\!] \ \emptyset \ C_1 D_1 \\
\quad -- \text{by } (if_1),\ \mathcal{I}nc[\![null(cons(y_1, x_1))]\!] \ \emptyset \ C_1 D_1 = \langle F, C_1, D_1\rangle \\
= \mathcal{S}ubl[\![\textbf{if } F \textbf{ then } nil \textbf{ else } e_{13}]\!] \ \emptyset \ C_1 D_1, \text{ where } e_{13} \text{ denotes false branch of } e_1 \\
\quad -- \text{by } (if_n),\ \mathcal{I}nc[\![e_{13}]\!] \ \{F \leftrightarrow F\} \ C_1 D_1 = \langle insert(y_1, r_1), C_1, D_1\rangle \\
\qquad \text{since } car(cons(y_1, x_1)) = y_1,\ cdr(cons(y_1, x_1)) = x_1, \text{ and } sort(x_1) = r_1; \\
\qquad insert(y_1, r_1) \text{ does not depend on } x, \text{ so } \mathcal{I}nc\mathcal{A}pply \text{ does not transform it} \\
= \langle \textbf{if } F \textbf{ then } nil \textbf{ else } insert(y_1, r_1), C_1, D_1\rangle
\end{array}
$$

and then applies $\mathcal{S}imp$ and $\mathcal{R}epl$ to the resulting expression:

$$
\begin{array}{l}
\mathcal{R}epl[\![\mathcal{S}imp[\![\textbf{if } F \textbf{ then } nil \textbf{ else } insert(y_1, r_1)]\!] \ \emptyset^{C_1}]\!] \ \emptyset^{C_1} \ C_1 \\
= \mathcal{R}epl[\![insert(y_1, r_1)]\!] \ \emptyset^{C_1} \ C_1 \\
= insert(y_1, r_1)
\end{array}
$$

3. We eliminate dead parameters of $sort'$, defined by $sort'(y_1,x_1,r_1) = insert(y_1,r_1)$. Clearly, x_1 is dead. We obtain a final definition of $sort'$:

$$sort'(y_1, r_1) = insert(y_1, r_1) \tag{3.28}$$

It is clear that $sort'(y_1, r_1)$ is asymptotically at least as fast as $sort(cons(y_1, x_1))$ since each transformation step above guarantees this relation. Therefore, $t(sort'(y, r)) \leq t(sort(cons(y, x)))$. Thus, (3.27) returns

$$\langle sort'(y, r), \ \ \{\langle sort(cons(y_1, x_1)), \ sort'(y_1, r_1), \ \{\langle sort(x_1), r_1, \emptyset\rangle\}\rangle\}\rangle$$

where $sort'$ is defined as in (3.28) and $insert$ is defined as in the original program in Figure 3.6.

3.5.2 Selection sort

Suppose the program is a selection sort that selects the least number in the list as the first number in the sorted list and sorts the rest recursively, as shown in Figure 3.7. Again, we start by introducing $sort'(y_1, x_1, r_1)$ to compute $sort(cons(y_1, x_1))$ incrementally. But while incrementalizing the unfolded $sort(cons(y_1, x_1))$ to get a definition of $sort'$, the application $least(cons(y_1, x_1))$ is transformed recursively, which results in the lifting of some conditions and bindings, and then applications of $rest$ are transformed under these conditions and bindings. As the result of these transformations, $sort'$ compares y_1 with the first number in r_1 to decide whether y_1 should stay before r_1, and, if not, recursively considers y_1 with the rest of r_1. But this is exactly the process of inserting y_1 into r_1 at the right place. Thus, to a certain degree, the derivation procedure *discovered* the insertion process from the selection sort via a series of transformations.

$$
\begin{array}{lcl}
sort(x) & : & \text{sort a list } x \text{ using selection sort} \\[4pt]
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else let } k = least(x) \textbf{ in} \\
 & & \qquad cons(k, sort(rest(x, k))) \\
least(x) & = & \textbf{if } null(cdr(x)) \textbf{ then } car(x) \\
 & & \textbf{else let } s = least(cdr(x)) \textbf{ in} \\
 & & \qquad \textbf{if } car(x) \leq s \textbf{ then } car(x) \textbf{ else } s \\
rest(x, k) & = & \textbf{if } k = car(x) \textbf{ then } cdr(x) \\
 & & \textbf{else } cons(car(x), rest(cdr(x), k))
\end{array}
$$

Figure 3.7: Example function definitions of selection *sort*, *least*, and *rest*

Since the derivation procedure is more complicated, an informal but complete derivation is given below. As just mentioned, to compute $sort(cons(y, x))$ incrementally using $sort(x) = r$, we start by introducing $sort'(y_1, x_1, r_1)$ for $sort(cons(y_1, x_1))$ with $sort(x_1) = r_1$.

1. Unfold $sort(cons(y_1, x_1))$:

$$\begin{array}{l} \textbf{if } null(cons(y_1,x_1)) \textbf{ then } nil \\ \textbf{else let } k = least(cons(y_1,x_1)) \textbf{ in } cons(k, sort(rest(cons(y_1,x_1),k))) \end{array} \tag{3.29}$$

2. Simplify condition $null(cons(y_1, x_1))$ in (3.29) to false and simplify (3.29) to:

$$\textbf{let } k = least(cons(y_1,x_1)) \textbf{ in } cons(k, sort(rest(cons(y_1,x_1),k))) \tag{3.30}$$

3. Consider application $least(cons(y_1, x_1))$ in (3.30) and introduce $least'(y_2, x_2, r_2)$ for $least(cons(y_2, x_2))$ with $sort(x_2) = r_2$.

   3.1. Unfold $least(cons(y_2, x_2))$:

$$\begin{array}{l} \textbf{if } null(cdr(cons(y_2,x_2))) \textbf{ then } car(cons(y_2,x_2)) \\ \textbf{else let } s = least(cdr(cons(y_2,x_2))) \textbf{ in} \\ \qquad \textbf{if } car(cons(y_2,x_2)) \le s \textbf{ then } car(cons(y_2,x_2)) \textbf{ else } s \end{array} \tag{3.31}$$

   3.2. Simplify the condition $null(cdr(cons(y_2, x_2)))$ to $null(x_2)$, and then replace $null(x_2)$ by $null(r_2)$ since when $null(x_2)$ is true (false)

$$null(sort(x_2)) \text{ is specialized to true (false).}$$

   In the true branch, simplify $car(cons(y_2, x_2))$ to y_2. In the false branch, simplify $least(cdr(cons(y_2, x_2)))$ to $least(x_2)$, and replace $least(x_2)$ by $car(r_2)$ since when $null(r_2)$ is false

$$sort(x_2) \text{ is specialized to } \textbf{let } k_2 = least(x_2) \textbf{ in } cons(k_2, sort(rest(x_2, k_2))),$$

   and then, in the body of the **let** expression, simplify $car(cons(y_2, x_2))$ to y_2. Thus, (3.31) becomes

$$\textbf{if } null(r_2) \textbf{ then } y_2 \textbf{ else let } s = car(r_2) \textbf{ in if } y_2 \le s \textbf{ then } y_2 \textbf{ else } s \tag{3.32}$$

   3.3. For $least'$ defined by $least'(y_2, x_2, r_2) =$ (3.32), eliminate dead parameter x_2.

   Replace $least(cons(y_1, x_1))$ by $least'(y_1, r_1)$, and unfold $least'(y_1, r_1)$ since $least'$ is not recursively defined and unfolding does not duplicate non-trivial computations. Thus, (3.30) becomes

$$\begin{array}{l} \textbf{let } k = (\textbf{if } null(r_1) \textbf{ then } y_1 \textbf{ else let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } y_1 \textbf{ else } s) \textbf{ in} \\ cons(k, sort(rest(cons(y_1,x_1),k))) \end{array} \tag{3.33}$$

4. Lift the first condition $null(r_1)$ out of the top-level **let**. Thus, (3.33) becomes

$$\begin{array}{l} \textbf{if } null(r_1) \textbf{ then let } k = y_1 \textbf{ in } cons(k, sort(rest(cons(y_1,x_1),k))) \\ \textbf{else let } k = (\textbf{let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } y_1 \textbf{ else } s) \textbf{ in} \\ \qquad cons(k, sort(rest(cons(y_1,x_1),k))) \end{array} \tag{3.34}$$

5. In the true branch of (3.34), simplify x_1 to nil. No functions are introduced for the applications of $rest$ and $sort$ since they do not depend on x. Then unfold the **let**:
$$cons(y_1, sort(rest(cons(y_1, nil), y_1))) \tag{3.35}$$
In the false branch of (3.34), lift the binding $s = car(r_1)$ out of the first **let**:
$$\begin{array}{l}\textbf{let } s = car(r_1) \textbf{ in let } k = (\textbf{if } y_1 \le s \textbf{ then } y_1 \textbf{ else } s) \textbf{ in} \\ \qquad\qquad\qquad\qquad cons(k, sort(rest(cons(y_1, x_1), k)))\end{array}$$
and then lift the condition $y_1 \le s$ out of the second **let**:
$$\begin{array}{l}\textbf{let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then let } k = y_1 \textbf{ in } cons(k, sort(rest(cons(y_1, x_1), k))) \\ \qquad\qquad\qquad\qquad \textbf{else let } k = s \textbf{ in } cons(k, sort(rest(cons(y_1, x_1), k)))\end{array} \tag{3.36}$$

6. In the true branch of (3.36), first consider $rest(cons(y_1, x_1), k)$, and introduce $rest'(y_3, x_3, k_3, r_3)$ for $rest(cons(y_3, x_3), k_3)$ with $sort(x_3) = r_3$ and also $k_3 \leftrightarrow y_3$, *etc.*

   6.1 Unfold $rest(cons(y_3, x_3), k_3)$:
$$\begin{array}{l}\textbf{if } k_3 = car(cons(y_3, x_3)) \textbf{ then } cdr(cons(y_3, x_3)) \\ \textbf{else } cons(car(cons(y_3, x_3)), rest(cdr(cons(y_3, x_3)), k_3))\end{array} \tag{3.37}$$

   6.2 Simplify the condition $k_3 = car(cons(y_3, x_3))$ to $k_3 = y_3$, and further simplify it to true, and thus (3.37) is to be simplified to the true branch $cdr(cons(y_3, x_3))$, which is then simplified to x_3.

   6.3 For $rest'$ defined by $rest'(y_3, x_3, k_3, r_3) = x_3$, eliminate dead parameters y_3, k_3, and r_3.

   Replace $rest(cons(y_1, x_1), k)$ by $rest'(x_1)$, and unfold $rest'(x_1)$ to x_1. Then, replace $sort(x_1)$ by r_1. Finally, unfold the **let**. Thus, the true branch of (3.36) becomes
$$cons(y_1, r_1) \tag{3.38}$$

7. In the false branch of (3.36), first consider $rest(cons(y_1, x_1), k)$, and introduce $rest'_1(y_4, x_4, k_4, r_4)$ for $rest(cons(y_4, x_4), k_4)$ with $sort(x_4) = r_4$ and also $y_4 \le s \leftrightarrow F, k_4 \leftrightarrow s$, *etc.*

   7.1 Unfold $rest(cons(y_4, x_4), k_4)$:
$$\begin{array}{l}\textbf{if } k_4 = car(cons(y_4, x_4)) \textbf{ then } cdr(cons(y_4, x_4)) \\ \textbf{else } cons(car(cons(y_4, x_4)), rest(cdr(cons(y_4, x_4)), k_4))\end{array} \tag{3.39}$$

   7.2 Simplify the condition $k_4 = car(cons(y_4, x_4))$ to $k_4 = y_4$, and further simplify it to false, and thus (3.39) is to be simplified to the false branch, which is then simplified to $cons(y_4, rest(x_4, k_4))$.

7.3 For $rest_1'$ defined by $rest_1'(y_4, x_4, k_4, r_4) = cons(y_4, rest(x_4, k_4))$, eliminate dead parameter r_4.

Replace $rest(cons(y_1,x_1),k)$ by $rest_1'(y_1,x_1,k)$ and unfold to $cons(y_1,rest(x_1,k))$. Then, replace $sort(cons(y_1, rest(x_1,k)))$ with $sort'(y_1, rest(x_1,k), cdr(r_1))$, since when $null(r_1)$ is false

$$sort(x_1) \text{ is specialized to } \quad \textbf{let } k_1 = least(x_1) \textbf{ in } cons(k_1, sort(rest(x_1, k_1))),$$

which implies $sort(rest(x_1, k)) = cdr(r_1)$ for $k = s = car(r_1) = k_1$. Finally, unfold the **let**. Thus, the false branch of (3.36) becomes

$$cons(s, sort'(y_1, rest(x_1, s), cdr(r_1))) \tag{3.40}$$

8. Putting (3.34) (3.35) (3.36) (3.38) (3.40) together, $sort'$ is defined by

$$\begin{aligned} sort'(y_1,x_1,r_1) = \textbf{if } & null(r_1) \textbf{ then } cons(y_1, sort(rest(cons(y_1, nil), y_1))) \\ & \textbf{else let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } cons(y, r_1) \\ & \qquad\qquad\qquad\qquad \textbf{else } cons(s, sort'(y_1,rest(x_1,s), cdr(r_1))) \end{aligned}$$

Eliminating dead parameter x_1, we obtain a final definition of $sort'$:

$$\begin{aligned} sort'(y_1, r_1) = \textbf{if } & null(r_1) \textbf{ then } cons(y_1, sort(rest(cons(y_1, nil), y_1))) \\ & \textbf{else let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } cons(y_1, r_1) \\ & \qquad\qquad\qquad\qquad \textbf{else } cons(s, sort'(y_1, cdr(r_1))) \end{aligned} \tag{3.41}$$

It is easy to see that, in the true branch, $sort(rest(cons(y_1, nil), y_1))$ returns nil in constant time given any number y_1; in the false branch, the **let** expression could be unfolded. Thus, $sort'$ exactly performs an insertion as the *insert* of Figure 3.6.

At the end, we have $sort(cons(y, x)) = sort'(y, r)$ for $sort(x) = r$, where $sort'$ is defined as in (3.41).

3.5.3 Merge sort

Suppose the program is a merge sort that divides the list into two sublists, recursively sorts the two sublists, and then merges the two sorted sublists, as in Figure 3.8. An insertion can be easily obtained if we are given the property that sorting the new list equals merging the new number into the previously sorted list.

$$\begin{array}{lll} sort'(y_1, x_1, r_1) & = sort(cons(y_1, x_1)), \quad \text{with } sort(x_1) = r_1 & \\ & & \text{function introduction} \\ & = merge(cons(y_1, nil), sort(x_1)) & \text{property of } merge \text{ and } sort \\ & = merge(cons(y_1, nil), r_1) & \text{replacement} \\ sort'(y_1, r_1) & = merge(cons(y_1, nil), r_1) & \text{dead parameter elimination} \\ sort(cons(y, x)) & = sort'(y, r), \quad \text{for } sort(x) = r & \text{using introduced function} \end{array}$$

$$
\begin{aligned}
&\mathit{sort}(x) && : \text{ sort a list } x \text{ using merge sort} \\
&\mathit{sort}(x) && = \textbf{if } \mathit{null}(x) \textbf{ then } \mathit{nil} \\
& && \quad \textbf{else if } \mathit{null}(\mathit{cdr}(x)) \textbf{ then } \mathit{cons}(\mathit{car}(x), \mathit{nil}) \\
& && \quad \textbf{else } \mathit{merge}(\mathit{sort}(\mathit{odd}(x)), \mathit{sort}(\mathit{even}(x))) \\
&\mathit{odd}(x) && = \textbf{if } \mathit{null}(x) \textbf{ then } \mathit{nil} \\
& && \quad \textbf{else } \mathit{cons}(\mathit{car}(x), \mathit{even}(\mathit{cdr}(x))) \\
&\mathit{even}(x) && = \textbf{if } \mathit{null}(x) \textbf{ then } \mathit{nil} \\
& && \quad \textbf{else } \mathit{odd}(\mathit{cdr}(x)) \\
&\mathit{merge}(x,y) && = \textbf{if } \mathit{null}(x) \textbf{ then } y \\
& && \quad \textbf{else if } \mathit{null}(y) \textbf{ then } x \\
& && \quad \textbf{else if } \mathit{car}(x) \le \mathit{car}(y) \textbf{ then} \\
& && \qquad \mathit{cons}(\mathit{car}(x), \mathit{merge}(\mathit{cdr}(x), y)) \\
& && \quad \textbf{else } \mathit{cons}(\mathit{car}(y), \mathit{merge}(x, \mathit{cdr}(y)))
\end{aligned}
$$

Figure 3.8: Example function definitions of merge *sort*, *odd*, *even*, and *merge*

The derived program $sort'$ basically performs an insertion with a constant factor overhead over the *insert* of Figure 3.6. The required property that relates *merge* and *sort* can be proved by a straightforward induction based on the associativity and commutativity of *merge*. However, if the above property is not given, then no incremental program can be derived using the derivation procedure. But what is interesting is the following.

Suppose we cache certain intermediate results during the merge sort, namely, the recursively sorted sublists. Then, following the derivation procedure, we can easily obtain an *incremental merge sort* that sorts the new list by recursively merging the new number with the appropriate intermediate results and, at the same time, maintains the corresponding intermediate results. Each merge takes two sublists of roughly equal lengths and returns a sorted list of doubled length, where the lengths of the lists being processed go from one to the length of x. The derivation will be shown in the next chapter, where systematic techniques for caching intermediate results are described.

Both insertion sort and selection sort take $O(n^2)$ time, where n is the length of the input list, and merge sort takes $O(n \log n)$ time. Insertion takes only $O(n)$ time; but it uses $O(n)$ space to store the previously sorted list. Incremental merge sort also takes $O(n)$ time; but it uses $O(n \log n)$ space to store intermediate results.

The derivation of the incremental merge sort suggests that the approach of exploiting cached values for incrementality is powerful: the power of caching sometimes obviates the reliance on a theorem prover for proving certain properties. We can also view it as trading space for theorem proving ability.

3.6 Related work

Related work in program analysis and transformation techniques has been summarized in Section 3.4. This section takes a closer look at related work in incremental computation, which is introduced in Chapter 2 and partitioned into three classes.

First of all, given particular problems with certain input changes, can the approach be used to derive as efficient incremental programs as those in the first class? The general answer is positive, but with three caveats. First, the particular problem needs to be coded naturally in the language for which our approach is presented. Second, the quality of a derived incremental program depends on the way the non-incremental program is coded, as seen in the sort examples in Section 3.5. Third, intermediate results and auxiliary information are needed for many incremental problems but may be difficult to determine. In this case, we can use the techniques in Chapters 4 and 5, or at least use the derivation procedure on programs that are extended to compute manually determined intermediate results and auxiliary information, and derive programs that incrementally maintain these intermediate results and auxiliary information.

Since the transformational approach is related to partial evaluation in some aspects, it is worthwhile to compare it with the work by Sundaresh and Hudak [SH91, Sun91] in the second class. The common aspect is that both works aim at obtaining incremental computation by transforming non-incremental programs. However, the two approaches follow different lines. Their work mostly *uses* partial evaluation, with extra efforts on partitioning program inputs and combining residual programs. Our method combines a series of analysis and transformation techniques that "parallel" those used in (generalized) partial evaluation, but with the goal of incrementalization in addition to specialization, and therefore employs overall more extensive and more complicated techniques. We believe a major limitation of the Sundaresh-Hudak framework is that it can only handle input changes according to a pre-given input partition, which is partly implied as a work in the second class.

Our work is closest in spirit to the finite differencing techniques of the third class. The name "finite differencing" was originally given by Paige and Koenig [PK82]. Their work generalizes Cocke and Kennedy's strength reduction [CK77] and provides a convenient framework for implementing a host of transformations including Earley's "iterator inversion" [Ear76]. They develop a set of rules for differentiating set-theoretic expressions and combine these rules using a chain rule to derive inexpensive programs with incremental loop bodies. Such techniques are indispensable as part of an optimizing compiler for languages like SETL or APL [PH87,CP89]. The APTS program transformation system [Pai90,Pai94] has been developed for such purposes. Our technique differs from theirs in that it applies to programs written in a standard language like Lisp. In general, such programs are written at a lower abstraction level so that a fixed set of rules for differentiating primitive operations involving complex objects like sets is not sufficient. The technique we propose can be regarded as a principle and a systematic approach, through which incrementality can be *discovered* in existing programs written in standard languages.

Smith's work in KIDS [Smi90,Smi91] is closely related to this work. KIDS is a semi-automatic program development system that aims to derive efficient programs from high-level specifications [SL90], as is APTS. Its version of finite differencing was developed for the optimization of its derived functional programs and has two basic operations: abstraction and simplification. Abstraction of a function f adds an extra cache parameter to f. Simplification simplifies the definition of f given the added cache parameter. However, as to *how* the cache parameter should be used in the simplification to provide incrementality, KIDS provides only the observation that distributive laws can often be applied. The Munich CIP project [BMPP89,Par90] has a strategy for finite differencing that captures similar ideas. It first "defines by a suitable embedding a function f'", and then "derives a recursive version of f' using generalized unfold/fold strategy", but it provides no special techniques for discovering incrementality. We believe that both works provide only general strategies with no precise procedure to follow and therefore are less automatable than ours.

Chapter 4

Caching intermediate results

The value of $f_0(x \oplus y)$ may often be computed faster by using not only the return value of $f_0(x)$, as discussed in Chapter 3, but also the values of some subcomputations performed in $f_0(x)$ that can not be retrieved from the return value of $f_0(x)$. These values are called *intermediate results* useful for computing f_0 incrementally under $\oplus$.

Examples where intermediate results are needed for incremental computation include incremental parsing [JG82] and incremental attribute evaluation [RTD83, Yeh83]. An incremental parser may cache, in addition to the derived parse tree, the $LR(0)$ state corresponding to each shift and reduction. An attribute evaluator may only return some designated synthesized attribute of the root [Kat84], but the corresponding incremental attribute evaluator may cache the whole attributed tree.

This chapter describes a three-stage method, called *cache-and-prune*, for statically transforming a program f_0 to cache intermediate results useful for the incremental computation under $\oplus$. The basic idea is to (I) extend f_0 to a program $\bar{f}_0$ that returns all intermediate results, (II) incrementalize $\bar{f}_0$ under $\oplus$ to obtain an incremental version $\bar{f}_0'$, and (III) using the dependencies in $\bar{f}_0'$, prune $\bar{f}_0$ to a program $\hat{f}_0$ that returns only the useful intermediate results and prune $\bar{f}_0'$ to a program $\hat{f}_0'$ that incrementally maintains the useful intermediate results.

Note that every program computes by fixed point iteration and, for efficiency, each iteration should be computed incrementally based on the previous iteration. The approach for caching intermediate results for incremental computation can be applied straightforwardly and systematically for improving program efficiency via caching. As an example, we will see that the classical linear-time Fibonacci function falls directly out of the approach. Previous work on caching relies on a fixed set of rules [ACK81,PK82], applies only to programs with certain properties or schemas [Bir80,Coh83,Pet84,Pet87], requires program annotations [KS86,SH91,Hoo92], *etc.*

Defining the problem. We use an asymptotic time model as in Chapter 3 and use $t(f(v_1,..,v_n))$ to denote the asymptotic time of computing $f(v_1,...,v_n)$. Since only asymptotic time is of concern, it is sufficient to consider only the values of function applications as candidate intermediate results to be cached. Of course, caching intermediate results takes extra space, which reflects the well-known principle of trading space for speed. This work assumes that there is unlimited space to be used for achieving the least asymptotic time possible. Pruning saves time as well as space for computing and maintaining intermediate results that are not useful for incremental computation.

Given a program f_0 and an input change operation $\oplus$, we aim to transform f_0 to cache intermediate results that are useful for computing f_0 incrementally under $\oplus$. For example, consider function *foo* of Figure 4.1 and input change operation $x' = x \oplus y = x+1$. Using the approach in Chapter 3 directly, the function foo' of Figure 4.2 can be derived. But computing $foo'(x,r)$ is not much faster than computing $foo(x+1)$ from scratch. This is an example where we can compute the value of $f_0(x \oplus y)$ faster by caching and using, in addition to the value of $f_0(x)$, the intermediate results computed in $f_0(x)$. For example, the value of $foo(x-1)+foo(x-2)$, which could be used in computing $foo'(x,r)$, is also computed by $foo(x)$ but can not be retrieved from r. We can cache this value and use it in computing the value of $foo(x+1)$ faster.

$foo(x):$ sum three preceding "foo" numbers of x

$$
\begin{aligned}
foo(x) &= \textbf{if } x \le 2 \textbf{ then } 1 \\
&\quad\ \textbf{else } boo(x) + foo(x-3) \\
boo(x) &= foo(x-1) + foo(x-2)
\end{aligned}
$$

Figure 4.1: Example function definitions of *foo* and *boo*

Mechanical transformations for caching intermediate results need consistent notations. We use <> to denote a tuple constructed by the transformation that bundles intermediate results with the original return value, with $1st$ returning the first element, which is always the original value, and rst returning a tuple of the remaining elements, which are the corresponding intermediate results. We use nth to get the nth element of such a tuple, and we use an infix operation @ to concatenate two such tuples.

For typographical convenience, $\bar{f}_0$ shall always denote the extended function that returns all intermediate results of f_0, $\bar{r}$ the cached result of $\bar{f}_0(x)$, and $\bar{f}_0'$ an incremental version of $\bar{f}_0$ under $\oplus$. Similarly, $\hat{f}_0$ shall denote the pruned function that returns only the intermediate results of f_0 useful for incremental computation, $\hat{r}$ the cached result of $\hat{f}_0(x)$, and $\hat{f}_0'$ a function that incrementally maintains the useful intermediate results.

We use function *foo* of Figure 4.1 and input change operation $x' = x \oplus y = x+1$ as a running example. At the end, we obtain the functions $\widehat{foo}$, $\widehat{boo}$, and $\widehat{foo}'$ shown in Figure 4.2. In particular, $\widehat{foo}'$ computes incrementally using only $O(1)$ time.

4.1 Overview of the approach

Since $f_0(x \oplus y)$ can be computed incrementally using only values that are available like the value of $f_0(x)$, we want to extend f_0 so that intermediate values computed

If $foo(x) = r$, then $foo'(x, r) = foo(x + 1)$.
For x of length n, $foo'(x, r)$ takes time $O(3^n)$;
$foo(x+1)$ takes time $O(3^n)$.

$foo(x) = 1st(\widehat{foo}(x))$.
For x of length n, $\widehat{foo}(x)$ takes time $O(3^n)$;
$foo(x)$ takes time $O(3^n)$.

If $\widehat{foo}(x) = \hat{r}$, then $\widehat{foo}'(x, \hat{r}) = \widehat{foo}(x + 1)$.
For x of length n, $\widehat{foo}'(x, \hat{r})$ takes time $O(1)$;
$\widehat{foo}(x+1)$ takes time $O(3^n)$.

$$
\begin{aligned}
foo'(x, r) &= \textbf{if } x \leq 1 \textbf{ then } 1 \\
& \textbf{ else if } x = 2 \textbf{ then } 3 \\
& \textbf{ else } r + foo(x-1) + foo(x-2) \\
\widehat{foo}(x) &= \textbf{if } x \leq 2 \textbf{ then } <1> \\
& \textbf{ else let } v_1 = \widehat{boo}(x) \textbf{ in} \\
& \qquad <1st(v_1) + foo(x-3),\ v_1> \\
\widehat{boo}(x) &= \textbf{let } v_1 = foo(x-1) \textbf{ in} \\
& \quad <v_1 + foo(x-2),\ <v_1>> \\
\widehat{foo}'(x, \hat{r}) &= \textbf{if } x \leq 1 \textbf{ then } <1> \\
& \textbf{ else if } x = 2 \textbf{ then } <3, <2, <1>>> \\
& \textbf{ else } <1st(\hat{r}) + 1st(2nd(\hat{r})), \\
& \qquad <1st(\hat{r}) + 1st(2nd(2nd(\hat{r}))), \\
& \qquad\quad <1st(\hat{r})>>>
\end{aligned}
$$

Figure 4.2: Resulting function definitions of foo', $\widehat{foo}$, $\widehat{boo}$, and $\widehat{foo}'$

in $f_0(x)$ that can also be used in computing $f_0(x \oplus y)$ are returned as well.

Selective caching method. A relatively straightforward method is mimicking the derivation approach in Chapter 3 to identify intermediate results of $f_0(x)$ that can be used in computing $f_0(x \oplus y)$ *but* whose values can not be retrieved from the cached result r of $f_0(x)$ and transforming $f_0(x)$ to cache and return these values. Such a *selective caching* method is as heavy-weight as the derivation approach in Chapter 3.

Moreover, suppose $g(x)$ captures these intermediate results of $f_0(x)$. Let $\hat{f}_0^1 = \langle f_0, g \rangle$. To compute $f_0(x \oplus y)$ incrementally, we need to maintain the value of $g(x \oplus y)$ to support incremental computation after further input changes. Thus, $\hat{f}_0^1(x \oplus y)$ needs to be computed incrementally using the cached result $\hat{r}^1$ of $\hat{f}_0^1(x)$. However, computing $g(x \oplus y)$ incrementally may introduce the need to cache other intermediate results of $f_0(x)$, i.e., there may be other intermediate results of $f_0(x)$, and thus of $\hat{f}_0^1(x)$, that can be used to compute $g(x \oplus y)$, and thus $\hat{f}_0^1(x \oplus y)$, but can not be retrieved even from $\hat{r}^1$. To capture these other intermediate results, the selective caching method needs to be applied again, to the extended program $\hat{f}_0^1$ and input change operation $\oplus$.

This process may repeat until we obtain a program $\hat{f}_0^i$ such that all intermediate results of $\hat{f}_0^i(x)$ that can be used in computing $\hat{f}_0^i(x \oplus y)$ can be retrieved from the cached result $\hat{r}^i$ of $\hat{f}_0^i(x)$. Intuitively, this always terminates since there exists an upper bound of such $\hat{f}_0^i$'s, namely, a program that returns all intermediate results of f_0. However, the number of repetitions depends at least on f_0 and $\oplus$. Moreover, each repetition is heavy-weight. Thus, we propose instead a simple three-stage method called *cache-and-prune*.

Cache-and-prune method. The cache-and-prune method consists of three stages. Whereas a light-weight dependency analysis may be iterated a number of times in Stage III, the heavy-weight derivation in Chapter 3 is performed only once in stage II.

Stage I constructs a program $\bar{f}_0$, an extended version of f_0, such that $\bar{f}_0(x)$ returns the values of all function calls made in computing $f_0(x)$. Basically, $\bar{f}_0(x)$ returns a tuple containing both the intermediate results and the value of $f_0(x)$, such that

$$1st(\bar{f}_0(x)) = f_0(x) \;\text{ and }\; t(\bar{f}_0(x)) \leq t(f_0(x)). \tag{4.1}$$

Stage II derives a function $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$, using the approach in Chapter 3, such that if $\bar{f}_0(x) = \bar{r}$, then we have if $\bar{f}_0(x \oplus y) = \bar{r}'$, then

$$\bar{f}_0'(x, y, \bar{r}) = \bar{r}' \;\text{ and }\; t(\bar{f}_0'(x, y, \bar{r})) \leq t(\bar{f}_0(x \oplus y)) \tag{4.2}$$

and thus, together with (4.1), we have

$$1st(\bar{f}_0'(x, y, \bar{r})) = 1st(\bar{f}_0(x \oplus y)) = f_0(x \oplus y). \tag{4.3}$$

Stage III generates a function $\hat{f}_0$, a pruned version of $\bar{f}_0$, such that $\hat{f}_0(x)$ returns $\Pi(\bar{r})$, where $\bar{r}$ is the return value of $\bar{f}_0(x)$, and $\Pi(\bar{r})$ projects out the first and other components of $\bar{r}$ on which $1st(\bar{f}_0'(x, y, \bar{r}))$ *transitively* depends. The dependency is transitive in the sense that if $1st(\bar{f}_0'(x, y, \bar{r}))$ depends on $\Pi_1(\bar{r})$, and $\Pi_1(\bar{f}_0'(x, y, \bar{r}))$ depends on $\Pi_2(\bar{r})$, then $1st(\bar{f}_0'(x, y, \bar{r}))$ depends also on $\Pi_2(\bar{r})$. This transitivity is caused by the need to *maintain* intermediate results corresponding to those that are *used* for computing $1st(\bar{f}_0'(x, y, \bar{r}))$. In other words, this stage eliminates those intermediate results cached in $\bar{r}$ that are not transitively needed in incrementally computing $1st(\bar{f}_0'(x, y, \bar{r}))$, the value of $f_0(x \oplus y)$.[1] In particular, if $f_0(x) = r$, then

$$1st(\hat{f}_0(x)) = r \;\text{ and }\; t(\hat{f}_0(x)) \leq t(f_0(x)). \tag{4.4}$$

Additionally, we obtain a function $\hat{f}_0'$, a pruned version of $\bar{f}_0'$, such that if $\bar{f}_0'(x, y, \bar{r})$ returns $\bar{r}'$, then $\hat{f}_0'(x, y, \hat{r})$, where $\hat{r}$ is $\Pi(\bar{r})$, returns $\Pi(\bar{r}')$. This pruning is possible because $\Pi(\bar{r}')$ depends only on $\Pi(\bar{r})$, which can be easily shown using the transitivity above. With the relationship between $\hat{f}_0$ and $\bar{f}_0$, together with (4.1) and (4.2), we can prove that if $f_0(x) = r$, then we have if $\hat{f}_0(x) = \hat{r}$ and $f_0(x \oplus y) = r'$, then

$$\hat{f}_0'(x, y, \hat{r}) = \hat{f}_0(x \oplus y) \;\text{ and }\; t(\hat{f}_0'(x, y, \hat{r})) \leq t(f_0(x \oplus y)) \tag{4.5}$$

and thus, together with (4.4), we have

$$1st(\hat{f}_0'(x, y, \hat{r})) = 1st(\hat{f}_0(x \oplus y)) = r'. \tag{4.6}$$

Thus, $\hat{f}_0'(x, y, \hat{r})$ incrementally computes the desired output and the corresponding intermediate results and is asymptotically at least as fast as computing the desired

[1] Note that this is different from the partial dead code elimination in [KRS94], where partial dead code refers to code that is dead on some but not all computation paths.

output from scratch. Therefore, we do not have to conduct a derivation on $\hat{f}_0$ and $\oplus$ to obtain such an incremental function.

At the end, putting (4.4), (4.5), and (4.6) together, we have if $f_0(x) = r$, then

$$1st(\hat{f}_0(x)) = r \quad \text{and} \quad t(\hat{f}_0(x)) \leq t(f_0(x)) \tag{4.7}$$

and if $f_0(x \oplus y) = r'$ and $\hat{f}_0(x) = \hat{r}$, then

$$1st(\hat{f}_0'(x, y, \hat{r})) = r', \quad \hat{f}_0'(x, y, \hat{r}) = \hat{f}_0(x \oplus y), \quad \text{and} \quad t(\hat{f}_0'(x, y, \hat{r})) \leq t(f_0(x \oplus y)). \tag{4.8}$$

i.e, the functions $\hat{f}_0$ and $\hat{f}_0'$ preserve the semantics and compute asymptotically at least as fast. Note, however, that $\hat{f}_0(x)$ may terminate more often than $f_0(x)$ and $\hat{f}_0'(x, y, \hat{r})$ may terminate more often than $f_0(x \oplus y)$ due to the transformations used in Stages II and III.

The three stages are described in Sections 4.2, 4.3, and 4.4, respectively. Section 4.5 discusses the program analysis and transformation techniques used and the time and space consumption. Section 4.6 gives some examples, including the classical Fibonacci function. Section 4.7 presents a comprehensive comparison with related work in caching.

4.2 Stage I: Caching all intermediate results

Stage I transforms program f_0 to embed all intermediate results in the return value. It consists of a straightforward extension transformation and administrative simplifications. Optimizations to this process are also made.

4.2.1 Extension

We first perform a local, structure-preserving transformation called *extension*. For each function definition $f(v_1, ..., v_n) = e$, we construct a function definition

$$\bar{f}(v_1, ..., v_n) = \mathcal{E}xt[\![e]\!] \tag{4.9}$$

where $\mathcal{E}xt[\![e]\!]$ extends an expression e to return the values of all function calls made in computing e, i.e., it considers subexpressions of e in applicative and left-to-right order, introduces bindings that name the results of function calls, builds up tuples of these values together with the values of the original subexpressions, and passes these values from subcomputations to enclosing computations.

The definition of $\mathcal{E}xt$ is given in Figure 4.3. We assume that each introduced binding uses a fresh variable name. For transforming a conditional expression, the transformation $\mathcal{P}ad[\![e]\!]$ generates a tuple of $\_$'s of length equal to the number of function applications in e, where $\_$ is a dummy constant that just occupies a spot. The lengths of the tuples generated by $\mathcal{P}ad[\![e_2]\!]$ and $\mathcal{P}ad[\![e_3]\!]$ can easily be determined statically. Actually, they are just the lengths of $rst(v_2)$ and $rst(v_3)$, respectively. This mechanism assures that the extended function returns a uniform tuple no matter what

$$
\begin{array}{lcl}
\mathcal{E}xt[\![v]\!] & = & < v > \\
\mathcal{E}xt[\![g(e_1,...,e_n)]\!] \quad \text{where } g \text{ is } c \text{ or } p & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\
 & & < g(1st(v_1),...,1st(v_n)) > @rst(v_1)@...@rst(v_n) \\
\mathcal{E}xt[\![f(e_1,...,e_n)]\!] & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\
 & & \textbf{let } v = \bar{f}(1st(v_1),...,1st(v_n)) \textbf{ in} \\
 & & < 1st(v) > @rst(v_1)@...@rst(v_n)@ < v > \\
\mathcal{E}xt[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in} \\
 & & \textbf{if } 1st(v_1) \textbf{ then let } v_2 = \mathcal{E}xt[\![e_2]\!] \textbf{ in} \\
 & & \qquad\qquad < 1st(v_2) > @rst(v_1)@rst(v_2)@\mathcal{P}ad[\![e_3]\!] \\
 & & \qquad \textbf{else let } v_3 = \mathcal{E}xt[\![e_3]\!] \textbf{ in} \\
 & & \qquad\qquad < 1st(v_3) > @rst(v_1)@\mathcal{P}ad[\![e_2]\!]@rst(v_3) \\
\mathcal{E}xt[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!] & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in} \\
 & & \textbf{let } v = 1st(v_1) \textbf{ in let } v_2 = \mathcal{E}xt[\![e_2]\!] \textbf{ in} \\
 & & \qquad\qquad < 1st(v_2) > @rst(v_1)@rst(v_2)
\end{array}
$$

Figure 4.3: Definition of $\mathcal{E}xt$

the value of the Boolean expression is, which makes the pruning stage simpler, since we do not have to consider pruning differently under different conditions.

$\bar{f}(v_1,...,v_n)$ and $f(v_1,...,v_n)$ perform essentially the same computation, and thus take the same asymptotic time. In particular, they have the same termination behavior, and, if they terminate,

$$1st(\bar{f}(v_1,...,v_n)) = f(v_1,...,v_n). \tag{4.10}$$

The result of this transformation is a set of extended function definitions that straightforwardly embed the values of all function calls in the return values. For functions *foo* and *boo* of Figure 4.1, after the extension transformation, we obtain the functions $\overline{foo}_1$ and $\overline{boo}_1$ as follows:

$$
\begin{array}{lll}
\overline{foo}_1(x) = & \textbf{let } v_1 = \textbf{let } v_{11} = <x> \textbf{ in let } v_{12} = <2> \textbf{ in} \\
& \qquad\qquad < 1st(v_{11}) \leq 1st(v_{12}) > @rst(v_{11})@rst(v_{12}) \quad \textbf{in} \\
& \textbf{if } 1st(v_1) \textbf{ then let } v_2 = <1> \textbf{ in} \\
& \qquad\qquad\qquad < 1st(v_2) > @rst(v_1)@rst(v_2)@ < \_,\_ > \\
& \qquad\quad \textbf{else let } v_3 = \textbf{let } v_{31} = \textbf{let } v_{311} = <x> \textbf{ in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } u_1 = \overline{boo}_1(1st(v_{311})) \textbf{ in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad < 1st(u_1), u_1 > @rst(v_{311}) \quad \textbf{in} \\
& \qquad\qquad\qquad\qquad\quad \textbf{let } v_{32} = \textbf{let } v_{321} = \textbf{let } v_{3211} = <x> \textbf{ in let } v_{3212} = <3> \textbf{ in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad <1st(v_{3211}) - 1st(v_{3212})> @ \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\; rst(v_{3211})@rst(v_{3212}) \quad \textbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } u_2 = \overline{foo}_1(1st(v_{321})) \textbf{ in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad < 1st(u_2), u_2 > @rst(v_{321}) \quad \textbf{in} \\
& \qquad\qquad\qquad\qquad\quad < 1st(v_{31}) + 1st(v_{32}) > @rst(v_{31})@rst(v_{32}) \quad \textbf{in} \\
& \qquad\qquad\quad < 1st(v_3) > @rst(v_1)@ <> @rst(v_3) \\
\overline{boo}_1(x) = & \textbf{let } v_1 = \textbf{let } v_{11} = \textbf{let } v_{111} = <x> \textbf{ in let } v_{112} = <1> \textbf{ in} \\
& \qquad\qquad\qquad\quad < 1st(v_{111}) - 1st(v_{112})@rst(v_{111})@rst(v_{112}) \quad \textbf{in} \\
& \qquad\quad \textbf{let } u_1 = \overline{foo}_1(1st(v_{11})) \textbf{ in} \\
& \qquad\quad < 1st(u_1), u_1 > @rst(v_{11}) \quad \textbf{in} \\
& \textbf{let } v_2 = \textbf{let } v_{21} = \textbf{let } v_{211} = <x> \textbf{ in let } v_{212} = <2> \textbf{ in} \\
& \qquad\qquad\qquad\quad < 1st(v_{211}) - 1st(v_{212}) > @rst(v_{211})@rst(v_{212}) \quad \textbf{in} \\
& \qquad\quad \textbf{let } u_2 = \overline{foo}_1(1st(v_{21})) \textbf{ in} \\
& \qquad\quad < 1st(u_2), u_2 > @rst(v_{21}) \quad \textbf{in} \\
& < 1st(v_1) + 1st(v_2) > @rst(v_1)@rst(v_2)
\end{array}
\tag{4.11}
$$

The straightforward extension implemented by transformation $\mathcal{E}xt$ is local and structure-preserving. However, it may introduce unnecessary bindings for values of expressions other than function applications, leave many tuple operations for passing intermediate results unsimplified, and place bindings at undesirable positions, such as within binding definitions. The result is complicated code and reduced readability.

4.2.2 Administrative simplification

Administrative simplifications are performed using a *cleaning* transformation to clean up the program obtained above. For each function definition $f(v_1, ..., v_n) = e$ obtained from the extension transformation, we obtain a function definition

$$f(v_1, ..., v_n) = \mathcal{C}lean[\![e]\!]\,\emptyset \tag{4.12}$$

where $\mathcal{C}lean[\![e]\!]\,I$ cleans up an expression e using the information set I at e, i.e., it examines subexpressions in applicative and left-to-right order, collects information sets at subexpressions, simplifies tuple operations for passing intermediate results, unwinds binding expressions that become unnecessary as a result of simplifying their subexpressions, and lifts bindings out of enclosing expressions when possible to enhance readability.

Here, an information set $I_{[e]}$ is a set of equations collected from the bindings introduced in the context of e. For example, if some $f(v_1, ..., v_n)$ is defined to be e, and e is **let** $v_1 = e_1$ **in let** $v_2 = e_2$ **in** e_3, then $I_{[e]} = \emptyset$ and $I_{[e_3]} = \{v_1 \leftrightarrow e_1, v_2 \leftrightarrow e_2\}$.

$\mathcal{C}lean$ uses a function $\mathcal{S}imp_{\mathcal{C}lean}$ to perform basic simplifications like tuple operations and binding unfolding, as summarized in Figure 4.4. Given an expression e and an information set I, we say that e can be simplified to e' under I if the corresponding condition $cond(I)$ holds, and we define $\mathcal{S}imp_{\mathcal{C}lean}[\![e]\!]I = e'$; otherwise, we define $\mathcal{S}imp_{\mathcal{C}lean}[\![e]\!]I = e$.

expression e	expression e'	condition $cond(I)$
$e_1 @ e_2$	$<e_{11}, ..., e_{1n_1}, e_{21}, ..., e_{2n_2}>$	$e_1 \leftrightarrow <e_{11}, ..., e_{1n_1}> \in I$, and $e_2 \leftrightarrow <e_{21}, ..., e_{2n_2}> \in I$
$1st(e)$	e_1	$e \leftrightarrow <e_1, e_2, ..., e_n> \in I$
$rst(e)$	$<e_2, ..., e_n>$	$e \leftrightarrow <e_1, e_2, ..., e_n> \in I$
let $v = e_1$ **in** e_2	$e_2[e_1/v]$	v is introduced by the extension and occurs at most once in e_2

Figure 4.4: Simplification for $\mathcal{C}lean$

$\mathcal{C}lean$ uses a function $\mathcal{S}ubl_{\mathcal{C}lean}$ to apply basic simplifications recursively to subexpressions and lift bindings out of enclosing expressions, as defined in Figure 4.5. As for $\mathcal{S}ubl$ in Chapter 3, the presentation of $\mathcal{S}ubl_{\mathcal{C}lean}$ is simplified by omitting detailed control structures that sequence it through the subexpressions. A subexpression is *reduced* if it is the result of having already applied $\mathcal{C}lean$ for the subexpression at that position; otherwise, it is not reduced. For an expression **let** $v = e_1$ **in** e_2 where e_1 is not itself a binding expression, if e_1 is a conditional expression, then, for further simplifying its two branches, $\mathcal{S}ubl_{\mathcal{C}lean}$ lifts the condition out; otherwise, if v is introduced by the extension transformation, $\mathcal{S}ubl_{\mathcal{C}lean}$ cleans e_2 with the assumption that v equals e_1 added to the information set.

Finally, we define the function $\mathcal{C}lean$ as in (4.13). If an expression e has subexpressions, then $\mathcal{C}lean$ calls $\mathcal{S}ubl_{\mathcal{C}lean}$ to recursively clean them. Then $\mathcal{C}lean$ calls $\mathcal{S}imp_{\mathcal{C}lean}$ to simplify the top-level expression.

$$\begin{array}{l} \mathcal{C}lean[\![e]\!]I = e'' \\ \qquad\qquad \text{where } e'' = \mathcal{S}imp_{\mathcal{C}lean}[\![e']\!]I \\ \qquad\qquad\qquad e' = \begin{cases} \mathcal{S}ubl_{\mathcal{C}lean}[\![e]\!]I & \text{if } e \text{ is not } v \\ e & \text{otherwise} \end{cases} \end{array} \tag{4.13}$$

$\mathcal{C}lean$ cleans out only some of the bindings introduced by the extension transformation and lifts some bindings and conditions. The resulting functions $\bar{f}$ still satisfy the properties stated around (4.10), i.e., the formula (4.1) holds.

After cleaning, we obtain a set of extended function definitions that are simpler, easier to read, and also easier for the subsequent stages to process. For the functions $\overline{foo}_1$ and $\overline{boo}_1$ in (4.11), after the cleaning transformation, we obtain the functions

$$
\begin{array}{ll}
\mathcal{S}ubl_{\mathcal{C}lean}[\![g(e_1,...,e_n)]\!]\ I \quad \text{where } g \text{ is } c,\ p, \text{ or } f & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![g(e_1,...,e_{i-1},e_i',e_{i+1},...,e_n)]\!]\ I & \text{if } e_1,...,e_{i-1} \text{ are reduced, not} \\
\qquad \text{where } e_i' = \mathcal{C}lean[\![e_i]\!]\ I & \quad \textbf{let}\text{, but } e_i \text{ is not reduced} \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1' \textbf{ in } g(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)]\!]\ I & \text{if } e_1,...,e_{i-1} \text{ are reduced, not} \\
\qquad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \quad \textbf{let}\text{, } e_i \text{ is reduced, but} \\
 & \quad e_i \text{ is } \textbf{let } v = e_1' \textbf{ in } e_2' \\
\quad = g(e_1,...,e_n) & \text{otherwise} \\
 & \\
\mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I & \text{if } e_1 \text{ is not reduced} \\
\qquad \text{where } e_1' = \mathcal{C}lean[\![e_1]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1' \textbf{ in if } e_2' \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I & \text{if } e_1 \text{ is reduced, and} \\
\qquad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \quad e_1 \text{ is } \textbf{let } v = e_1' \textbf{ in } e_2' \\
\quad = \textbf{if } e_1 \textbf{ then } \mathcal{C}lean[\![e_2]\!]\ I \textbf{ else } \mathcal{C}lean[\![e_3]\!]\ I & \text{otherwise} \\
 & \\
\mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1' \textbf{ in } e_2]\!]\ I & \text{if } e_1 \text{ is not reduced} \\
\qquad \text{where } e_1' = \mathcal{C}lean[\![e_1]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v' = e_1' \textbf{ in let } v = e_2' \textbf{ in } e_2]\!]\ I & \text{if } e_1 \text{ is reduced, and} \\
\qquad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \quad e_1 \text{ is } \textbf{let } v' = e_1' \textbf{ in } e_2' \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_1' \textbf{ then let } v = e_2' \textbf{ in } e_2 \textbf{ else let } v = e_3' \textbf{ in } e_2]\!]\ I & \text{if } e_1 \text{ is reduced, and} \\
\qquad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \quad e_1 \text{ is } \textbf{if } e_1' \textbf{ then } e_2' \textbf{ else } e_3' \\
\quad = \textbf{let } v = e_1 \textbf{ in } Clean[\![e_2]\!]\ I' & \text{otherwise} \\
\qquad \text{where } I' = \left\{ \begin{array}{ll} I \cup \{v \leftrightarrow e_1\} & \text{if } v \text{ is introduced} \\ I & \text{otherwise} \end{array} \right. &
\end{array}
$$

Figure 4.5: Definition of $\mathcal{S}ubl_{\mathcal{C}lean}$

$\overline{foo}$ and $\overline{boo}$ as follows:

$$
\begin{array}{ll}
\overline{foo}(x) = & \textbf{if } x \leq 2 \textbf{ then } <1, \_, \_> \\
& \textbf{else let } u_1 = \overline{boo}(x) \textbf{ in} \\
& \quad \textbf{let } u_2 = \overline{foo}(x-3) \textbf{ in} \\
& \quad < 1st(u_1){+}1st(u_2),\ u_1,\ u_2 >
\end{array}
\qquad
\begin{array}{ll}
\overline{boo}(x) = & \textbf{let } u_1 = \overline{foo}(x-1) \textbf{ in} \\
& \textbf{let } u_2 = \overline{foo}(x-2) \textbf{ in} \\
& < 1st(u_1){+}1st(u_2),\ u_1,\ u_2 >
\end{array}
\tag{4.14}
$$

4.2.3 Optimization

An obvious optimization can be incorporated into the extension transformation, i.e, we can introduce bindings only for subexpressions that contain function applications. Thus, there would be fewer tuple operations for passing intermediate results and fewer bindings to be unwound or lifted, leaving less work for the administrative simplifications.

To do this, we replace the transformation $\mathcal{E}xt$ with the transformation $\mathcal{E}xt1$ in Figure 4.6. The notion of *reduced* for $\mathcal{E}xt1$ is similar to that for $\mathcal{S}ubl_{\mathcal{C}lean}$. We use $cf(e)$ to denote that expression e contains a function application, and $ncf(e)$ to denote that e does not contain a function application.

For functions *foo* and *boo* of Figure 4.1, after the optimized extension transformation, we obtain the functions $\overline{foo}_2$ and $\overline{boo}_2$ as follows:

$$
\begin{array}{llll}
\overline{foo}_2(x) = & \multicolumn{3}{l}{\textbf{if } x \leq 2 \textbf{ then } <1, \_, \_>} \\
& \textbf{else let } v_3 = & \textbf{let } v_{31} = & \textbf{let } u_1 = \overline{boo}_2(x) \textbf{ in} \\
& & & < 1st(u_1), u_1 > @ <> \quad \textbf{in} \\
& & \textbf{let } v_{32} = & \textbf{let } u_2 = \overline{foo}_2(x-3) \textbf{ in} \\
& & & < 1st(u_2), u_2 > @ <> \quad \textbf{in} \\
& & \multicolumn{2}{l}{< 1st(v_{31}) + 1st(v_{32}) > @rst(v_{31})@rst(v_{32}) \quad \textbf{in}} \\
& \multicolumn{3}{l}{\quad < 1st(v_3) > @ <> @rst(v_3)} \\
\overline{boo}_2(x) = & \textbf{let } v_1 = & \multicolumn{2}{l}{\textbf{let } u_1 = \overline{foo}_2(x-1) \textbf{ in}} \\
& & \multicolumn{2}{l}{< 1st(u_1), u_1 > @ <> \quad \textbf{in}} \\
& \textbf{let } v_2 = & \multicolumn{2}{l}{\textbf{let } u_2 = \overline{foo}_2(x-2) \textbf{ in}} \\
& & \multicolumn{2}{l}{< 1st(u_2), u_2 > @ <> \quad \textbf{in}} \\
& \multicolumn{3}{l}{< 1st(v_1) + 1st(v_2) > @rst(v_1)@rst(v_2)}
\end{array}
\tag{4.15}
$$

Then, after the cleaning transformation on them, we obtain the same functions $\overline{foo}$ and $\overline{boo}$ as in (4.14).

4.3 Stage II: Incrementalization

Stage II derives a function $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$. Basically, one may identify subcomputations in the expanded $\bar{f}_0(x \oplus y)$ whose values can be retrieved from the cached result $\bar{r}$ of $\bar{f}_0(x)$, replace them by corresponding retrievals, and capture the resulting way of computing $\bar{f}_0(x \oplus y)$ in the incremental version $\bar{f}_0'(x, y, \bar{r})$. Such a derivation method is given in Chapter 3, and, depending on the power one expects from the derivation, the method can be made semi-automatic or fully-automatic. Two concerns specific to the prune-and-cache method and relating the different stages are addressed here.

$\mathcal{E}xt1[\![g(e_1,...,e_n)]\!]\ A \quad$ where g is c, p, or f

$$
\begin{array}{ll}
= \textbf{let } v_i = \mathcal{E}xt1[\![e_i]\!]\ \emptyset \textbf{ in } \mathcal{E}xt1[\![g(e_1,...,e_n)]\!]\ (A \cup \{\langle v_i, e_i\rangle\}) & \text{if } e_1,...,e_{i-1} \text{ are reduced,} \\
 & \quad e_i \text{ is not reduced, and } cf(e_i) \\
= \mathcal{E}xt1[\![g(e_1,...,e_n)]\!]\ A & \text{if } e_1,...,e_{i-1} \text{ are reduced,} \\
 & \quad e_i \text{ is not reduced, and } ncf(e_i) \\
= < g(e'_1,...,e'_n) > @e''_1@...@e''_n & \text{if } e_1,...,e_n \text{ are reduced, and} \\
 & \quad g \text{ is } c \text{ or } p \\
= \textbf{let } v = \bar{g}(e'_1,...,e'_n) \textbf{ in } < 1st(v) > @e''_1@...@e''_n@ < v > & \text{otherwise, i.e., if } e_1,...,e_n \text{ are} \\
 & \quad \text{reduced, and } g \text{ is } f
\end{array}
$$

$$
\text{where } e'_i = \begin{cases} 1st(v_i) & \text{if } \langle v_i, e_i\rangle \in A \\ e_i & \text{otherwise} \end{cases} \text{ and } e''_i = \begin{cases} rst(v_i) & \text{if } \langle v_i, e_i\rangle \in A \\ <> & \text{otherwise} \end{cases} \text{ for } i = 1..n
$$

$\mathcal{E}xt1[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ A$

$$
\begin{array}{ll}
= \textbf{let } v_1 = \mathcal{E}xt1[\![e_1]\!]\ \emptyset \textbf{ in } \mathcal{E}xt1[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ \{\langle v_1, e_1\rangle\} & \text{if } e_1 \text{ is not reduced, and } cf(e_1) \\
= \mathcal{E}xt1[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ \emptyset & \text{if } e_1 \text{ is not reduced, and } ncf(e_1) \\
= \textbf{if } e'_1 \textbf{ then } e^*_2 \textbf{ else } e^*_3 & \text{otherwise}
\end{array}
$$

$$
\text{where } e^*_2 = \begin{cases} \textbf{let } v_2 = \mathcal{E}xt1[\![e_2]\!]\ \emptyset \textbf{ in } < 1st(v_2) > @e''_1@rst(v_2)@\mathcal{P}ad[\![e_3]\!] & \text{if } cf(e_2) \\ < e_2 > @e''_1@\mathcal{P}ad[\![e_3]\!] & \text{otherwise} \end{cases}
$$

$$
e^*_3 = \begin{cases} \textbf{let } v_3 = \mathcal{E}xt1[\![e_3]\!]\ \emptyset \textbf{ in } < 1st(v_3) > @e''_1@\mathcal{P}ad[\![e_2]\!]@rst(v_3) & \text{if } cf(e_3) \\ < e_3 > @e''_1@\mathcal{P}ad[\![e_2]\!] & \text{otherwise} \end{cases}
$$

$$
\text{where } e'_1 = \begin{cases} 1st(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ e_1 & \text{otherwise} \end{cases} \text{ and } e''_1 = \begin{cases} rst(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ <> & \text{otherwise} \end{cases}
$$

$\mathcal{E}xt1[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ A$

$$
\begin{array}{ll}
= \textbf{let } v_1 = \mathcal{E}xt1[\![e_1]\!]\ \emptyset \textbf{ in } \mathcal{E}xt1[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ \{\langle v_1, e_1\rangle\} & \text{if } e_1 \text{ is not reduced, and } cf(e_1) \\
= \mathcal{E}xt1[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ \emptyset & \text{if } e_1 \text{ is not reduced, and } ncf(e_1) \\
= \textbf{let } v = e'_1 \textbf{ in } e^*_2 & \text{otherwise}
\end{array}
$$

$$
\text{where } e^*_2 = \begin{cases} \textbf{let } v_2 = \mathcal{E}xt1[\![e_2]\!]\ \emptyset \textbf{ in } < 1st(v_2) > @e''_1@rst(v_2) & \text{if } cf(e_2) \\ < e_2 > @e''_1 & \text{otherwise} \end{cases}
$$

$$
\text{where } e'_1 = \begin{cases} 1st(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ e_1 & \text{otherwise} \end{cases} \text{ and } e''_1 = \begin{cases} rst(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ <> & \text{otherwise} \end{cases}
$$

Figure 4.6: Definition of $\mathcal{E}xt1$

First, secondary to the goal of making the incremental program $\bar{f}_0'(x, y, \bar{r})$ as fast as possible, we want to make it use as few different intermediate results in $\bar{r}$ as possible. To do this, we require that the derivation not use intermediate results that are embedded in the results of enclosing computations so that the unused intermediate results can be pruned out by Stage III. We could improve Stage I to avoid caching intermediate results that can be statically determined to be embedded in the results of enclosing computations. But we may do better by addressing the issue also in Stage II, where we may have more powerful reasoning support.

Second, not only do we want $\bar{f}_0'(x, y, \bar{r})$ to be no slower than $f_0(x \oplus y)$, as can be guaranteed with the approach in Chapter 3, but we also want it to be no slower than $f_0'(x, y, r)$. To assure this, we require that the derivation replace a subcomputation in the expanded $\bar{f}_0(x \oplus y)$ by a retrieval from an intermediate result in $\bar{r}$ other than $1st(\bar{r})$ only if the subcomputation is also a subcomputation in $\bar{f}_0(x)$.[2] This requirement helps assure that caching intermediate results is worthwhile, i.e., the time spent in maintaining intermediate results will not surpass that saved by using them, as will be explained in Section 4.4.1.

Consider the function $\overline{foo}$ that caches all intermediate results of *foo* in (4.14). To derive an incremental version of $\overline{foo}$ under $\oplus$ using the approach in Chapter 3, we transform $\overline{foo}(x \oplus y) = \overline{foo}(x+1)$, with $\overline{foo}(x) = \bar{r}$:

1. unfold $\overline{foo}(x+1)$, simplify primitives

$$
\begin{array}{ll}
= & \textbf{if } x \le 1 \textbf{ then } <1,\_,\_> \\
& \textbf{else let } u_{11} = \overline{foo}(x) \textbf{ in} \\
& \quad\quad \textbf{let } u_{12} = \overline{foo}(x-1) \textbf{ in} \\
& \quad\quad \textbf{let } u_1 = <1st(u_{11})+1st(u_{12}), \\
& \quad\quad\quad\quad\quad\quad u_{11},\ u_{12}> \textbf{ in} \\
& \quad\quad \textbf{let } u_2 = \overline{foo}(x-2) \textbf{ in} \\
& \quad\quad <1st(u_1)+1st(u_2),\ u_1,\ u_2>
\end{array}
$$

2. separate cases, replace applications of $\overline{foo}$ by retrievals

$$
\begin{array}{ll}
= & \textbf{if } x \le 1 \textbf{ then } <1,\_,\_> \\
& \textbf{else if } x = 2 \textbf{ then} \\
& \quad\quad <3,\ <2, <1,\_,\_>, <1,\_,\_>>,\ <1,\_,\_>> \\
& \textbf{else let } u_{11} = \bar{r} \textbf{ in} \\
& \quad\quad \textbf{let } u_{12} = 2nd(2nd(\bar{r})) \textbf{ in} \\
& \quad\quad \textbf{let } u_1 = <1st(u_{11})+1st(u_{12}),\ u_{11},\ u_{12}> \textbf{ in} \\
& \quad\quad \textbf{let } u_2 = 3rd(2nd(\bar{r})) \textbf{ in} \\
& \quad\quad <1st(u_1)+1st(u_2),\ u_1,\ u_2>
\end{array}
$$

and we obtain an incremental function $\overline{foo}'$ such that, if $\overline{foo}(x) = \bar{r}$, then $\overline{foo}'(x, \bar{r}) = \overline{foo}(x+1)$, as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\overline{foo}'(x, \bar{r})} \\
= & \textbf{if } x \le 1 \textbf{ then } <1,\_,\_> \\
& \textbf{else if } x = 2 \textbf{ then } <3,\ <2, <1,\_,\_>, <1,\_,\_>>,\ <1,\_,\_>> \\
& \textbf{else} <1st(\bar{r})+1st(2nd(\bar{r})),\ <1st(\bar{r})+1st(2nd(2nd(\bar{r}))), \bar{r}, 2nd(2nd(\bar{r}))>,\ 3rd(2nd(\bar{r}))>
\end{array}
\tag{4.16}
$$

Clearly, $\overline{foo}'(x, \bar{r})$ computes $\overline{foo}(x+1)$ in only $O(1)$ time.

4.4 Stage III: Pruning

The input to the pruning stage is $\bar{f}_0$, a function that caches all intermediate results of f_0 obtained from Stage I, and $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$ obtained from Stage II, together with a set of other function definitions used in computing $\bar{f}_0$ and

[2] In practice, this is the best any derivation method could do without the power of a general theorem prover as discussed in Chapter 3.

$\bar{f_0}'$, obtained from Stage I and II. The goal is to prune $\bar{f_0}$, so that it returns only the value of f_0 and the intermediate results useful for incremental computation under $\oplus$, and prune $\bar{f_0}'$, so that it incrementally computes only the value of f_0 and the useful intermediate results.

To achieve this goal, we analyze function $\bar{f_0}'$ to determine the components of $\bar{r}$, the value of $\bar{f_0}(x)$, on which $1st(\bar{f_0}'(x,y,\bar{r}))$, the value of $f_0(x \oplus y)$, transitively depends. Two issues arise as we need to maintain these components: transitive dependencies and cost. We first depict the transitive dependencies and address the cost issue. Then we give an algorithm that computes the needed components based on a dependency analysis using domain projections [Sco82,Gun92]. With this result, we prune both the functions $\bar{f_0}$ and $\bar{f_0}'$.

4.4.1 Maintaining intermediate results: transitive dependency and cost

Transitive dependency. The function application $\bar{f_0}'(x,y,\bar{r})$ returns the value of $f_0(x \oplus y)$ in the first component and all corresponding intermediate results in the other components. To determine which components in the value $\bar{r}$ are needed for incremental computation, we start with the first component in the value of $\bar{f_0}'(x,y,\bar{r})$ and find out the components of $\bar{r}$ on which this value depends. These components may include those other than the first one of $\bar{r}$. To support incremental computation after further input changes, we need to maintain these components of $\bar{f_0}'(x,y,\bar{r})$ as well as the first component. They may depend on even other components of $\bar{r}$, forming a kind of transitive dependency.

Figure 4.7 illustrates the transitive dependencies for the example *foo* under change operation $\oplus$. By definitions of *foo* and *boo* and associativity of '+', we have

$$\begin{aligned} foo(x+1) &= boo(x+1) + foo(x-2) \\ &= (foo(x)+foo(x-1)) + foo(x-2) \\ &= foo(x) + (foo(x-1)+foo(x-2)) = foo(x) + boo(x). \end{aligned}$$

Thus, to compute the value v_1' of $foo(x+1)$, $\overline{foo}'$ uses the value v_1 of $foo(x)$ and the intermediate result v_2 of $boo(x)$ returned by $\overline{foo}(x)$. Therefore, the corresponding value v_1' of $foo(x+1)$ and the intermediate result v_2' of $boo(x+1)$ need to be maintained. The value v_1' of $foo(x+1)$ has just been considered. To compute the intermediate result v_2' of $boo(x+1)$, $\overline{foo}'$ uses the value v_1 of $foo(x)$ and the intermediate result v_3 of $foo(x-1)$ returned by $\overline{foo}(x)$. Therefore, the corresponding value v_1' of $foo(x+1)$ and the intermediate result v_3' of $foo(x)$ also need to be maintained. Again, the value v_1' of $foo(x+1)$ has just been considered. To compute the value v_3' of $foo(x)$, $\overline{foo}'$ just uses the value v_1 of $foo(x)$ returned by $\overline{foo}(x)$.

Thus, to summarize, the value v_1' of $foo(x+1)$ transitively depends on the components of intermediate results corresponding to v_1, v_2, and v_3, which are maintained as $v_1' = v_1 + v_2$, $v_2' = v_1 + v_3$, and $v_3' = v_1$, respectively. Other components of intermediate results are not needed and therefore do not need to be computed or maintained; they can be be pruned out.

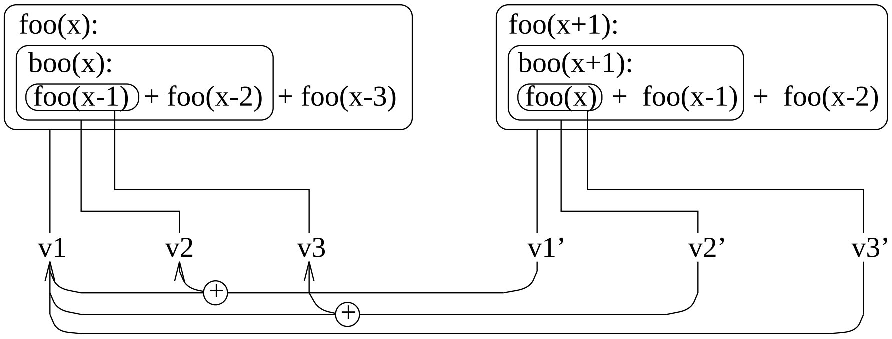

Figure 4.7: Transitive dependencies

Cost. Is it always true that the time spent in maintaining intermediate results will not surpass that saved by using them?

First, we consider the problem in general. Given a way of computing a function f, let g be a function that computes some intermediate results of f in the way f does, and let $\hat{f} = \langle f, g \rangle$. Suppose $f'(x, y, r)$ computes $f(x \oplus y)$ given $r = f(x)$, and $\hat{f}'(x, y, r)$ computes $\hat{f}(x \oplus y)$ given $\hat{r} = \hat{f}(x)$. Then in general, it is not true that $t(\hat{f}'(x, y, \hat{r})) \leq t(f'(x, y, r))$. This is mainly because f, and thus g, could be arbitrary. This is true even if all these functions compute with the best asymptotic time. What it says is that maintaining arbitrary intermediate results may not be worthwhile for incremental computation.

But, consider the particular functions f' and $\hat{f}'$ derived using the derivation approach. Recall the second requirement in incrementalization: the derivation replaces a subcomputation in the expanded $\hat{f}_0(x \oplus y)$ by a retrieval from an intermediate result in $\hat{r}$ other than $1st(\hat{r})$ only if the subcomputation is also a subcomputation in $\hat{f}_0(x)$. Thus, suppose we compute $\hat{f}(x \oplus y)$ using the cached result $\hat{r}$ of $\hat{f}(x)$, and suppose computing $1st(\hat{f}(x \oplus y))$, i.e., $f(x \oplus y)$, uses a subcomputation $g(x)$ in $f(x)$, and the value of $g(x)$ can be retrieved from $\hat{r}$ but not $1st(\hat{r})$, i.e., r. Then, on the one hand, the value of $g(x \oplus y)$ needs to be maintained by $\hat{f}'(x, y, \hat{r})$; on the other hand, if we compute $f(x \oplus y)$ using only the cached result r of $f(x)$, then the subcomputation $g(x)$ remains in $f'(x, y, r)$, i.e., $f'(x, y, r)$ has the cost of recomputing $g(x)$.

Now, for intermediate results of f like the value of g above, if (a) the size of y is bounded, (b) when the size of y is bounded, the time of computing $x \oplus y$ is bounded, and (c) g is at most *linear-power exponential* time, i.e., g is polynomial time or exponential time but with linear exponent, then we have

$$t(\hat{f}'(x, y, \hat{r})) \leq t(f'(x, y, r)). \tag{4.17}$$

It is easy to see that the three conditions are true for all practical and feasible incremental applications and, therefore, we assume that they are satisfied. To prove

(4.17), we notice that

$$
\begin{array}{lll}
t(\hat{f}'(x,y,\hat{r})) & \leq t(f'(x,y,r)) + t(x \oplus y) + t(g(x')) & \text{by definition of } \hat{f}' \text{ and derivation} \\
& \leq t(f'(x,y,r)) + t(g(x)) & \text{by conditions on } y, \oplus, \text{ and } g \\
& \leq t(f'(x,y,r)) + t(f'(x,y,r)) & \text{by } g \text{ being subcomputation of } f' \\
& \leq t(f'(x,y,r)) & \text{by definition of } t
\end{array}
$$

We conclude that, with the conditions above, using and maintaining intermediate results is always asymptotically at least as fast. Therefore, in order to achieve as fast incremental computation as possible, we should compute the closure of the transitive dependencies for maintaining intermediate results.

4.4.2 Dependency analysis using projections

We first describe our use of projections to represent components of the tuple values constructed in Stage I and manipulated by Stage II. Then, we give a backward dependency analysis that determines which components of $\bar{r}$ are needed for computing certain components of $\bar{f}_0'(x,y,\bar{r})$. Finally, we present an algorithm that computes the closure of the transitive dependencies for maintaining intermediate results.

Projections. Our domain of interest D contains $\perp$, indicating a computation diverges, values d returned by functions in the original program f_0, and constructed tuples $<d_1,...,d_n>$, where each d_i is (recursively) an element of D (other than $\perp$). The length of a constructed tuple is statically bounded, but the depth of tuple nesting may not be bounded, since it is dynamically determined. Intuitively, any components of a constructed tuple value can be replaced by the dummy constant $\_$, introduced in Stage I, if we do not care about the values of those components. If a subcomputation involves $\_$, then the result of that subcomputation is $\_$, but the result of the parent computation need not be $\_$. For any value d in domain D, $\perp \sqsubseteq d$. For two values d_1 and d_2 other than $\perp$'s in D, $d_1 \sqsubseteq d_2$ iff

$$
\begin{array}{l}
d_1 = \_, \quad d_1 = d_2, \quad \text{or} \\
d_1 = <d_{11},...,d_{1n}>, \ d_2 = <d_{21},...,d_{2n}>, \ \text{and } d_{1i} \sqsubseteq d_{2i} \text{ for } i=1..n.
\end{array}
$$

A projection over the domain D is a function $\Pi : D \to D$ such that $\Pi(\Pi(d)) = \Pi(d) \sqsubseteq d$ for any $d \in D$. Three important projections are ID, ABS, and BOT. ID is the identity function $ID(d) = d$. ABS is the function $ABS(d) = \_$ for any $d \neq \perp$. BOT is the function $BOT(d) = \perp$.

A non-bottom projection Π of interest here can be represented as a set of selection functions π, each of which is a sequence of 1^{st}, 2^{nd}, ..., n^{th}. The null sequence is denoted ϵ. Intuitively, if Π contains a sequence $i_k^{th} i_{k-1}^{th} ... i_1^{th}$, then the i_kth element of the i_{k-1}th element of the $\cdots$ of the i_1th element of Π's argument is selected, and if Π contains ϵ, then all components of Π's argument are selected. A projection Π replaces those components of its argument that are not selected with the constant $\_$. For example $\{1^{st}\}$, $\{1^{st}, 1^{st}2^{nd}\}$, and $\{1^{st}1^{st}2^{nd}, \epsilon\}$ are projections, and

$$
\{1^{st}, 1^{st}2^{nd}\}(<d_1, <<d_{211}, d_{212}>, d_{22}>>) \;=\; <d_1, <<d_{211}, d_{212}>, \_ >>
$$

For convenience of presentation, we use $\Pi_{(i)}$ to denote the set $\{\pi \mid \pi\, i^{th} \in \Pi\}$, i.e., $\Pi_{(i)}$ is the part of Π that considers the ith component. With the set representation, a projection $\Pi = ID$ iff $\epsilon \in \Pi$ or $\Pi_{(i)} = ID$ for $i = 1..n$ for arguments of Π of length n. A projection $\Pi = ABS$ iff $\Pi = \emptyset$. For $\Pi \notin \{ID, ABS\}$, $\Pi(< d_1, ..., d_n >) = < \Pi_{(1)}(d_1), ..., \Pi_{(n)}(d_n) >$. For any two projections Π_1 and Π_2 other than *BOT*'s, $\Pi_1 \sqsubseteq \Pi_2$ iff

$\Pi_1 = ABS$, $\Pi_2 = ID$, or
$\Pi_{1(i)} \sqsubseteq \Pi_{2(i)}$ for $i = 1..n$ for arguments of Π_1 and Π_2 of length n.

Dependency analysis. To compute which components of $\bar{r}$ are needed for computing certain components of $\bar{f}_0'(x, y, \bar{r})$, we apply a backward dependency analysis to the program $\bar{f}_0'$.

Following the style of [WH87], for each function f of n parameters, and each i from 1 to n, we define f^i to be a *dependency transformer* that takes a projection that is applied to the result of f and returns a projection that is *sufficient* to be applied to the ith parameter. The *sufficiency condition* that f^i must satisfy is: if $\Pi_i = f^i\Pi$ then

$$\Pi(f(v_1, ..., v_i, ..., v_n)) \sqsubseteq f(v_1, ..., \Pi_i(v_i), ..., v_n) \tag{4.18}$$

Similarly, we define e^v to be a dependency transformer that takes a projection that is applied to e and returns a projection that is sufficient to be applied to every instance of v in e. A similar sufficiency condition must be satisfied: if $\Pi' = e^v\Pi$ then

$$\Pi(e) \sqsubseteq e[\Pi'(v)/v] \tag{4.19}$$

For a function f whose definition is $f(v_1, ..., v_n) = e$, we define $f^i\Pi = e^{v_i}\Pi$. The definition of e^v may in turn refer to f^i, thus the definitions may be mutually recursive. We define

$$e^v BOT = BOT \qquad \text{and} \qquad e^v ABS = ABS. \tag{4.20}$$

For $\Pi \neq BOT, ABS$, we give the definition of $e^v\Pi$ in Figure 4.8. Note that, the argument Π to e^v must be *ID* if e is a variable whose value is not a constructed tuple, or an application of a constructor or a primitive function that is not $<>$ or ith. We can easily show that each rule guarantees sufficient information. Thus, the sufficiency conditions are satisfied by recursion induction.

Let $i_{\bar{r}}$ be the index of $\bar{r}$ in the parameters of $\bar{f}_0'$. With the above definitions, we know that $\bar{f}_0'^{\,i_{\bar{r}}}\Pi$ computes how much of $\bar{r}$ is needed when Π of $\bar{f}_0'(x, y, \bar{r})$ is needed.

To compute $f^i\Pi$ for some f^i and $\Pi \neq BOT, ABS$ (otherwise, we can use (4.20)), if the definition of f^i does not involve recursion, then we can compute directly using the definition. If the definition of f^i involves recursion, then the argument projections and resulting projections of some dependency transformers may contain selection functions of unbounded depth. To approximate the result, we restrict the selection functions of the projections to be of bounded depth d, namely, if a projection contains a selection function $i_k^{th}\, i_{k-1}^{th} ... i_1^{th}$ but $k > d$, then we truncate it to $i_d^{th}\, i_{d-1}^{th} ... i_1^{th}$. A simple choice for the depth bound would be 1. A more prudent choice could be the length of the longest cycle that contains f in the call graph. This limits the

$$
\begin{array}{lll}
v^v\,\Pi & = & \Pi \\
u^v\,\Pi & = & \mathit{ABS} \qquad\qquad\qquad \text{if } v \neq u \\
(< e_1, ..., e_n >)^v\,\Pi & = & e_1^v\,\Pi_{(1)} \;\cup\; ... \;\cup\; e_n^v\,\Pi_{(n)} \\
(ith(e))^v\,\Pi & = & e^v\,\{\pi\, i^{th} \mid \pi \in \Pi\} \\
(g(e_1, ..., e_n))^v\,\Pi & = & e_1^v\; \mathit{ID} \;\cup ... \cup\; e_n^v\, \mathit{ID} \qquad \text{if } g \text{ is } c \text{ or } p \text{ but not } <> \text{ or } ith \\
(f(e_1, ..., e_n))^v\,\Pi & = & e_1^v(f^1\,\Pi) \;\cup\; ... \;\cup\; e_n^v(f^n\,\Pi) \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^v\,\Pi & = & e_1^v\; \mathit{ID} \;\cup e_2^v\Pi \;\cup\; e_3^v\Pi \\
(\textbf{let } u = e_1 \textbf{ in } e_2)^v\,\Pi & = & e_1^v(e_2^u\Pi) \;\cup\; e_2^v\Pi
\end{array}
$$

Figure 4.8: Definition of $e^v\Pi$ for $\Pi \neq \mathit{BOT}, \mathit{ABS}$

domain of projections to be finite. Now, to solve the recursive definitions of these dependency transformers, we just compute the limits of the ascending chains by starting at $f^i\Pi = \mathit{ABS}$ for every f^i and Π and iterating using the definitions. This iteration with the approximated domain of projections always terminates, since when the depth of nesting being examined is bounded, the ascending chains are finite.

Computing the closure of the transitive dependencies. To compute the components of $\bar{r}$ on which $1st(\bar{f}_0'(x, y, \bar{r}))$ transitively depends, we start with Π being $\{1^{st}\}$ and compute the smallest projection Π of $\bar{r}$ on which $\Pi(\bar{f}_0'(x, y, \bar{r}))$ depends, i.e., the smallest projection Π such that

$$\{1^{st}\} \sqsubseteq \Pi \;\text{ and }\; \Pi(\bar{f}_0'(x, y, \bar{r})) \sqsubseteq \bar{f}_0'(x, y, \Pi(\bar{r})). \tag{4.21}$$

Of course, the projection $\Pi = \mathit{ID}$ is always a solution. But our goal is to make Π as small as possible, and thus to avoid as much unnecessary caching as possible.

Since $\bar{f}_0'^{\bar{r}}\Pi$ computes the components of $\bar{r}$ on which $\Pi(\bar{f}_0'(x, y, \bar{r}))$ depends, we define

$$
\begin{array}{lll}
\Pi^{(0)} & = & \{1^{st}\} \\
\Pi^{(i+1)} & = & \Pi^{(i)} \;\cup\; \bar{f}_0'^{\bar{r}}\,\Pi^{(i)}
\end{array}
\tag{4.22}
$$

and compute the least fixed point of Π. In other words, Π is the least projection that satisfies $\{1^{st}\} \sqsubseteq \Pi$ and $\bar{f}_0'^{\bar{r}}\Pi \sqsubseteq \Pi$. We call this projection the *closure projection*. Note that the above computation always terminates since $\bar{f}_0'^{\bar{r}}\Pi^{(i)}$ terminates and returns only sets of selection functions of bounded depth.

The time complexity of the closure computation depends on the required size of the projection domain and the complexity of the dependency analysis. Suppose d is the maximum depth of selection functions we consider, and l is the maximum length of the constructed tuples, i.e., the largest number of function applications in a function definition in the program f_0. Then the maximum number c of disjoint components in these projections is at most l^d, which characterizes the maximum size of the projection domain.

We estimate the complexity of the dependency analysis in the simplest manner. Consider the program $\bar{f}'_0$. Let n be the number of function definitions, and a be the maximum number of parameters in any of these definitions. Then there are at most na dependency transformers. Since an argument projection may contain any of c components, there are at most 2^c argument projections to each transformer. Thus, the number of projections $f^i\Pi$ to be computed is at most $na2^c$. Now, let s_f be the maximum number of transformers used in a transformer definition, i.e., the number of function applications in a function definition. Being careful, we can recompute each $f^i\Pi$ only when any computed projections used by $f^i\Pi$ change, where each can change at most c times. Thus, the total number of computations of $f^i\Pi$ using its immediate definition is at most $na2^c cs_f$. Each such computation takes at most sc time, where s be the maximum size of a function definition, i.e., the number of subexpressions in the defining expression, and c is the time needed to compute operations, such as union, on two projections. Therefore, the total time is at most $na2^c c^2 ss_f$.

If we limit depth of selection functions to be independent of the number of function definitions, then a, c, s, and s_f are all constant factors determined by the size of a function definition. Thus the total time is linear in the number of function definitions, although the constant factors could be very big.

Now that the above estimate includes the computations of all $f^i\Pi$, computing the dependency closure takes at most c projection unions, each taking at most c time. Thus, the total time of closure computation can be no worse than the above bound.

Example. Applying the dependency analysis to the function $\overline{foo}'$ in (4.16), we get

$$\begin{aligned}
\overline{foo'}^2\Pi = {} & (x \leq 1)^{\bar{r}}\ ID\ \cup (<1,_,_>)^{\bar{r}}\Pi \cup \\
& (x = 2)^{\bar{r}}\ ID\ \cup (<3,<2,<1,_,_>,<1,_,_>>,<1,_,_>>)^{\bar{r}}\Pi \cup \\
& (<1st(\bar{r})+1st(2nd(\bar{r})),\ <1st(\bar{r})+1st(2nd(2nd(\bar{r}))),\ \bar{r},\ 2nd(2nd(\bar{r}))>,\ 3rd(2nd(\bar{r}))>)^{\bar{r}}\Pi \\
= {} & (1st(\bar{r})+1st(2nd(\bar{r})))^{\bar{r}}\Pi_{(1)} \cup \\
& (1st(\bar{r})+1st(2nd(2nd(\bar{r}))))^{\bar{r}}\Pi_{(2)(1)} \cup (\bar{r})^{\bar{r}}\Pi_{(2)(2)} \cup (2nd(2nd(\bar{r})))^{\bar{r}}\Pi_{(2)(3)} \cup \\
& (3rd(2nd(\bar{r})))^{\bar{r}}\Pi_{(3)}
\end{aligned}$$

For this example, since the definition of $\overline{foo'}^2$ is not recursive, we can compute $\overline{foo}'^2\Pi$ for a given Π directly without iteration and approximation. For example,

$$\begin{aligned}
\overline{foo}'^2\{1^{st}\} &= \{1^{st}, 1^{st}2^{nd}\} \\
\overline{foo}'^2\{1^{st}2^{nd}\} &= \{1^{st}, 1^{st}2^{nd}2^{nd}\} \\
\overline{foo}'^2\{1^{st}2^{nd}2^{nd}\} &= \{1^{st}\}
\end{aligned}$$

which illustrates the dependencies depicted in Figure 4.7. An example where the dependency transformer is defined recursively is shown in the merge sort example in Section 4.6. Now, we compute the projection for the closure of the transitive dependencies:

$$\begin{aligned}
\Pi^{(1)} &= \Pi^{(0)} \cup \overline{foo}'^2\Pi^{(0)} &&= \{1^{st}, 1^{st}2^{nd}\} \\
\Pi^{(2)} &= \Pi^{(1)} \cup \overline{foo}'^2\Pi^{(1)} &&= \{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\} \\
\Pi^{(3)} &= \Pi^{(2)} \cup \overline{foo}'^2\Pi^{(2)} &&= \{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}
\end{aligned}$$

We obtain the projection $\{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}$.

4.4.3 Pruning under the closure projection

With the closure projection Π obtained above, this section prunes the extended function $\bar{f}_0$ to get a function $\hat{f}_0$ such that $\Pi(\bar{f}_0(x)) \sqsubseteq \hat{f}_0(x)$, and prune the incremental function $\bar{f}_0'$ to get a function $\hat{f}_0'$ such that $\Pi(\bar{f}_0'(x,y,\bar{r})) \sqsubseteq \hat{f}_0'(x,y,\Pi(\bar{r}))$. Of course, setting $\hat{f}_0$ to be $\bar{f}_0$ and $\hat{f}_0'$ to be $\bar{f}_0'$ would always work, but we only want to do this if Π is *ID*, otherwise we want to make $\hat{f}_0(x)$ as close to $\Pi(\bar{f}_0(x))$, and $\hat{f}_0'(x,y,\Pi(\bar{r}))$ as close to $\Pi(\bar{f}_0'(x,y,\bar{r}))$ as possible, and thereby avoid caching and maintaining unnecessary intermediate results as much as possible.

To do this. For each expression e that defines a function $f(v_1, ...v_n)$, we associate a projection with each subexpression of e indicating how much of the subexpression is needed assuming Π of $\bar{f}_0$ (respectively $\bar{f}_0'$) is needed. The definition and computation of the associated projections can be done in a fashion similar to the dependency analysis. For the program $\bar{f}_0'$ and the closure projection Π, the final projection computed associated with each variable will be the same as computed for the variable using dependency analysis.

When the computation reaches the limit of the ascending chain of projections, subexpressions associated with *ID* are left unchanged in the resulting function, and subexpressions associated with *ABS* are replaced by $\_$. If a variable whose value is a constructed tuple is associated with a projection Π other than *ID* or *ABS*, then we construct a tuple with the components selected by Π filled with the corresponding selections and the rest filled with $\_$. For example, if a variable v is associated with a projection $\{1^{st}, 1^{st}2^{nd}\}$, and v represents a tuple of length three whose second component is a tuple of length two, then v is replaced by $<1st(v),\ <1st(2nd(v)), \_>,\ \_>$.

As the result of such replacements, we have $\Pi(\bar{f}_0(x)) \sqsubseteq \hat{f}_0(x)$, but not $\hat{f}_0(x) = \Pi(\bar{f}_0(x))$ as anticipated in Section 4.1. Nevertheless, the resulting $\hat{f}_0$ is still good enough to guarantee (4.7). We can just project $\Pi(\bar{r})$ out of the return value of $\hat{f}_0(x)$. But we do have $\hat{f}_0'(x,y,\Pi(\bar{r})) = \Pi(\bar{f}_0'(x,y,\bar{r}))$. Thus, assuming $\hat{r} = \Pi(\bar{r})$, we have (4.8). As a matter of fact, we intend to use the function $\hat{f}_0$ only once to get the initial value, and then use the function $\hat{f}_0'$ repeatedly to compute all successive values. Recall that $\hat{f}_0'$ incrementally computes the desired output and the corresponding intermediate results, as shown in (4.8).

Consider the functions $\overline{foo}$ and $\overline{boo}$ in (4.14). Only $\{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}$ of $\overline{foo}(x)$ is needed, therefore only $\{1^{st}, 1^{st}2^{nd}\}$ of $\overline{boo}(x)$ is needed, and therefore only $\{1^{st}\}$ of $\overline{foo}(x-3)$ is needed. Thus, $\widehat{boo}_1(x)$ returns only $\{1^{st}, 1^{st}2^{nd}\}$ of $\overline{boo}(x)$, and $\widehat{foo}_1(x)$ returns only $\{1^{st}\}$ of $\overline{foo}(x)$ and the result $\{1^{st}, 1^{st}2^{nd}\}$ of $\widehat{boo}_1(x)$, as follows:

$$
\begin{array}{rl}
\widehat{foo}_1(x) = & \textbf{if } x \le 2 \textbf{ then } <1, \_, \_> \\
& \textbf{else let } u_1 = \widehat{boo}_1(x) \textbf{ in} \\
& \quad \textbf{let } u_2 = \widehat{foo}_1(x-3) \textbf{ in} \\
& \quad <1st(u_1)+1st(u_2),\ <1st(u_1), <1st(2nd(u_1)), \_, \_>, \_>,\ \_> \\
\widehat{boo}_1(x) = & \textbf{let } u_1 = \widehat{foo}_1(x-1) \textbf{ in} \\
& \textbf{let } u_2 = \widehat{foo}_1(x-2) \textbf{ in} \\
& <1st(u_1)+1st(u_2),\ <1st(u_1), \_, \_>,\ \_>
\end{array}
\tag{4.23}
$$

Consider the function $\overline{foo}'$ in (4.16). Only $\{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}$ of $\overline{foo}'(x, y, \bar{r})$ is needed. Thus, $\widehat{foo}_1(x, y, \hat{r}_1)$ returns only the corresponding components. We have, if $\widehat{foo}_1(x) = \hat{r}_1$, then $\widehat{foo}'_1(x, \hat{r}_1) = \widehat{foo}_1(x+1)$.

$$
\begin{array}{l}
\widehat{foo}'_1(x, \hat{r}_1) \\
= \ \textbf{if } x \leq 1 \textbf{ then } <1, \_, \_> \\
\quad \textbf{else if } x = 2 \textbf{ then } <3, <2, <1, \_, \_>, <\_, \_, \_>>, <\_, \_, \_>> \\
\quad \textbf{else } <1st(\hat{r}_1)+1st(2nd(\hat{r}_1)),\ <1st(\hat{r}_1)+1st(2nd(2nd(\hat{r}_1))), <1st(\hat{r}_1), \_, \_>, \_>,\ \_>
\end{array}
\tag{4.24}
$$

Simplification. After the replacements, a number of simplifications can be applied to the resulting functions: (a) unfolding a **let** expression if a binding variable occurs at most once in the body due to some replacements by _'s, (b) combining unnecessarily split components resulting from some replacements for variables whose values are constructed tuples, (c) lifting common selection computations to avoid unnecessarily computing a compound value and using only part of it, and (d) replacing occurrences of $1st(\hat{f}(e_1, ..., e_n))$ by occurrences of $f(e_1, ..., e_n)$.

For the function $\widehat{foo}_1$ in (4.23), we unfold the binding for u_2, replace the occurrence of $1st(\widehat{foo}_1(x-3))$ by $foo(x-3)$, and merge separate components of u_1. For the function $\widehat{boo}_1$ in (4.23), we unfold the binding for u_2, replace the occurrence of $1st(\widehat{foo}_1(x-2))$ by $foo(x-2)$, and lift $1st(u_1)$. We obtain

$$
\begin{array}{ll}
\widehat{foo}_2(x) = & \textbf{if } x \leq 2 \textbf{ then } <1, \_, \_> \\
& \textbf{else let } u_1 = \widehat{boo}_2(x) \textbf{ in} \\
& \quad <1st(u_1)+foo(x-3),\ u_1,\ \_>
\end{array}
\qquad
\begin{array}{ll}
\widehat{boo}_2(x) = & \textbf{let } v_1 = foo(x-1) \textbf{ in} \\
& <v_1+foo(x-2),\ <v_1, \_, \_>,\ \_>
\end{array}
\tag{4.25}
$$

Function $\widehat{foo}'_1$ remains the same.

Finally, we can eliminate _ components. But we must be careful if such a component precedes a non-_ component in a tuple, since our selectors ith's follow the indexing, which need to be changed accordingly. In particular, if k of the components preceding a component i are eliminated from a tuple, we must replace all uses of the selector ith for the tuple with $(i-k)$th. This elimination needs to be done consistently for $\hat{f}_0$ and $\hat{f}'_0$. At the end, we obtain the function $\hat{f}_0$, which caches only the useful intermediate results for incremental computation under $\oplus$, and the function $\hat{f}'_0$, which incrementally maintains only the useful intermediate results.

Theses simplifications and eliminations can be fully automated. For the functions $\widehat{foo}_2$ and $\widehat{boo}_2$ in (4.25) and $\widehat{foo}'_1$ in (4.24), we eliminate unnecessary _ components and obtain the functions $\widehat{foo}$, $\widehat{boo}$, and $\widehat{foo}'$ as given in Figure 4.2. The overall effect is that only $\{1^{st}\}$ and part of $\{2^{nd}\}$ are returned; and for the part of $\{2^{nd}\}$, only $\{1^{st}\}$ and part of $\{2^{nd}\}$ is returned; and for the part of $\{2^{nd}\}$ of $\{2^{nd}\}$, only $\{1^{st}\}$ is returned.

4.5 Discussion

We have obtained not only the extended function $\hat{f}_0$, which caches appropriate intermediate results, but also the corresponding function $\hat{f}'_0$ that incrementally maintains these intermediate results. The functions $\hat{f}_0$ and $\hat{f}_0'$ preserve the semantics of computations and compute asymptotically at least as fast, as described in (4.7) and (4.8).

The cache-and-prune method consists of three independent stages, and thus is modular. It has certain nice properties. Stage I gives us maximality by providing all the intermediate results possibly used by Stage II. Stage II uses these intermediate results for the exclusive purpose of incrementalization. Stage III gives us a kind of minimality by preserving only the intermediate results actually used by Stage II. Therefore, the whole approach is optimal with respect to the incrementalization method of Stage II. Stages I and III are simple, clean, and fully-automatable.

Transformation and analysis techniques. In cooperation with the approach for deriving incremental programs, we achieve the goal of identifying and maintaining intermediate results useful for incremental computation. The idea of caching all intermediate results followed by incrementalization can be regarded as a realization of the reduction from Kleene's course-of-values recursion to primitive recursion [Kle52, Ner95]; subsequent pruning straightforwardly eliminates unnecessary computations in the resulting function. We summarize techniques that are relevant to the program analyses and transformations used for caching and pruning.

First, the transformation $\mathcal{E}xt$ is similar to the construction of call-by-value complete recursive programs by Cartwright [Car84]. However, a call-by-value computation sequence returned by such a program is a flat list of all intermediate results, while our extended function returns a computation tree, a structure that mirrors the hierarchy of function calls. The transformations in Stage I also mimics the CPS transformations in some aspects [Plo75,LD93]: sequencing subexpressions, naming intermediate results, passing the collected information, and performing administrative reductions on the resulting program. However, they are simpler than the CPS transformations since the collected intermediate results are passed directly to the return values, rather than to the continuation functions.

Second, the backward dependency analysis and pruning transformations in Stage III use domain projections to specify sufficient information, which is natural and thus simple. Other uses of projections include the strictness analysis by Wadler and Hughes [WH87], where necessary information needs to be specified and thus accounts for some complications, and binding-time analysis by Launchbury [Lau89], which is a forward analysis and is proved equivalent to strictness analysis [Lau91]. The necessity interpretation by Jones and Le Métayer [JL89] is in the same spirit of our analysis, where their notion of necessity patterns correspond to our notion of projections. While necessity patterns specify heads and tails of list values, the projections here specify specific components of tuple values and thus provide more accurate information.

Since the dependency analysis and pruning transformations simply eliminate dead components and related computations on compound values, it would be useful for general program optimizations in context. For example, in many functional programs, we create compound values only to take them apart somewhere else, and perhaps we only use some of the components. It would be nice to avoid constructing and passing the unnecessary components. Related work is done in optimizing compilers that eliminate unnecessary tuple constructions and destructions in functional programs; for example, the Id compiler [Tra86] does tuple elimination. We think our analyses and transformations provide a straightforward solution to such problems. For us, it is more lightweight than trying to adopt any of the existing techniques.

There are a couple of analyses and transformations not yet mentioned that we believe could be incorporated in our framework. First, type analysis is very useful for many program manipulations, e.g., for the incrementalization in Stage II. We could easily equip the transformations in Stage I and III with corresponding manipulations needed for types. Second, Stage III replaces irrelevant components with the constant _'s and performs a number of simplifications, where further manipulation with projections may help perform more simplifications like *component lifting*. For example, if we lift the single element in the second component of the second component of $\widehat{foo}$ and $\widehat{foo}'$ of Figure 4.2, and simplify the selection $1st(2nd(2nd(\hat{r})))$ in $\widehat{foo}'$ to be $2nd(2nd(\hat{r}))$, we obtain $\widehat{foo}$ as in Figure 4.2, and $\widehat{boo}$ and $\widehat{foo}'$ as follows:

$$\widehat{boo}(x) = \textbf{let } v_1 = foo(x-1) \textbf{ in } < v_1 + foo(x-2),\ v_1 >$$

$$\begin{aligned}\widehat{foo}'(x,\hat{r}) = \ & \textbf{if } x \le 1 \textbf{ then } < 1 > \\ & \textbf{else if } x = 2 \textbf{ then } < 3,\ < 2, 1 >> \\ & \textbf{else } < 1st(\hat{r}) + 1st(2nd(\hat{r})), \\ & \qquad < 1st(\hat{r}) + 2nd(2nd(\hat{r})),\ 1st(\hat{r}) >>\end{aligned}$$

Cost model and time/space trade-off. The basic motivation for caching is to trade space for speed. Ideally, we would have a cost model for time and a cost model for space, and decide what to cache depending on the trade-off between time and space required by the application. There are standard constructions for mechanical time analysis [Weg75,Ros89], though further study is needed; automatic space analysis and the trade-off between time and space are problems open for study.

This work assumes that there is unlimited space to be used for achieving the least asymptotic time possible. Thus, we cache only values of function applications, assuming other program constructs take constant time. For example, if the value of $f(x) + g(x)$ is needed in the incremental program, then we cache the values of $f(x)$ and $g(x)$ and compute the sum from the two cached values. Note that we assume that space is unlimited, not that it is free. Each of the three stages make an effort to reduce space consumption without adversely affecting asymptotic time performance.

One could be more mindful of economizing cache space by avoiding caching values of function applications unless they are absolutely needed. For example, if the value of $f(x) + g(x)$ is needed in the incremental program, but neither $f(x)$ nor $g(x)$ is needed separately, then we can cache just the value of $f(x) + g(x)$; coincidently, this also improves the speed of this example by a slight constant amount.

On the other hand, we could be more mindful of constant speed-up regardless of additional space consumption by caching the values of all program constructs, not just function applications. For example, we would cache the values of $f(x)$, $g(x)$, and $f(x) + g(x)$ respectively for their respective uses in the incremental program, thus saving the time to compute the sum, but consuming the space to store the sum.

Other choices of the time-space trade-off may also be required by applications, e.g., a fixed cache space for achieving the least running time possible. For some applications, we may need to consider the number of times a given value is needed.

4.6 Examples

4.6.1 Fibonacci function

Consider Fibonacci function fib of Figure 4.9 and the input change operation $x' = x \oplus y = x + 1$. Using the derivation approach in Chapter 3 directly, we can obtain

$$
\begin{array}{rl}
fib(x): & \text{compute Fibonacci number } x \\[4pt]
fib(x) = & \textbf{if } x \le 1 \textbf{ then } 1 \\
& \textbf{else } fib(x-1) + fib(x-2)
\end{array}
$$

Figure 4.9: Example function definition of fib

the function fib' below such that, if $fib(x) = r$, then $fib'(x, r) = fib(x + 1)$:

$$
\begin{array}{rl}
fib'(x,r) = & \textbf{if } x \le 0 \textbf{ then } 1 \\
& \textbf{else if } x = 1 \textbf{ then } 2 \\
& \textbf{else } r + fib(x-1)
\end{array}
\tag{4.26}
$$

But $fib'(x, r)$ takes time $O(2^x)$, no better than computing $fib(x + 1)$ from scratch. Now, we apply the cache-and-prune method, as follows:

I. We apply the optimized extension transformation $\mathcal{E}xt1$ and obtain a function $\overline{fib}_1$:

$$
\begin{array}{rl}
\overline{fib}_1(x) = & \textbf{if } x \le 1 \textbf{ then } <1, \_, \_> \\
& \textbf{else let } v_3 = \textbf{let } v_{31} = \textbf{let } v_1 = \overline{fib}_1(x-1) \textbf{ in} \\
& \qquad\qquad\qquad\qquad <1st(v_1), v_1> \qquad \textbf{in} \\
& \qquad\qquad\quad \textbf{let } v_{32} = \textbf{let } v_2 = \overline{fib}_1(x-2) \textbf{ in} \\
& \qquad\qquad\qquad\qquad <1st(v_2), v_2> \qquad \textbf{in} \\
& \qquad\qquad\quad <1st(v_{31}) + 1st(v_{32})> @rst(v_{31})@rst(v_{32}) \\
& \qquad <1st(v_3)> @ <> @ <rst(v_3)>
\end{array}
$$

We apply the cleaning transformation and obtain an extended function $\overline{fib}$:

$$
\begin{array}{rl}
\overline{fib}(x) = & \textbf{if } x \le 1 \textbf{ then } <1, \_, \_> \\
& \textbf{else let } v_1 = \overline{fib}(x-1) \textbf{ in} \\
& \qquad \textbf{let } v_2 = \overline{fib}(x-2) \textbf{ in} \\
& \qquad <1st(v_1) + 1st(v_2),\ v_1,\ v_2>
\end{array}
\tag{4.27}
$$

II. We derive an incremental version of $\overline{fib}$ under $\oplus$ using the approach in Chapter 3, i.e., we transform $\overline{fib}(x \oplus y) = \overline{fib}(x+1)$, with $\overline{fib}(x) = \bar{r}$:

1. unfold $\overline{fib}(x{+}1)$, simplify primitives

$$\begin{array}{ll} = & \textbf{if } x \le 0 \textbf{ then } <1,\_,\_> \\ & \textbf{else let } v_1 = \overline{fib}(x) \textbf{ in} \\ & \quad \textbf{let } v_2 = \overline{fib}(x-1) \textbf{ in} \\ & \quad <1st(v_1){+}1st(v_2),\ v_1,\ v_2> \end{array}$$

2. separate cases, replace applications of $\overline{fib}$

$$\begin{array}{ll} = & \textbf{if } x \le 0 \textbf{ then } <1,\_,\_> \\ & \textbf{else if } x = 1 \textbf{ then} \\ & \quad <2,\ <1,\_,\_>,\ <1,\_,\_>> \\ & \textbf{else let } v_1 = \bar{r} \textbf{ in} \\ & \quad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\ & \quad <1st(v_1){+}1st(v_2),\ v_1,\ v_2> \end{array}$$

and obtain function $\overline{fib}'$ such that, if $\overline{fib}(x) = \bar{r}$, then $\overline{fib}'(x, \bar{r}) = \overline{fib}(x+1)$:

$$\begin{array}{rl} \overline{fib}'(x,\bar{r}) = & \textbf{if } x \le 0 \textbf{ then } <1,\_,\_> \\ & \textbf{else if } x = 1 \textbf{ then } <2,\ <1,\_,\_>,\ <1,\_,\_>> \\ & \textbf{else } <1st(\bar{r}){+}1st(2nd(\bar{r})),\ \bar{r},\ 2nd(\bar{r})> \end{array} \tag{4.28}$$

III. Using dependency analysis for $\overline{fib}'$ in a similar way as for $\overline{foo}'$ but simpler, we obtain the closure projection $\{1^{st}, 1^{st}2^{nd}\}$. To prune, we first obtain:

$$\begin{array}{rl} \widehat{fib}_1(x) = & \textbf{if } x \le 1 \textbf{ then } <1,\_,\_> \\ & \textbf{else let } v_1 = \overline{fib}(x-1) \textbf{ in} \\ & \quad \textbf{let } v_2 = \overline{fib}(x-2) \textbf{ in} \\ & \quad <1st(v_1){+}1st(v_2),\ <1st(v_1),\_,\_>,\ \_> \end{array}$$

$$\begin{array}{rl} \widehat{fib}'_1(x,\hat{r}_1) = & \textbf{if } x \le 0 \textbf{ then } <1,\_,\_> \\ & \textbf{else if } x = 1 \textbf{ then } <2,\ <1,\_,\_>,\ <1,\_,\_>> \\ & \textbf{else } <1st(\bar{r}_1){+}1st(2nd(\bar{r}_1)),\ <1st(\bar{r}_1),\_,\_>,\ \_> \end{array}$$

Then, we simplify $\widehat{fib}_1$ and $\widehat{fib}'_1$, remove $\_$ components, and lift the single component in the second component of $\widehat{fib}_1$ and $\widehat{fib}'_1$ as discussed in Section 4.5. We obtain:

$$\begin{array}{rl} \widehat{fib}(x) = & \textbf{if } x \le 1 \textbf{ then } <1> \\ & \textbf{else let } u_1 = fib(x-1) \textbf{ in} \\ & \quad <u_1 + fib(x-2),\ u_1> \end{array} \tag{4.29}$$

$$\begin{array}{rl} \widehat{fib}'(x,\hat{r}) = & \textbf{if } x \le 0 \textbf{ then } <1> \\ & \textbf{else if } x = 1 \textbf{ then } <2,1> \\ & \textbf{else } <1st(\hat{r}) + 2nd(\hat{r}),\ 1st(\hat{r})> \end{array} \tag{4.30}$$

Clearly, $\widehat{fib}'(x,\hat{r})$ takes only time $O(1)$. Note that: $fib(x) = 1st(\widehat{fib}(x))$ and, if $\widehat{fib}(x) = \hat{r}$, then $\widehat{fib}'(x,\hat{r}) = \widehat{fib}(x+1)$. Using the definition of $\widehat{fib}'$ above in this last equation, we obtain a new definition for $\widehat{fib}$:

$$\begin{array}{rl} \widehat{fib}(x+1) = & \textbf{if } x \le 0 \textbf{ then } <1> \\ & \textbf{else if } x = 1 \textbf{ then } <2,1> \\ & \textbf{else let } \hat{r} = \widehat{fib}(x) \textbf{ in } <1st(\hat{r}){+}2nd(\hat{r}),\ 1st(\hat{r})> \end{array}$$

Letting $v = x + 1$, we get

$$\begin{array}{rl} \widehat{fib}(v) = & \textbf{if } v \le 1 \textbf{ then } <1> \\ & \textbf{else if } v = 2 \textbf{ then } <2,1> \\ & \textbf{else let } \hat{r} = \widehat{fib}(v-1) \textbf{ in } <1st(\hat{r}){+}2nd(\hat{r}),\ 1st(\hat{r})> \end{array} \tag{4.31}$$

We define $fib(v) = 1st(\widehat{fib}(v))$ using the definition of $\widehat{fib}$ in (4.31). Clearly, this computes the Fibonacci function in linear time, as desired.

4.6.2 Merge sort

Consider the merge sort function *sort* of Figure 3.8 and input change operation $x \oplus y = cons(y, x)$.

I. We cache all intermediate results of *sort* and obtain the extended functions:

$$\begin{array}{l}
\overline{sort}(x)\\
= \textbf{if } null(x) \textbf{ then}\\
\qquad < nil, _, _, _, _, _ >\\
\quad \textbf{else if } null(cdr(x)) \textbf{ then}\\
\qquad < x, _, _, _, _, _ >\\
\quad \textbf{else let } v_{11} = \overline{odd}(x) \textbf{ in}\\
\qquad \textbf{let } u_1 = \overline{sort}(1st(v_{11})) \textbf{ in}\\
\qquad \textbf{let } v_{21} = \overline{even}(x) \textbf{ in}\\
\qquad \textbf{let } u_2 = \overline{sort}(1st(v_{21})) \textbf{ in}\\
\qquad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in}\\
\qquad < 1st(v), v_{11}, u_1, v_{21}, u_2, v >
\end{array}$$

$$\begin{array}{lcl}
\overline{odd}(x) & = & \textbf{if } null(x) \textbf{ then } < nil, _ >\\
 & & \textbf{else let } v_1 = \overline{even}(cdr(x)) \textbf{ in}\\
 & & \qquad < cons(car(x), 1st(v_1)),\ v_1 >\\
\overline{even}(x) & = & \textbf{if } null(x) \textbf{ then } < nil, _ >\\
 & & \textbf{else let } v_1 = \overline{odd}(cdr(x)) \textbf{ in}\\
 & & \qquad < 1st(v_1),\ v_1 >\\
\overline{merge}(x, y) & & \\
 & = & \textbf{if } null(x) \textbf{ then } < y, _, _ >\\
 & & \textbf{else if } null(y) \textbf{ then } < x, _, _ >\\
 & & \textbf{else if } car(x) \le car(y) \textbf{ then}\\
 & & \qquad \textbf{let } v_1 = \overline{merge}(cdr(x), y) \textbf{ in}\\
 & & \qquad < cons(car(x), 1st(v_1)),\ v_1, _ >\\
 & & \textbf{else let } v_2 = \overline{merge}(x, cdr(y)) \textbf{ in}\\
 & & \qquad < cons(car(y), 1st(v_2)),\ _, v_2 >
\end{array} \tag{4.32}$$

II. We derive an incremental version of $\overline{sort}$ under $\oplus$ using the approach in Chapter 3, i.e., we transform $\overline{sort}(x \oplus y) = \overline{sort}(cons(y, x))$, with $\overline{sort}(x) = \bar{r}$:

1. unfold $\overline{sort}(cons(y, x))$, simplify

$$\begin{array}{l}
= \textbf{if } null(x) \textbf{ then}\\
\qquad < cons(y, nil), _, _, _, _, _ >\\
\quad \textbf{else let } v_1 = \overline{even}(x) \textbf{ in}\\
\qquad \textbf{let } v_{11} = < cons(y, 1st(v_1)),\\
\qquad\qquad\qquad v_1 > \textbf{ in}\\
\qquad \textbf{let } u_1 = \overline{sort}(1st(v_{11})) \textbf{ in}\\
\qquad \textbf{let } v_2 = \overline{odd}(x) \textbf{ in}\\
\qquad \textbf{let } v_{21} = < 1st(v_2), v_2 > \textbf{ in}\\
\qquad \textbf{let } u_2 = \overline{sort}(1st(v_{21})) \textbf{ in}\\
\qquad \textbf{let } v = \overline{merge}(1st(u_1),\\
\qquad\qquad\qquad 1st(u_2)) \textbf{ in}\\
\qquad < 1st(v), v_{11}, u_1, v_{21}, u_2, v >
\end{array}$$

2. separate cases, replace applications of $\overline{sort}$

$$\begin{array}{l}
= \textbf{if } null(1st(\bar{r})) \textbf{ then}\\
\qquad < cons(y, nil), _, _, _, _, _ >\\
\quad \textbf{else if } null(cdr(1st(\bar{r}))) \textbf{ then}\\
\qquad \textbf{let } v_{11} = < cons(y, nil), < nil, < nil >>> \textbf{ in}\\
\qquad \textbf{let } v_{21} = < 1st(\bar{r}), < 1st(\bar{r}), < nil >>> \textbf{ in}\\
\qquad \textbf{let } v = \overline{merge}(cons(y, nil), 1st(\bar{r})) \textbf{ in}\\
\qquad < 1st(v), v_{11}, < cons(y, nil) >, v_{21}, < 1st(\bar{r}) >,\\
\qquad\quad v >\\
\quad \textbf{else let } v_1 = 4th(\bar{r}) \textbf{ in}\\
\qquad \textbf{let } v_{11} = < cons(y, 1st(v_1)), v_1 > \textbf{ in}\\
\qquad \textbf{let } u_1 = \overline{sort}'(y, 1st(v_1), 5th(\bar{r})) \textbf{ in}\\
\qquad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in}\\
\qquad \textbf{let } v_{21} = < 1st(v_2), v_2 > \textbf{ in}\\
\qquad \textbf{let } u_2 = 3rd(\bar{r}) \textbf{ in}\\
\qquad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in}\\
\qquad < 1st(v), v_{11}, u_1, v_{21}, u_2, v >
\end{array}$$

and obtain function $\overline{sort}'$ below such that, if $\overline{sort}(x) = \bar{r}$, then $\overline{sort}'(y,\bar{r}) = \overline{sort}(cons(y,x))$:

$$
\begin{array}{rl}
\overline{sort}'(y,\bar{r}) = & \textbf{if } null(1st(\bar{r})) \textbf{ then} \\
& \quad < cons(y,nil), \_, \_, \_, \_, \_ > \\
& \textbf{else if } null(cdr(1st(\bar{r}))) \textbf{ then} \\
& \quad \textbf{let } v_{11} = < cons(y,nil), \ < nil, < nil >>> \ \textbf{in} \\
& \quad \textbf{let } v_{21} = < 1st(\bar{r}), \ < 1st(\bar{r}), < nil >>> \ \textbf{in} \\
& \quad \textbf{let } v = \overline{merge}(cons(y,nil), 1st(\bar{r})) \textbf{ in} \\
& \quad < 1st(v), \ v_{11}, < cons(y,nil) >, \ v_{21}, < 1st(\bar{r}) >, \ v > \\
& \textbf{else let } v_1 = 4th(\bar{r}) \textbf{ in} \\
& \quad \textbf{let } v_{112} = < cons(y, 1st(v_1)), v_1 > \ \textbf{in} \\
& \quad \textbf{let } u_1 = \overline{sort}'(y, 5th(\bar{r})) \textbf{ in} \\
& \quad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\
& \quad \textbf{let } v_{212} = < 1st(v_2), v_2 > \ \textbf{in} \\
& \quad \textbf{let } u_2 = 3rd(\bar{r}) \textbf{ in} \\
& \quad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in} \\
& \quad < 1st(v), v_{112}, u_1, v_{212}, u_2, v >
\end{array}
\tag{4.33}
$$

III. First, using the dependency analysis, for $\Pi \neq ABS$, we have

$$
\begin{array}{rcl}
\overline{sort}'^2\Pi & = & (null(1st(\bar{r})))^{\bar{r}}\, ID \ \cup \ ABS \ \cup \\
& & (null(cdr(1st(\bar{r}))))^{\bar{r}}\, ID \ \cup (1st(\bar{r}))^{\bar{r}}(\overline{merge}^2\{1^{st}\}) \cup \\
& & (4th(\bar{r}))^{\bar{r}}((1st(v_1))^{v_1}\Pi_{(2)(1)} \ \cup \ \Pi_{(2)(2)}) \cup \\
& & (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^2((1st(u_1))^{u_1}(\overline{merge}^1((1st(v))^{v}\,\Pi_{(1)} \ \cup \ \Pi_{(6)})) \ \cup \ \Pi_{(3)})) \cup \\
& & (2nd(\bar{r}))^{\bar{r}}((1st(v_2))^{v_2}\Pi_{(4)(1)} \ \cup \ \Pi_{(4)(2)}) \cup \\
& & (3rd(\bar{r}))^{\bar{r}}((1st(u_2))^{u_2}(\overline{merge}^2((1st(v))^{v}\,\Pi_{(1)} \ \cup \ \Pi_{(6)})) \ \cup \ \Pi_{(5)})
\end{array}
$$

which is recursively defined, and can be simplified for $1^{st} \in \Pi$. With $\Pi_{(1)} = ID$ and $\overline{merge}^1\{1^{st}\} = \overline{merge}^2\{1^{st}\} = ID$, we have

$$
\begin{array}{rcl}
\overline{sort}'^{2(0)}\Pi & = & ABS \\
\overline{sort}'^{2(i+1)}\Pi & = & \{1^{st}\} \cup \\
& & (4th(\bar{r}))^{\bar{r}}((1st(v_1))^{v_1}\Pi_{(2)(1)} \ \cup \ \Pi_{(2)(2)}) \cup \\
& & (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}(\{1^{st}\} \ \cup \ \Pi_{(3)})) \cup \\
& & (2nd(\bar{r}))^{\bar{r}}((1st(v_2))^{v_2}\Pi_{(4)(1)} \ \cup \ \Pi_{(4)(2)}) \cup \\
& & (3rd(\bar{r}))^{\bar{r}}(\{1^{st}\} \ \cup \ \Pi_{(5)})
\end{array}
\tag{4.34}
$$

Limiting the depth of selection functions to be 1, we compute the closure of the transitive dependencies for $\overline{sort}'^2$ and obtain:

$$
\begin{array}{lll}
\Pi^{(0)} = \{1^{st}\} & \text{and, by (4.34),} & \overline{sort}'^{2(i+1)}\Pi^{(0)} = \{1^{st}\} \cup (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}\{1^{st}\}) \cup \{3^{rd}\} \\
 & & \overline{sort}'^{2(0)}\Pi^{(0)} = ABS \\
 & & \overline{sort}'^{2(1)}\Pi^{(0)} = \{1^{st}, 3^{rd}\} \\
 & & \overline{sort}'^{2(2)}\Pi^{(0)} = \{1^{st}, 3^{rd}, 5^{th}\} \\
 & & \overline{sort}'^{2(3)}\Pi^{(0)} = \{1^{st}, 3^{rd}, 5^{th}\} \\
\Pi^{(1)} = \{1^{st}, 3^{rd}, 5^{th}\} & \text{and, by (4.34),} & \overline{sort}'^{2(i+1)}\Pi^{(1)} = \{1^{st}\} \cup (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}ID) \cup \{3^{rd}\} \\
 & & \overline{sort}'^{2(i+1)}ID = \{1^{st}\} \cup \{4^{th}\} \cup (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}ID) \cup \{2^{nd}\} \cup \{3^{rd}\} \\
 & & \overline{sort}'^{2(0)}\Pi^{(1)} = ABS \\
 & & \overline{sort}'^{2(1)}\Pi^{(1)} = \{1^{st}, 3^{rd}\} \\
 & & \overline{sort}'^{2(2)}\Pi^{(1)} = \{1^{st}, 3^{rd}, 5^{th}\} \\
 & & \overline{sort}'^{2(3)}\Pi^{(1)} = \{1^{st}, 3^{rd}, 5^{th}\} \\
\Pi^{(2)} = \{1^{st}, 3^{rd}, 5^{th}\} & &
\end{array}
$$

Thus, we get the closure projection $\{1^{st}, 3^{rd}, 5^{th}\}$. Actually, for this example, we could directly see that

$$\overline{sort}'^2\{1^{st}, 3^{rd}, 5^{th}\} \;=\; \{1^{st}\} \;\cup\; (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}ID) \;\cup\; (3rd(\bar{r}))^{\bar{r}}ID \;\subseteq\; \{1^{st}, 5^{th}, 3^{rd}\}$$

which matches the intuition that the first component of $\overline{sort}'$ depends only on $3rd(\bar{r})$ and $5th(\bar{r})$, and the third and fifth components depend on $3rd(\bar{r})$ and $5th(\bar{r})$ too. Now, we prune functions $\overline{sort}$ and $\overline{sort}'$, and we obtain

$$
\begin{array}{ll}
\widehat{sort}_1(x) = & \textbf{if } null(x) \textbf{ then} \\
 & \quad < nil, \_, \_, \_, \_, \_ > \\
 & \textbf{else if } null(cdr(x)) \textbf{ then} \\
 & \quad < x, \_, \_, \_, \_, \_ > \\
 & \textbf{else let } u_1 = \widehat{sort}(odd(x)) \textbf{ in} \\
 & \quad \textbf{let } u_2 = \widehat{sort}(even(x)) \textbf{ in} \\
 & \quad < merge(1st(u_1), 1st(u_2)), \_, u_1, \_, u_2, \_ >
\end{array}
$$

$$
\begin{array}{ll}
\widehat{sort}'_1(y, \widehat{r}_1) = & \textbf{if } null(1st(\widehat{r}_1)) \textbf{ then} \\
 & \quad < cons(y, nil), \_, \_, \_, \_, \_ > \\
 & \textbf{else if } null(cdr(1st(\widehat{r}_1))) \textbf{ then} \\
 & \quad < merge(cons(y, nil), 1st(\widehat{r}_1)), \_, < cons(y, nil) >, \_, < 1st(\widehat{r}_1) >, \_ > \\
 & \textbf{else let } u_1 = \widehat{sort}'(y, 5th(\widehat{r}_1)) \textbf{ in} \\
 & \quad \textbf{let } u_2 = 3rd(\widehat{r}_1) \textbf{ in} \\
 & \quad < merge(1st(u_1), 1st(u_2)), \_, u_1, \_, u_2, \_ >
\end{array}
$$

Finally, we eliminate _ components, adjust the indexing, and obtain

$$
\begin{array}{ll}
\widehat{sort}(x) = & \textbf{if } null(x) \textbf{ then } < nil > \\
 & \textbf{else if } null(cdr(x)) \textbf{ then } < x > \\
 & \textbf{else let } u_1 = \widehat{sort}(odd(x)) \textbf{ in} \\
 & \quad \textbf{let } u_2 = \widehat{sort}(even(x)) \textbf{ in} \\
 & \quad < merge(1st(u_1), 1st(u_2)),\; u_1,\; u_2 >
\end{array}
\tag{4.35}
$$

$$
\begin{array}{rl}
\widehat{sort}'(y, \hat{r}) = & \textbf{if } null(1st(\hat{r})) \textbf{ then } < cons(y, nil) > \\
& \textbf{else if } null(cdr(1st(\hat{r}))) \textbf{ then} \\
& \quad < merge(cons(y, nil), 1st(\hat{r})),\ < cons(y, nil) >,\ < 1st(\hat{r}) >> \\
& \textbf{else let } u_1 = \widehat{sort}'(y, 3rd(\hat{r})) \textbf{ in} \\
& \quad \textbf{let } u_2 = 2nd(\hat{r}) \textbf{ in} \\
& \quad < merge(1st(u_1), 1st(u_2)),\ u_1,\ u_2 >
\end{array}
\tag{4.36}
$$

For x of length n, merge sort $sort$ takes $O(n \log n)$ time. Incremental merge sort $\widehat{sort}'$ takes only $O(n)$ time, although it uses $O(n \log n)$ space to store the intermediate results of the previous sort.

4.6.3 Attribute evaluation using Katayama functions

Given an attribute grammar, a set of recursive functions can be constructed to evaluate the attribute values for any derivation tree of the grammar [Kat84]. Basically, each function evaluates a synthesized attribute of a non-terminal, and the value of a synthesized attribute of the root symbol is the final return value of interest. Thus, for the given grammar, the set of recursive functions takes a derivation tree of the grammar as input, and returns the value of a synthesized attribute at the root as output.

We consider subtree replacement as the input change operation, given by a new subtree and a path from the root of the whole tree to the root of the subtree to be replaced.

First, caching all intermediate results leads to a set of extended recursive functions that returns an attributed tree instead of just the value of an synthesized attribute at the root. Then, incrementalizing the set of extended functions under a subtree replacement is just composing a new attributed tree from the old one, evaluating only attributes whose values are affected by the subtree replacement, yielding a set of incremental recursive functions.

Suppose a given batch attribute evaluation program evaluates each attribute only once, then the derived incremental program computes in $O(|\text{PATH}| + |\text{AFFECTED}|)$ time, where PATH is the path from the root of the whole tree to the root of the new subtree, and AFFECTED is the set of attributes whose values are different in the new tree than in the old after the subtree replacement [RTD83].

4.6.4 Local neighborhood problems in image processing

We have mentioned that the general principles underlying our approach also apply to other languages. This section gives an example where the cache-and-prune method is used to improve imperative programs with arrays.

In image processing, computing information about local neighborhoods is common [Wel86,Web92,Zab94,ZW94]. A simple but typical example is the local summation problem [Wel86,Zab94]: given an n-by-n image, compute, for each pixel, the local sum of its m-by-m neighborhood. The straightforward naive algorithm, given in Figure 4.10(a), takes $O(n^2m^2)$ time, while an efficient algorithm using dynamic programming, given in Figure 4.10(b), takes $O(n^2)$ time. We show how to obtain the efficient algorithm systematically following our approach. For simplicity, initializations for the array margins are ignored.

(a)

```
for i := 0 to n do
  for j := 0 to n do
    sum := 0;
    for k := 0 to m do
      for l := 0 to m do
        sum := sum + a[i+k, j+l];
    b[i, j] := sum
```

(b)

```
for i := 0 to n do
  for j := 0 to n do
    c[i+m, j] := c[i+m, j-1]-a[i+m, j-1]
                                +a[i+m, j+m];
    b[i, j] := b[i-1, j] - c[i-1, j] + c[i+m, j]
```

Figure 4.10: Programs for local summation problem

Inner loop. To improve the naive algorithm, call it f, we first improve the inner loop, call it g, with any fixed index i for the outer loop. We will obtain an incremental program for the program g_1, defined below, under input change operation $j_1' = j_1+1$, and uses the incremental program for each iteration of g.

```
for j := 0 to j1 do
  sum := 0;
  for k := 0 to m do
    for l := 0 to m do
      sum := sum + a[i+k, j+l];
  b[i, j] := sum
```

I. Caching all non-trivial intermediate results, we obtain the program $\bar{g}_1$ as follows:

1. save results of the innermost loop

```
for j := 0 to j1 do
  sum := 0;
  for k := 0 to m do
    row := 0;
    for l := 0 to m do
      row := row + a[i+k, j+l];
      sum := sum + a[i+k, j+l];
    c[i, j, k] := row;
  b[i, j] := sum
```

2. eliminate common computation

```
for j := 0 to j1 do
  sum := 0;
  for k := 0 to m do
    row := 0;
    for l := 0 to m do
      row := row + a[i+k, j+l];
    c[i, j, k] := row;
    sum := sum + c[i, j, k];
  b[i, j] := sum
```

II. Incrementalizing $\bar{g_1}$ on new input $j_1{+}1$, we obtain the program $\bar{g_1}'$ as follows:

1. unfold $\bar{g_1}$ on new index $j_1{+}1$, and save computing iterations 1 to j_1

$$
\begin{array}{l}
sum := 0; \\
\textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\quad row := 0; \\
\quad \textbf{for } l := 0 \textbf{ to } m \textbf{ do} \\
\quad\quad row := row + a[i{+}k, j_1{+}1{+}l]; \\
\quad c[i, j_1{+}1, k] := row; \\
\quad sum := sum + c[i, j_1{+}1, k]; \\
b[i, j_1{+}1] := sum
\end{array}
$$

2. replace $c[i, j_1{+}1, k]$ by $c[i, j_1, k] - a[i{+}k, j_1] + a[i{+}k, j_1{+}1{+}m]$

$$
\begin{array}{l}
sum := 0; \\
\textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\quad row := 0; \\
\quad \textbf{for } l := 0 \textbf{ to } m \textbf{ do} \\
\quad\quad row := row + a[i{+}k, j_1{+}1{+}l]; \\
\quad c[i, j_1{+}1, k] := c[i, j_1, k] - a[i{+}k, j_1] \\
\quad\quad\quad\quad +a[i{+}k, j_1{+}1{+}m]; \\
\quad sum := sum + c[i, j_1{+}1, k]; \\
b[i, j_1{+}1] := sum
\end{array}
$$

III. Pruning the program $\bar{g_1}'$, we obtain the program $\hat{g_1}'$:

$$
\begin{array}{l}
sum := 0; \\
\textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\quad c[i, j_1{+}1, k] := c[i, j_1, k] - a[i{+}k, j_1] + a[i{+}k, j_1{+}1{+}m]; \\
\quad sum := sum + c[i, j_1{+}1, k]; \\
b[i, j_1{+}1] := sum
\end{array}
$$

Using $\hat{g_1}'$ for each iteration of g, we replace $j_1{+}1$ by j (and thus j_1 by $j{-}1$) and add the outside loop for j to go from 0 to n. We obtain an optimized version of g:

$$
\begin{array}{l}
\textbf{for } j := 0 \textbf{ to } n \textbf{ do} \\
\quad sum := 0; \\
\quad \textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\quad\quad c[i, j, k] := c[i, j{-}1, k] - a[i{+}k, j{-}1] + a[i{+}k, j{+}m]; \\
\quad\quad sum := sum + c[i, j, k]; \\
\quad b[i, j] := sum
\end{array}
\tag{4.37}
$$

Using the optimized version of g in f, we obtain an optimized program f:

$$
\begin{array}{l}
\textbf{for } i := 0 \textbf{ to } n \textbf{ do} \\
\quad \textbf{for } j := 0 \textbf{ to } n \textbf{ do} \\
\quad\quad sum := 0; \\
\quad\quad \textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\quad\quad\quad c[i, j, k] := c[i, j{-}1, k] - a[i{+}k, j{-}1] + a[i{+}k, j{+}m]; \\
\quad\quad\quad sum := sum + c[i, j, k]; \\
\quad\quad b[i, j] := sum
\end{array}
\tag{4.38}
$$

Outer loop. We then improve the outer loop of the new program f in (4.38). We will obtain an incremental program for the program f_1, defined below, under input change operation $i_1' = i_1{+}1$, and uses the incremental program for each iteration of the new f.

$$
\begin{array}{l}
\textbf{for } i := 0 \textbf{ to } i_1 \textbf{ do} \\
\quad \textbf{for } j := 0 \textbf{ to } n \textbf{ do} \\
\quad\quad sum := 0; \\
\quad\quad \textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\quad\quad\quad c[i, j, k] := c[i, j{-}1, k] - a[i{+}k, j{-}1] + a[i{+}k, j{+}m]; \\
\quad\quad\quad sum := sum + c[i, j, k]; \\
\quad\quad b[i, j] := sum
\end{array}
$$

I. Caching all intermediate results, we obtain the program $\bar{f}_1$ same as f_1, since all intermediate results have already been returned.

II. Incrementalizing $\bar{f}_1$ on new input $i_1{+}1$, we obtain the program $\bar{f}_1{}'$ as follow:

1. unfold $\bar{f}_1$ on new index $i_1{+}1$, and save computing iterations 1 to i_1

$$
\begin{array}{l}
\textbf{for } j := 0 \textbf{ to } n \textbf{ do} \\
\quad sum := 0; \\
\quad \textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\qquad c[i_1{+}1, j, k] := c[i_1{+}1, j{-}1, k] \\
\qquad\qquad\qquad\qquad -a[i_1{+}1{+}k, j{-}1] \\
\qquad\qquad\qquad\qquad +a[i_1{+}1{+}k, j{+}m]; \\
\qquad sum := sum + c[i_1{+}1, j, k]; \\
\quad b[i_1{+}1, j] := sum
\end{array}
$$

2. replace $b[i_1{+}1, j]$ by $b[i_1, j] - c[i_1, j, 0] + c[i_1{+}1{+}m, j, 0]$

$$
\begin{array}{l}
\textbf{for } j := 0 \textbf{ to } n \textbf{ do} \\
\quad sum := 0; \\
\quad \textbf{for } k := 0 \textbf{ to } m \textbf{ do} \\
\qquad c[i_1{+}1, j, k] := c[i_1{+}1, j{-}1, k] \\
\qquad\qquad\qquad\qquad -a[i_1{+}1{+}k, j{-}1] \\
\qquad\qquad\qquad\qquad +a[i_1{+}1{+}k, j{+}m]; \\
\qquad sum := sum + c[i_1{+}1, j, k]; \\
\quad b[i_1{+}1, j] := b[i_1, j] - c[i_1, j, 0] \\
\qquad\qquad\qquad +c[i_1{+}1{+}m, j, 0]
\end{array}
$$

III. Pruning the program $\bar{f}_1{}'$, we obtain the program $\hat{f}_1{}'$ as follows:

1. prune unneeded code

$$
\begin{array}{l}
\textbf{for } j := 0 \textbf{ to } n \textbf{ do} \\
\quad c[i_1{+}1{+}m, j, 0] := c[i_1{+}1{+}m, j{-}1, 0] \\
\qquad\qquad\qquad\qquad -a[i_1{+}1{+}m, j{-}1] \\
\qquad\qquad\qquad\qquad +a[i_1{+}1{+}m, j{+}m]; \\
\quad b[i_1{+}1, j] := b[i_1, j] - c[i_1, j, 0] \\
\qquad\qquad\qquad +c[i_1{+}1{+}m, j, 0]
\end{array}
$$

2. prune unneeded array dimension

$$
\begin{array}{l}
\textbf{for } j := 0 \textbf{ to } n \textbf{ do} \\
\quad c[i_1{+}1{+}m, j] := c[i_1{+}1{+}m, j{-}1] \\
\qquad\qquad\qquad\qquad -a[i_1{+}1{+}m, j{-}1] \\
\qquad\qquad\qquad\qquad +a[i_1{+}1{+}m, j{+}m]; \\
\quad b[i_1{+}1, j] := b[i_1, j] - c[i_1, j] \\
\qquad\qquad\qquad +c[i_1{+}1{+}m, j]
\end{array}
$$

Using $\hat{f}_1{}'$ for each iteration of f, we replacing $i_1{+}1$ by i (and thus i_1 by $i{-}1$) and add the outside loop for i to go from 0 to n. We obtain the final optimized version in Figure 4.10(b). Only four $\pm$ operations are needed for each pixel, no matter how large m is. Thus, the whole program takes only $O(n^2)$ time.

4.7 Related work

The cache-and-prune method uses a number of program analysis and transformation techniques that have been summarized in Section 4.5. Here we compare our work with related work in program improvement using caching techniques. Caching has been the basis of many techniques for developing efficient programs and optimizing programs. Bird [Bir80] and Cohen [Coh83] provide nice overviews. Most of these techniques fall into one of the following three classes.

Separate caching. In the first class, a global cache separate from a subject program is employed to record values of subcomputations that may be needed later, and certain strategies are chosen for using and managing the cache. We call this technique *separate caching*. It corresponds to the "exact tabulation" in [Bir80] and the "large-table method" in [Coh83]. The initial idea of memoization, "memo" functions, proposed by Michie [Mic68], belongs to this class. Thus, some uses of the word "memoization" mainly refers to techniques in this class [Par90].

In recent years, there has been additional work on general strategies for separate caching. For example, Hughes [Hug85] discusses lazy memo-functions that are suitable for use in systems with lazy evaluation. Mostow and Cohen [MC85] discuss some issues for speeding up Interlisp programs by caching in the presence of side effects. Pugh [Pug88a] provides some improved cache replacement strategies for a simple functional language. Two trends seem obvious: studying *specialized* cache strategies for classes of problems, and adding *annotations* or certain *specifications* to subject programs that provide hints to the cache strategies. An example of the former is the stable decomposition scheme proposed by Pugh and Teitelbaum [PT89], who also advocate a closer study of using memoization for incremental evaluation. Examples of the latter include work by Keller and Sleep [KS86], which proposes annotating applicative languages, work by Sundaresh and Hudak [SH91,Sun91], which decides what to cache based on given input partitions of programs, and work by Hoover [Hoo92], which proposes annotating an imperative language.

The pros and cons of separate caching are well discussed by Bird [Bir80] and Cohen [Coh83]. To summarize, the idea is simple, and the subject programs are basically unchanged. But the caching methods are dynamic, and thus are fundamentally interpretive. Moreover, the strategies for the use and management of the separate cache are hard to be both general and powerful at the same time, and therefore are sources of inefficiency.

Schema-based integrated caching. In the second and third classes, the above drawbacks are overcome by transforming subject programs to integrate caching into the transformed programs. Techniques in the second class apply transformations based on special properties and schemas of subject programs. We call this *schema-based integrated caching*. A nice survey of most of these ideas can be found in [Bir80], following which some uses of the word "tabulation" mainly refer to techniques in this class [Par90]. Typical examples of these techniques are dynamic programming [AHU74], schemas of redundancies [Coh83], and tupling [Pet84,Pet87,Chi93,CK93]. Dynamic programming applies to problems that can be divided into subproblems and solved from small subproblems to larger ones by storing and using results of smaller ones. Work on schemas of redundancies studies several forms of redundant recursive calls and their mathematical properties and provides transformations to eliminate them. Tupling looks for a recurrent pattern in computing intermediate results, groups those computed in the pattern into a tuple, and transforms the program to compute the tuple progressively.

Note that separate caching with a specialized cache strategy for a certain class of problems can be used for schema-based integrated caching for this class of problems. More precisely, for any problem that fits into this class, we treat the corresponding program as fitting into a certain schema. We can then integrate the specialized cache strategy by transforming the corresponding program and obtain a transformed program with schema-based integrated caching. In this case, the separate caching corresponds to an interpretive mechanism, the transformation with integration is like compiling, and a transformed program corresponds to a compiled program.

While integrating caching into transformed programs eliminates the interpretive overhead of separate caching, a drawback of schema-based integrated caching is its lack of generality.

Principle-based integrated caching. Techniques in the third class analyze and transform programs according to general principles. We call this *principle-based integrated caching*. Often, such principles are used to derive a relatively complete set of strategies and rules for programs written in a certain language, and these strategies and rules are used to transform programs. For example, the conventional optimizing-compiler technique of strength reduction [All69,CK77,ACK81] identifies subcomputations like multiplications that can be replaced with subcomputations like additions while maintaining the values of these subcomputations. Similarly, the APTS program transformation system [Pai83,Pai90,Pai94] identifies set expressions in SETL that can be maintained using finite differencing rules [PK82].

Sometimes it is not sufficient to have only a fixed set of strategies and rules. Seeking more flexibility and broader applicability, KIDS [Smi90] advocates certain high-level strategies but leaves the choice of which intermediate results to maintain to manual decisions. CIP [Par90] also proposes a general strategy for caching, but it may even lead to less efficient programs. Recently, certain principles that can directly guide program transformations have been proposed. Webber's *principle of least computation* [Web93,Web95] avoids subcomputations whose values have been computed before or are not needed. Basically, first-order purely functional programs are transformed into *trace grammars*, which are *thinned* using this principle and then transformed back. This approach can not reduce the strength of primitive operators or use any auxiliary information. The heavy inference engine for thinning leads to some clever optimizations but is computationally exorbitant. Hall's *principle of redistributing intermediate results* [Hal90,Hal91] finds paths from subcomputations to multiple uses of their values. However, the method uses a great deal of program design knowledge, including annotations of invariances, test-case inputs, and proofs of correctness. Also, it guarantees correctness of the transformed programs only on the test-case inputs.

Our approach to the problem of program improvement via caching is a principled approach that integrates caching into the transformed programs. The intrinsic iterative computation property of programs drives the incremental computation of each iteration, which in turn drives the decision of what intermediate results to cache. The approach is a crucial complement to any incremental computation technique for achieving the goal of program improvement.

Our approach is not limited to using a fixed set of rules for program analysis and transformation. On the contrary, we can even use the approach to derive such rules when necessary. Compared to the general approaches advocated by KIDS or CIP, our approach is more algorithmic and automatable.

Chapter 5

Discovering auxiliary information

Efficient incremental computation of $f_0(x \oplus y)$ may sometimes require *auxiliary information* about x other than the intermediate results computed by $f_0(x)$. This information needs to be maintained efficiently as well.

Some approaches to incremental computation have exploited specific kinds of auxiliary information, e.g., auxiliary arithmetic associated with some classical strength-reduction rules [ACK81], dynamic mappings maintained by finite differencing rules for aggregate primitives in SETL [PK82] and INC [YS91], and auxiliary data structures for problems with certain properties like stable decomposition [PT89]. However, until now, systematic discovery of auxiliary information for arbitrary programs has not been studied.

This chapter presents a two-phase method that discovers a general class of auxiliary information for any incremental computation problem. Phase A transforms $f(x \oplus y)$ to expose *candidate auxiliary information* — subcomputations of $f(x \oplus y)$ that depend only on x and whose values are not embedded in the return value or intermediate results of $f(x)$. Phase B determines the usage of the candidate auxiliary information for the incremental computation — merges this information with f, derives an incremental version of the resulting program, and prunes out the part of the information that is not useful for the incremental computation.

All the program analyses and transformations are combined with considerations for caching intermediate results. We obtain a systematic approach that transforms non-incremental programs into efficient incremental programs that use and maintain useful auxiliary information as well as useful intermediate results. The use of auxiliary information allows us to achieve a greater degree of incrementality than otherwise possible. The approach can be applied directly to incrementalization problems for interactive systems, strength reduction in optimizing compilers, and finite differencing in transformational programming. We give examples for these program improvements that have applications in list processing, VLSI design, and graph algorithms.

Defining the Problem. We use an asymptotic time model and an unlimited space model as in Chapter 4. Again, our primary goal is to improve the asymptotic running time of the incremental computation, and we attempt to save space by maintaining only information useful for achieving this.

Given a program f_0 and an input change operation $\oplus$, we aim to extend f_0 to compute appropriate auxiliary information and obtain an incremental program that computes $f_0(x \oplus y)$ using this information and maintains the corresponding information. For example, consider function cmp of Figure 5.1 and input change operation $x' = x \oplus y = cons(y, x)$. Using the approach in Chapter 3, the function cmp'

$cmp(x)$: compare sum of odd and product of even positions of list x

$$cmp(x) = sum(odd(x)) \leq prod(even(x))$$

$$\begin{aligned} odd(x) &= \textbf{if } null(x) \textbf{ then } nil \\ &\quad\ \textbf{else } cons(car(x), even(cdr(x))) \\ even(x) &= \textbf{if } null(x) \textbf{ then } nil \\ &\quad\ \textbf{else } odd(cdr(x)) \end{aligned} \qquad \begin{aligned} sum(x) &= \textbf{if } null(x) \textbf{ then } 0 \\ &\quad\ \textbf{else } car(x) + sum(cdr(x)) \\ prod(x) &= \textbf{if } null(x) \textbf{ then } 1 \\ &\quad\ \textbf{else } car(x) * prod(cdr(x)) \end{aligned}$$

Figure 5.1: Example function definitions of cmp, odd, $even$, sum, and $prod$

---

of Figure 5.2 can be derived; using the method in Chapter 4, the functions $\widehat{cmp}$ and $\widehat{cmp}'$ can be obtained. But none of them is much faster than computing from scratch. For this example, auxiliary information needs to be used to compute $f_0(x \oplus y)$ quickly. In particular, the values of $sum(even(x))$ and $prod(odd(x))$ are crucial for computing $cmp(cons(y, x))$ incrementally but are not computed in $cmp(x)$ at all. We can compute these two pieces of auxiliary information, use them in computing $cmp(cons(y, x))$, and maintain them as well.

For typographical convenience, $\check{f}_0$ shall denote the function that returns all candidate auxiliary information for computing f_0 under $\oplus$. We shall use $\bar{\check{f}}_0$ to denote a function that returns both all intermediate results and all candidate auxiliary information of f_0, $\bar{\check{r}}$ the cached result of $\bar{\check{f}}_0(x)$, and $\bar{\check{f}}_0'$ an incremental version of $\bar{\check{f}}_0$ under $\oplus$. Similarly, $\widetilde{f_0}$ shall denote the pruned function that returns only the useful intermediate results and auxiliary information, $\tilde{r}$ the cached result of $\widetilde{f_0}(x)$, and $\widetilde{f_0}'$ a function that incrementally maintains the useful intermediate results and auxiliary information.

We use function cmp of Figure 5.1 and input change operation $x \oplus y = cons(y, x)$ as a running example. At the end, we obtain the functions $\widetilde{cmp}$ and $\widetilde{cmp}'$ shown in Figure 5.2. In particular, $\widetilde{cmp}'$ computes incrementally using $O(1)$ time.

5.1 Phase A: Discovering candidate auxiliary information

Auxiliary information is, by definition, useful information not computed by the original program f_0, so it can not be obtained directly from f_0. However, auxiliary information is information depending only on x that can speed up the computation of $f_0(x \oplus y)$. Seeking to obtain such information systematically, we come to the idea that when computing $f_0(x \oplus y)$, for example in the manner of $f_0'(x, y, r)$, there are

$cmp'(x,y) = cmp(cons(y,x))$. For x of length n, $cmp'(x,y)$ takes time $O(n)$; $cmp(cons(y,x))$ takes time $O(n)$.	$cmp'(x,y) =$ **if** $null(x)$ **then** $y \leq 1$ **else** $y + sum(even(x)) \leq prod(odd(x))$
$cmp(x) = 1st(\widehat{cmp}(x))$. For x of length n, $\widehat{cmp}(x)$ takes time $O(n)$; $cmp(x)$ takes time $O(n)$.	$\widehat{cmp}(x) =$ **let** $v_1 = odd(x)$ **in** **let** $v_2 = even(x)$ **in** $< sum(v_1) \leq prod(v_2), v_1, v_2 >$
If $\widehat{cmp}(x) = \widehat{r}$, then $\widehat{cmp}'(y,\widehat{r}) = \widehat{cmp}(cons(y,x))$. For x of length n, $\widehat{cmp}'(y,\hat{r})$ takes time $O(n)$; $\widehat{cmp}(cons(y,x))$ takes time $O(n)$.	$\widehat{cmp}'(y,\widehat{r}) = < y + sum(3rd(\widehat{r})) \leq prod(2nd(\widehat{r}))$, $cons(y, 3rd(\widehat{r}))$, $2nd(\widehat{r}) >$
$cmp(x) = 1st(\widetilde{cmp}(x))$. For x of length n, $\widetilde{cmp}(x)$ takes time $O(n)$; $cmp(x)$ takes time $O(n)$.	$\widetilde{cmp}(x) =$ **let** $v_1 = odd(x)$ **in** **let** $u_1 = sum(v_1)$ **in** **let** $v_2 = even(x)$ **in** **let** $u_2 = prod(v_2)$ **in** $< u_1 \leq u_2$, u_1, u_2, $sum(v_2), prod(v_1) >$
If $\widetilde{cmp}(x) = \widetilde{r}$, then $\widetilde{cmp}'(y,\widetilde{r}) = \widetilde{cmp}(cons(y,x))$. For x of length n, $\widetilde{cmp}'(y,\widetilde{r})$ takes time $O(1)$; $\widetilde{cmp}(cons(y,x))$ takes time $O(n)$.	$\widetilde{cmp}'(y,\widetilde{r}) = < y + 4th(\widetilde{r}) \leq 5th(\widetilde{r})$, $y + 4th(\widetilde{r})$, $5th(\widetilde{r})$, $2nd(\widetilde{r})$, $y * 3rd(\widetilde{r}) >$

Figure 5.2: Resulting function definitions of cmp', $\widehat{cmp}$, $\widehat{cmp}'$, $\widetilde{cmp}$, and $\widetilde{cmp}'$

often subcomputations that depend only on x and r, but not on y, and whose values can not be retrieved from the return value or intermediate results of $f_0(x)$. If the values of these subcomputations were available, then we could perhaps make f_0' faster.

To obtain such candidate auxiliary information, the basic idea is to transform $f_0(x \oplus y)$ as for incrementalization and to collect subcomputations in the transformed $f_0(x \oplus y)$ that depend only on x and whose values can not be retrieved from the return value or intermediate results of $f_0(x)$. Note that computing *intermediate results* of $f_0(x)$ incrementally, with *their* corresponding auxiliary information, is often crucial for efficient incremental computation. Thus, we modify the basic idea just described so that it starts with $\bar{f}_0(x \oplus y)$ instead of $f_0(x \oplus y)$.

Phase A has three steps. Step 1 extends f_0 to a function $\bar{f}_0$ that caches all intermediate results. Step 2 transforms $\bar{f}_0(x \oplus y)$ into a function $\bar{f}_0^{\backprime}$ that exposes candidate auxiliary information. Step 3 constructs a function $\check{f}_0$ that computes only the candidate auxiliary information in $\bar{f}_0^{\backprime}$.

5.1.1 Step A.1: Caching all intermediate results with improvements

Extending f_0 to cache all intermediate results uses the transformations in Stage I of Chapter 4. It first performs a straightforward extension transformation to embed all intermediate results in the final return value and then performs administrative simplifications.

Certain improvements can be made to the extension transformation to avoid caching redundant intermediate results. These improvements become more important for discovering auxiliary information, since the resulting program should be much simpler and therefore easier to treat in subsequent analyses and transformations. These improvements also benefit the modified version of this extension transformation used in Step A.3 for collecting candidate auxiliary information.

First of all, before applying the extension transformation, common subcomputations in both branches of a conditional expression are lifted out of the conditional. This simplifies programs in general. For caching all intermediate results, this lifting saves the extension transformation from caching values of common subcomputations at different positions in different branches, which makes it easier to reason about using these values for incremental computation. The same effect can be achieved by explicitly allocating, for values of common subcomputations in different branches, the same slot in each corresponding branch.

Next, we concentrate on major improvements. The basic idea underlying them is to avoid caching values of function applications that are already embedded in the values of their enclosing computations, since these omitted values can be retrieved from the results of the enclosing applications. These improvements are based on an *embedding analysis*.

Embedding analysis. First, we compute *embedding relations*. We use $Mf(f,i)$ to indicate whether the value of v_i is embedded in the value of $f(v_1,...,v_n)$, and we use $Me(e,v)$ to indicate whether the value of variable v is embedded in the value of expression e. These relations must satisfy the following safety requirements:

$$\begin{array}{l} \text{if } Mf(f,i) = true, \text{ then there exists an expression } f_i^{-1} \\ \qquad\qquad\qquad\qquad \text{such that, if } u = f(v_1,...,v_n), \text{ then } v_i = f_i^{-1}(u) \\ \text{if } Me(e,v) = true, \text{ then there exists an expression } e_v^{-1} \\ \qquad\qquad\qquad\qquad \text{such that, if } u = e, \text{ then } v = e_v^{-1}(u) \end{array} \tag{5.1}$$

For each function definition $f(v_1,...,v_n) = e_f$, we define $Mf(f,i) = Me(e_f, v_i)$, and we define Me recursively as in Figure 5.3. For a primitive function p, $\exists p_i^{-1}$ denotes $true$ if p has an inverse for the ith argument, and $false$ otherwise. For a conditional expression, $if^{e_1}_{e_2 e_3}$ denotes $true$ if the value of e_1 can be determined statically or inferred from the value of **if** e_1 **then** e_2 **else** e_3, and $false$ otherwise. For example, $if^{e_1}_{e_2 e_3}$ is true if e_1 is T or F, or if the two branches of the conditional expression return applications of different constructors. For a Boolean expression e_1, $e_1 \vdash Me(e,v)$ means that whenever e_1 is true, the value of v is embedded in the value of e. In order that the embedding analysis does not obviate useful caching, it considers a value to be embedded only if the value can be retrieved from the value of its immediately enclosing computation in constant time; in particular, this constraint applies to the retrievals when $\exists p_i^{-1}$ or $if^{e_1}_{e_2 e_3}$ is true.

We can easily show by induction that the safety requirements (5.1) are satisfied. To compute Mf, we start with $Mf(f,i) = true$ for every f and i and iterate using the above definitions to compute the greatest fixed point in the point-wise extension of

$$
\begin{array}{lcl}
Me(u,v) & = & \begin{cases} true & \text{if } v = u \\ false & \text{otherwise} \end{cases} \\
Me(c(e_1,...,e_n),v) & = & Me(e_1,v) \ \vee \ ... \ \vee \ Me(e_n,v) \\
Me(p(e_1,...,e_n),v) & = & \big(\exists p_1^{-1} \wedge Me(e_1,v)\big) \ \vee \ ... \ \vee \ \big(\exists p_n^{-1} \wedge Me(e_n,v)\big) \\
Me(f(e_1,...,e_n),v) & = & \big(Mf(f,1) \wedge Me(e_1,v)\big) \ \vee \ ... \ \vee \ \big(Mf(f,n) \wedge Me(e_n,v)\big) \\
Me(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, v) & = & if^{e_1}_{e_2 e_3} \wedge \big(e_1 \vdash Me(e_2,v)\big) \wedge \big(e_1 \vdash Me(e_3,v)\big) \\
Me(\textbf{let } u = e_1 \textbf{ in } e_2, v) & = & Me(e_2,v) \ \vee \ \big(Me(e_1,v) \wedge Me(e_2,u)\big)
\end{array}
$$

Figure 5.3: Definition of Me

the Boolean domain with $false \sqsubseteq true$. The iteration always terminates since these definitions are monotonic and the domain is finite.

Next, we compute *embedding tags*. For each function definition $f(v_1,...,v_n) = e_f$, we associate an embedding tag $Mtag$ with each subexpression e of e_f, indicating whether the value of e is embedded in the value of e_f, i.e., if $Mtag(e) = true$, then there exists e^{-1} such that, for all computations of e_f, if the computation of e_f contains a computation of e, then $e^{-1}(e_f) = e$, otherwise $e^{-1}(e_f) = \_$. $Mtag$ can be defined in a similar fashion to Me. We define $Mtag(e_f) = true$, and define the $true$ values of $Mtag$ for subexpressions e of e_f as in Figure 5.4; the tags of other subexpressions of e_f are defined to be $false$. These tags can be computed directly once the above

$$
\begin{array}{ll}
\text{if } Mtag(c(e_1,...,e_n)) = true & \text{then } Mtag(e_i) = true, \ \text{for } i = 1..n \\
\text{if } Mtag(p(e_1,...,e_n)) = true & \text{then } Mtag(e_i) = true \text{ if } \exists p_i^{-1}, \ \text{for } i = 1..n \\
\text{if } Mtag(f(e_1,...,e_n)) = true & \text{then } Mtag(e_i) = true \text{ if } Mf(f,i), \ \text{for } i = 1..n \\
\text{if } Mtag(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = true & \text{then } Mtag(e_i) = true \text{ if } if^{e_1}_{e_2 e_3}, \ \text{for } i = 1,2,3 \\
\text{if } Mtag(\textbf{let } v = e_1 \textbf{ in } e_2) = true & \text{then } Mtag(e_2) = true; \ Mtag(e_1) = true \text{ if } Me(e_2,v)
\end{array}
$$

Figure 5.4: Definition of $Mtag$

embedding relations are computed.

Finally, we use the embedding tags to compute, for each function f, an *embedding-all* property $Mall$ indicating whether all intermediate results of f are embedded in the value of f. We define, for each function $f(v_1,...,v_n) = e_f$,

$$Mall(f) = \bigwedge_{\substack{\text{all function applications} \\ g(e_1,...,e_n) \text{ occurring in } e_f}} Mtag(g(e_1,...,e_m)) \wedge Mall(g) \tag{5.2}$$

where $Mtag$ is with respect to e_f. To compute $Mall$, we start with $Mall(f) = true$ for all f and iterate using the definition in (5.2) until the greatest fixed point is reached. This fixed point exists for similar reasons as for Mf.

Improvements. The above embedding analysis is used to improve the extension transformation as follows. First, if $Mall(f) = true$, i.e., if all intermediate results of f are embedded in the value of f, then we do not construct an extended function for f. This makes the transformation for caching all intermediate results *idempotent*, i.e., if caching all intermediate results for a set F of function definitions yields a set $\bar{F}$ of function definitions, then caching all intermediate results for $\bar{F}$ yields the same set $\bar{F}$.

If there is a function not all of whose intermediate results are embedded in its return value, then an extended function for it needs to be defined as in (4.9). We modify the definition of $\mathcal{E}xt[\![f(e_1, ..., e_n)]\!]$ as follows. If $Mall(f) = true$, which includes the case where f does not contain function applications, then, due to the first improvement, f is not extended, so we reference the value of f directly:

$$\begin{array}{rl}\mathcal{E}xt[\![f(e_1, ..., e_n)]\!] = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\ & \textbf{let } v = f(1st(v_1), ..., 1st(v_n)) \textbf{ in} \\ & < v > @\ rst(v_1)\ @ \ ...\ @\ rst(v_n)\ @ < v > \end{array} \tag{5.3}$$

Furthermore, if $Mall(f) = true$, and $Mtag(f(e_1, ..., e_n)) = true$, i.e, the value of $f(e_1, ..., e_n)$ is embedded in the value of its enclosing application, then we avoid caching the value of f separately:

$$\begin{array}{rl}\mathcal{E}xt[\![f(e_1, ..., e_n)]\!] = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\ & < f(1st(v_1), ..., 1st(v_n)) > @\ rst(v_1)\ @\ ...\ @\ rst(v_n) \end{array} \tag{5.4}$$

To summarize, the transformation $\mathcal{E}xt$ remains the same as in Figure 4.3 except that the rule for a function application $f(e_1, ..., e_n)$ is replaced with the following: if $Mall(f) = true$ and $Mtag(f(e_1, ..., e_n)) = true$, then define $\mathcal{E}xt[\![f(e_1, ..., e_n)]\!]$ as in (5.4); else if $Mall(f) = true$ but $Mtag(f(e_1, ..., e_n)) = false$, then define $\mathcal{E}xt[\![f(e_1, ..., e_n)]\!]$ as in (5.3); otherwise, define $\mathcal{E}xt[\![f(e_1, ..., e_n)]\!]$ as in Figure 4.3. Function applications $f(e_1, ..., e_n)$ such that $Mall(f) = true$ and $Mtag(f(e_1, ..., e_n)) = true$ should not be counted by $\mathcal{P}ad$. The lengths of tuples generated by $\mathcal{P}ad$ can still be statically determined.

For the function *cmp* of Figure 5.1, this improved extension transformation yields the following functions:

$$\begin{array}{rl} \overline{cmp}(x) = & \textbf{let } v_1 = odd(x) \textbf{ in} \\ & \textbf{let } u_1 = \overline{sum}(v_1) \textbf{ in} \\ & \textbf{let } v_2 = even(x) \textbf{ in} \\ & \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\ & < 1st(u_1) \le 1st(u_2), \\ & \quad v_1, u_1, v_2, u_2 > \end{array} \qquad \begin{array}{rl} \overline{sum}(x) = & \textbf{if } null(x) \textbf{ then } < 0, \_ > \\ & \textbf{else let } v_1 = \overline{sum}(cdr(x)) \textbf{ in} \\ & \qquad < car(x) + 1st(v_1),\ v_1 > \\ \overline{prod}(x) = & \textbf{if } null(x) \textbf{ then } < 1, \_ > \\ & \textbf{else let } v_1 = \overline{prod}(cdr(x)) \textbf{ in} \\ & \qquad < car(x) * 1st(v_1),\ v_1 > \end{array} \tag{5.5}$$

Functions *odd* and *even* are not extended, since all their intermediate results are embedded in their return values.

5.1.2 Step A.2: Exposing auxiliary information by incrementalization

This step transforms $\bar{f}_0(x \oplus y)$ to expose subcomputations depending only on x and whose values can not be retrieved from the cached result of $\bar{f}_0(x)$. It uses analyses and transformations similar to those in Chapter 3 that derives an incremental program $\bar{f}_0'(x, y, \bar{r})$, by expanding subcomputations of $\bar{f}_0(x \oplus y)$ depending on both x and y and replacing those depending only on x by retrievals from $\bar{r}$ when possible.

Our goal here is not to quickly retrieve values from $\bar{r}$, but to find potentially useful auxiliary information, i.e., subcomputations depending on x (and $\bar{r}$) but not y whose values can *not* be retrieved from $\bar{r}$. Thus, time considerations in Chapter 3 are dropped here but picked up after Step A.3, as discussed in Section 5.3.

In particular, in Chapter 3, a recursive application of a function f is replaced by an application of an incremental version f' only if a fast retrieval from some cached result of the previous computation can be used as the argument for the parameter of f' that corresponds to a cached result. For example, if an incremental version $f'(x, y, r)$ is introduced to compute $f(x \oplus y)$ incrementally for $r = f(x)$, then in Chapter 3, a function application $f(g(x) \oplus h(y))$ is replaced by an application of f' only if some fast retrieval $p(r)$ for the value of $f(g(x))$ can be used as the argument for the parameter r of $f'(x, y, r)$, in which case the application is replaced by $f'(g(x), h(y), p(r))$. In Step A.2 here, an application of f is replaced by an application of f' also when a retrieval can not be found; in this case, the value needed for the cache parameter is computed directly, so for this example, the application $f(g(x) \oplus h(y))$ is replaced by $f'(g(x), h(y), f(g(x)))$. It is easy to see that, in this case, $f(g(x))$ becomes a piece of candidate auxiliary information.

Since the functions obtained from this step may be different from the incremental functions f' obtained in Chapter 3, we denote them by $f^{\backprime}$.

For the function $\overline{cmp}$ in (5.5) and input change operation $x \oplus y = cons(y, x)$, we transform the computation of $\overline{cmp}(cons(y, x))$, with $\overline{cmp}(x) = \bar{r}$:

$$
\begin{array}{lll}
\text{1. unfold } \overline{cmp}(cons(y,x)) & \text{2. unfold } odd, \overline{sum}, even\text{; simp.} & \text{3. replace appls. of } even, odd \\
= \textbf{let } v_1 = odd(cons(y,x)) \textbf{ in} & = \textbf{let } v_1' = even(x) \textbf{ in} & = \textbf{let } v_1' = 4th(\bar{r}) \textbf{ in} \\
\quad \textbf{let } u_1 = \overline{sum}(v_1) \textbf{ in} & \quad \textbf{let } u_1' = \overline{sum}(v_1') \textbf{ in} & \quad \textbf{let } u_1' = \overline{sum}(v_1') \textbf{ in} \\
\quad \textbf{let } v_2 = even(cons(y,x)) \textbf{ in} & \quad \textbf{let } v_2 = odd(x) \textbf{ in} & \quad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\
\quad \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} & \quad \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} & \quad \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\
\quad < 1st(u_1) \le 1st(u_2), & \quad < y{+}1st(u_1') \le 1st(u_2), & \quad < y{+}1st(u_1') \le 1st(u_2), \\
\qquad v_1,\ u_1,\ v_2,\ u_2 > & \qquad cons(y, v_1'), & \qquad cons(y, v_1'), \\
 & \qquad < y{+}1st(u_1'), u_1' >, v_2,\ u_2 > & \qquad < y{+}1st(u_1'), u_1' >, v_2,\ u_2 >
\end{array}
$$

Simplification yields the following function $\overline{cmp}^{\backprime}$ such that, if $\overline{cmp}(x) = \bar{r}$, then $\overline{cmp}^{\backprime}(y, \bar{r}) = \overline{cmp}(cons(y, x))$:

$$
\begin{array}{rl}
\overline{cmp}^{\backprime}(y, \bar{r}) = & \textbf{let } u_1' = \overline{sum}(4th(\bar{r})) \textbf{ in} \\
 & \textbf{let } u_2 = \overline{prod}(2nd(\bar{r})) \textbf{ in} \\
 & < y{+}1st(u_1') \le 1st(u_2),\ cons(y, 4th(\bar{r})),\ < y{+}1st(u_1'), u_1' >,\ 2nd(\bar{r}),\ u_2 >
\end{array}
\tag{5.6}
$$

where $\overline{sum}$ and $\overline{prod}$ are defined in (5.5).

5.1.3 Step A.3: Collecting candidate auxiliary information

This step collects candidate auxiliary information, i.e., intermediate results of $\bar{f}_0^{\backprime}(x, y, \bar{r})$ that depend only on x and $\bar{r}$. It is similar to Step A.1 in that both collect intermediate results; they differ in that Step A.1 collects all intermediate results, while this step collects only those that depend only on x and $\bar{r}$.

Forward Dependency Analysis. We first use a forward dependency analysis to identify subcomputations of $\bar{f_0}'(x, y, \bar{r})$ that depend only on x and $\bar{r}$. The analysis is in the same spirit as binding-time analysis [JSS85,Lau88] for partial evaluation, if we regard the arguments corresponding to x and $\bar{r}$ as static and the rest as dynamic. We compute the following sets, called *forward dependency sets*, directly.

For each function $f(v_1, ..., v_n) = e_f$, we compute a set Σ_f that contains the indices of the arguments of f such that, in all uses of f, the values of these arguments depend only on x and $\bar{r}$, and, for each subexpression e of e_f, we compute a set $\Sigma_{[e]}$ that contains the free variables in e that depend only on x and $\bar{r}$. The recursive definitions of these sets are given in Figure 5.5, where $FV(e)$ denotes the set of free variables in e and is defined as follows:

$$
\begin{array}{lcl}
FV(v) & = & \{v\} \\
FV(g(e_1, ..., e_n)) \quad \text{where } g \text{ is } c,\ p, \text{ or } f & = & FV(e_1) \cup ... \cup FV(e_n) \\
FV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & FV(e_1) \cup FV(e_2) \cup FV(e_3) \\
FV(\textbf{let } v = e_1 \textbf{ in } e_2) & = & FV(e_1) \cup (FV(e_2) \setminus \{v\})
\end{array}
$$

For each function $f(v_1, ..., v_n) = e_f$, define $\Sigma_{[e_f]} = \{v_i \mid i \in \Sigma_f\}$, and, for each subexpression e of e_f,

$$
\begin{array}{ll}
\text{if } e \text{ is } c(e_1, ..., e_n) \text{ or } p(e_1, ..., e_n) & \text{then } \Sigma_{[e_1]} = ... = \Sigma_{[e_n]} = \Sigma_{[e]} \\
\text{if } e \text{ is } f_1(e_1, ..., e_n) & \text{then } \Sigma_{[e_1]} = ... = \Sigma_{[e_n]} = \Sigma_{[e]} \text{ and } \Sigma_{f_1} = \{i \mid FV(e_i) \subseteq \Sigma_{[e]}\} \cap \Sigma_{f_1} \\
\text{if } e \text{ is } \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & \text{then } \Sigma_{[e_1]} = \Sigma_{[e_2]} = \Sigma_{[e_3]} = \Sigma_{[e]} \\
\text{if } e \text{ is } \textbf{let } v = e_1 \textbf{ in } e_2 & \text{then } \Sigma_{[e_1]} = \Sigma_{[e]} \text{ and } \Sigma_{[e_2]} = \begin{cases} \Sigma_{[e]} \cup \{v\} & \text{if } FV(e_1) \subseteq \Sigma_{[e]} \\ \Sigma_{[e]} \setminus \{v\} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.5: Definition of Σ

To compute these sets, we start with $\Sigma_{\bar{f_0}'}$ containing the indices of the arguments of $\bar{f_0}'$ corresponding to x and $\bar{r}$, and, for all other functions f, Σ_f containing the indices of all arguments of f, and iterate until a fixed point is reached. This iteration always terminates since, for each function f, f has a fixed arity, Σ_f decreases, and a lower bound $\emptyset$ exists.

For the running example, we start with $\Sigma_{\overline{cmp}'} = \{2\}$ and $\Sigma_{\overline{sum}} = \Sigma_{\overline{prod}} = \{1\}$. We obtain $\Sigma_{\overline{cmp}'} = \{2\}$ and $\Sigma_{\overline{sum}} = \Sigma_{\overline{prod}} = \{1\}$. For every subexpression e in the definition of $\overline{cmp}'(y, \bar{r})$, $\bar{r} \in \Sigma_{[e]}$. For every subexpression e in the definitions of $\overline{sum}(x)$ and $\overline{prod}(x)$, $\Sigma_{[e]} = \{x\}$.

Collection Transformation. We next use a collection transformation to collect the candidate auxiliary information. The main difference between this collection transformation and the extension transformation in Step A.1 is that, in the former, the value originally computed by a subexpression is returned only if it depends only on x and $\bar{r}$, while in the latter, the value originally computed by a subexpression is always returned.

Basically, for each function $f(v_1, ..., v_n) = e$ called in the program for $\bar{f_0}$ and such that $\Sigma_f \neq \emptyset$, we construct a function definition

$$\check{f}(v_{i_1}, ..., v_{i_k}) = \mathcal{C}ol[\![e]\!] \tag{5.7}$$

where $\Sigma_f = \{i_1, ..., i_k\}$ and $1 \leq i_1 < ... < i_k \leq n$. $\mathcal{C}ol[\![e]\!]$ collects the results of intermediate function applications in e that have been statically determined to depend only on x and $\bar{r}$. Note, however, that an improvement similar to that in Step A.1 is made, namely, we avoid constructing such a collected version for f if $\Sigma_f = \{1, ..., n\}$ and $Mall(f) = true$.

The transformation $\mathcal{C}ol$ always first examines whether its argument expression e has been determined to depend only on x and $\bar{r}$, i.e., $FV(e) \subseteq \Sigma_{[e]}$. If so, $\mathcal{C}ol[\![e]\!] = \mathcal{E}xt[\![e]\!]$, where $\mathcal{E}xt$ is the improved extension transformation defined in Step A.1. Otherwise, $\mathcal{C}ol[\![e]\!]$ is defined as in Figure 5.6, where $\breve{\mathcal{P}ad}[\![e]\!]$ generates a tuple of _'s of length equal to the number of the function applications in e, except that function applications $f(e_1, ..., e_n)$ such that $\Sigma_f = \emptyset$, or $\Sigma_f = \{1, ..., n\}$ but $Mall(f) = true$ and $Mtag(f(e_1, ..., e_n)) = true$ are not counted. Note that if e has been determined to depend only on x and $\bar{r}$, then $1st(\mathcal{C}ol[\![e]\!])$ is just the original value of e; otherwise, $\mathcal{C}ol[\![e]\!]$ contains only values of intermediate function applications.

$$
\begin{array}{lcll}
\mathcal{C}ol[\![v]\!] & = & <> & \\
\mathcal{C}ol[\![g(e_1, ..., e_n)]\!] \;\; \text{where } g \text{ is } c \text{ or } p & & & \\
 & = & \mathcal{C}ol[\![e_1]\!] \;@\; ... \;@\; \mathcal{C}ol[\![e_n]\!] & \\
\mathcal{C}ol[\![f(e_1, ..., e_n)]\!] & = & \textbf{let } v_1 = \mathcal{C}ol[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{C}ol[\![e_n]\!] \textbf{ in } e'_1 \;@\; ... \;@\; e'_n \;@\; e' & \\
 & & \text{where } e'_i = \begin{cases} rst(v_i) & \text{if } i \in \Sigma_f \\ v_i & \text{otherwise} \end{cases} & \\
 & & \qquad e' = \begin{cases} <> & \text{if } \Sigma_f = \emptyset \\ <\check{f}(1st(v_{i_1}), ..., 1st(v_{i_k}))> & \text{otherwise} \end{cases} & \\
 & & \qquad \text{where } \Sigma_f = \{i_1, ..., i_k\} \text{ and } 1 \leq i_1 < ... < i_k \leq n & \\
\mathcal{C}ol[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] & = & \textbf{let } v_1 = \mathcal{C}ol[\![e_1]\!] \textbf{ in} & \text{if } FV(e_1) \subseteq \Sigma_{[e_1]} \\
 & & \textbf{if } 1st(v_1) \textbf{ then let } v_2 = \mathcal{C}ol[\![e_2]\!] \textbf{ in} & \\
 & & \qquad\qquad rst(v_1) \;@\; v_2 \;@\; \breve{\mathcal{P}ad}[\![e_3]\!] & \\
 & & \qquad \textbf{else let } v_3 = \mathcal{C}ol[\![e_3]\!] \textbf{ in} & \\
 & & \qquad\qquad rst(v_1) \;@\; \breve{\mathcal{P}ad}[\![e_2]\!] \;@\; v_3 & \\
 & = & \textbf{let } v_1 = \mathcal{C}ol[\![e_1]\!] \textbf{ in let } v_2 = \mathcal{C}ol[\![e_2]\!] \textbf{ in} & \text{otherwise} \\
 & & \qquad\qquad \textbf{let } v_3 = \mathcal{C}ol[\![e_3]\!] \textbf{ in} & \\
 & & \qquad\qquad v_1 \;@\; v_2 \;@\; v_3 & \\
\mathcal{C}ol[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!] & = & \textbf{let } v_1 = \mathcal{C}ol[\![e_1]\!] \textbf{ in} & \text{if } FV(e_1) \subseteq \Sigma_{[e_1]} \\
 & & \textbf{let } v = 1st(v_1) \textbf{ in let } v_2 = \mathcal{C}ol[\![e_2]\!] \textbf{ in} & \\
 & & \qquad\qquad rst(v_1) \;@\; v_2 & \\
 & = & \textbf{let } v_1 = \mathcal{C}ol[\![e_1]\!] \textbf{ in let } v_2 = \mathcal{C}ol[\![e_2]\!] \textbf{ in} & \text{otherwise} \\
 & & \qquad\qquad v_1 \;@\; v_2 &
\end{array}
$$

Figure 5.6: Definition of $\mathcal{C}ol$

For the function $\overline{cmp}'$ in (5.6), collecting all intermediate results that depend only on its second parameter yields

$$\check{cmp}(\bar{r}) = \; < \overline{sum}(4th(\bar{r})),\ \overline{prod}(2nd(\bar{r})) > \tag{5.8}$$

where $\overline{sum}$ and $\overline{prod}$ are defined as in (5.5). We can see that computing the auxiliary information $\check{cmp}(\bar{r})$ is no slower than computing $cmp(x)$. We will see that this guarantees that incremental computation using the program obtained at the end is at least as fast as computing cmp from scratch.

5.2 Phase B: Using auxiliary information

Phase B determines which pieces of the collected candidate auxiliary information are useful for incremental computation of $f_0(x \oplus y)$ and exactly how they can be used. The basic idea is to merge the candidate auxiliary information with the original computation of $f_0(x)$, derive an incremental version for the resulting program, and determine the least information useful for computing the value of $f_0(x \oplus y)$ in that incremental version.

However, we want the incremental computation of $f_0(x \oplus y)$ to have access to the auxiliary information *in addition to* the intermediate results of $f_0(x)$. Thus, we merge the candidate auxiliary information in $\check{f}_0(x, \bar{r})$ with $\bar{f}_0(x)$ instead of $f_0(x)$. After deriving an incremental version for the resulting program, we prune out the useless auxiliary information and the useless intermediate results.

Phase B has three steps. Step 1 merges $\check{f}_0$ with $\bar{f}_0$ to form a function $\check{\bar{f}}_0$ that returns candidate auxiliary information as well as all intermediate results. It also determines a projection Π_0 that projects the return value of f_0 out of $\check{\bar{f}}_0$. Step 2 incrementalizes $\check{\bar{f}}_0$ under $\oplus$ to obtain an incremental version $\check{\bar{f}}_0'$. Step 3 prunes out of $\check{\bar{f}}_0$ and $\check{\bar{f}}_0'$ the intermediate results and auxiliary information that are not useful.

5.2.1 Step B.1: Combining intermediate results and auxiliary information

To merge the candidate auxiliary information with $\bar{f}_0$, we could simply attach it onto $\bar{f}_0$ by defining a function $\check{\bar{f}}_0$ to be the pair of $\bar{f}_0$ and $\check{f}_0$:

$$\check{\bar{f}}_0(x) = \quad \textbf{let } \bar{r} = \bar{f}_0(x) \textbf{ in let } \check{r} = \check{f}_0(x, \bar{r}) \textbf{ in } < \bar{r}, \check{r} >$$

and use the projection $\Pi_0(\check{\bar{r}}) = 1st(1st(\check{\bar{r}}))$ to project out the original return value of f_0. However, we can do better by using a transformation to integrate the computation of $\check{f}_0$ more tightly into the computation of $\bar{f}_0$, as opposed to carrying out two disjoint computations. The integrated computation is usually more efficient; so is its incremental version.

We do not describe the integration in detail. Basically, it uses traditional transformation techniques [BD77] like those used in tupling tactic [Fea82] and partial evaluation [JGS93]: introducing functions to compute function applications, unfolding, simplifying primitive function applications, driving, replacing recursive applications with introduced functions, *etc.* We require only that $\Pi_0(\bar{\check{f}}_0(x))$ always project out $1st(\bar{f}_0(x))$, which is the value of $f_0(x)$, and that the values of all other components of $\bar{f}_0(x)$ and $\check{f}_0(x,\bar{r})$ be embedded in the value of $\bar{\check{f}}_0(x)$. This allows re-arranging the order of the components in the return value into the most straightforward form and adjust the projection Π_0.

For the functions $\overline{cmp}$ in (5.5) and $c\check{m}p$ in (5.8), we first define a function

$$\overline{c\check{m}p}(x) = \ \ \mathbf{let}\ \bar{r} = \overline{cmp}(x)\ \mathbf{in}\ \mathbf{let}\ \check{r} = c\check{m}p(\bar{r})\ \mathbf{in}\ <\bar{r},\check{r}>$$

and a projection $\Pi_0(\bar{\check{r}}) = 1st(1st(\bar{\check{r}}))$. Next, we transform $\overline{c\check{m}p}(x)$ to integrate the computations of $\overline{cmp}$ and $c\check{m}p$:

1. unfold $\overline{c\check{m}p}$, then $\overline{cmp}$ and $c\check{m}p$

$$\begin{array}{rl} = & \mathbf{let}\ \bar{r} = \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\ & \qquad\quad \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\ & \qquad\quad \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\ & \qquad\quad \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\ & \qquad\quad <1st(u_1) \le 1st(u_2), \\ & \qquad\qquad v_1, u_1, v_2, u_2 >\ \mathbf{in} \\ & \mathbf{let}\ \check{r} = <\overline{sum}(4th(\bar{r})), \\ & \qquad\qquad \overline{prod}(2nd(\bar{r}))>\ \mathbf{in} \\ & <\bar{r},\check{r}> \end{array}$$

2. lift bindings and simplify

$$\begin{array}{rl} = & \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\ & \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\ & \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\ & \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\ & \mathbf{let}\ \bar{r} = <1st(u_1) \le 1st(u_2), \\ & \qquad\qquad v_1, u_1, v_2, u_2 >\ \mathbf{in} \\ & \mathbf{let}\ \check{r} = <\overline{sum}(v_2), \\ & \qquad\qquad \overline{prod}(v_1)>\ \mathbf{in} \\ & <\bar{r},\check{r}> \end{array}$$

3. unfold bindings for $\bar{r}$ and $\check{r}$

$$\begin{array}{rl} = & \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\ & \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\ & \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\ & \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\ & <<1st(u_1) \le 1st(u_2), \\ & \qquad v_1, u_1, v_2, u_2 >, \\ & \quad <\overline{sum}(v_2), \overline{prod}(v_1)>> \end{array}$$

Simplifying the return value and Π_0, we obtain the following function $\overline{c\check{m}p}$:

$$\begin{array}{rl} \overline{c\check{m}p}(x) = & \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\ & \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\ & \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\ & \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\ & <1st(u_1) \le 1st(u_2),\ v_1,\ u_1,\ v_2,\ u_2,\ \overline{sum}(v_2),\ \overline{prod}(v_1)> \end{array} \tag{5.9}$$

and the projection $\Pi_0(\bar{\check{r}}) = 1st(\bar{\check{r}})$.

5.2.2 Step B.2: Incrementalization

To derive an incremental version $\bar{\check{f}}_0'$ of $\bar{\check{f}}_0$ under $\oplus$, we can use the method in Chapter 3. Basically, it identifies subcomputations in $\bar{\check{f}}_0(x \oplus y)$ whose values can be retrieved from the cached result $\bar{\check{r}}$ of $\bar{\check{f}}_0(x)$, replaces them by corresponding retrievals, and captures the resulting way of computing $\bar{\check{f}}_0(x \oplus y)$ in the incremental version $\bar{\check{f}}_0{}'(x, y, \bar{\check{r}})$.

For the function $\overline{c\check{m}p}$ in (5.9) and input change operation $x \oplus y = cons(y,x)$, we derive an incremental version of $\overline{c\check{m}p}$ under $\oplus$:

1. unfold $\overline{c\check{m}p}(cons(y,x))$

$$\begin{array}{rl} = & \mathbf{let}\ v_1 = odd(cons(y,x))\ \mathbf{in} \\ & \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\ & \mathbf{let}\ v_2 = even(cons(y,x))\ \mathbf{in} \\ & \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\ & <1st(u_1) \le 1st(u_2), \\ & \quad v_1, u_1, v_2, u_2, \\ & \quad \overline{sum}(v_2), \overline{prod}(v_1)> \end{array}$$

2. transform applications

$$\begin{array}{rl} = & \mathbf{let}\ v_1' = even(x)\ \mathbf{in} \\ & \mathbf{let}\ u_1' = \overline{sum}(v_1')\ \mathbf{in} \\ & \mathbf{let}\ v_2 = odd(x)\ \mathbf{in} \\ & \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\ & \mathbf{let}\ u_4' = \overline{prod}(v_1')\ \mathbf{in} \\ & <y + 1st(u_1') \le 1st(u_2), \\ & \quad cons(y, v_1'), \\ & \quad <y+1st(u_1'), u_1'>, v_2,\ u_2, \\ & \quad \overline{sum}(v_2), <y*1st(u_4'), u_4'>> \end{array}$$

3. replace appls. by retrievals

$$\begin{array}{rl} = & \mathbf{let}\ v_1' = 4th(\bar{\check{r}})\ \mathbf{in} \\ & \mathbf{let}\ u_1' = 6th(\bar{\check{r}})\ \mathbf{in} \\ & \mathbf{let}\ v_2 = 2nd(\bar{\check{r}})\ \mathbf{in} \\ & \mathbf{let}\ u_2 = 7th(\bar{\check{r}})\ \mathbf{in} \\ & \mathbf{let}\ u_4' = 5th(\bar{\check{r}})\ \mathbf{in} \\ & <y + 1st(u_1') \le 1st(u_2), \\ & \quad cons(y, v_1'), \\ & \quad <y+1st(u_1'), u_1'>, v_2,\ u_2, \\ & \quad 3rd(\bar{\check{r}}), <y*1st(u_4'), u_4'>> \end{array}$$

Simplification yields the following incremental version $\bar{\check{cmp}}'$ such that, if $\bar{\check{cmp}}(x) = \bar{\check{r}}$, then $\bar{\check{cmp}}'(y, \bar{\check{r}}) = \bar{\check{cmp}}(cons(y, x))$:

$$\begin{array}{rl} \bar{\check{cmp}}'(y,\bar{\check{r}}) = & <y{+}1st(6th(\bar{\check{r}})) \le 1st(7th(\bar{\check{r}})), \\ & cons(y, 4th(\bar{\check{r}})),\ <y{+}1st(6th(\bar{\check{r}})), 6th(\bar{\check{r}})>,\ 2nd(\bar{\check{r}}),\ 7th(\bar{\check{r}}), \\ & 3rd(\bar{\check{r}}),\ <y{*}1st(5th(\bar{\check{r}})), 5th(\bar{\check{r}})>> \end{array} \tag{5.10}$$

Clearly, $\bar{\check{cmp}}'(y, \bar{\check{r}})$ computes $\bar{\check{cmp}}(cons(y, x))$ in only $O(1)$ time.

5.2.3 Step B.3: Pruning

To prune $\bar{\check{f_0}}$ and $\bar{\check{f_0}}'$, we use the analyses and transformations in Stage III of Chapter 4. A backward dependency analysis determines the components of $\bar{\check{r}}$ and subcomputations of $\check{f_0}'$ whose values are useful in computing $\Pi_0(\check{f_0}'(x, y, \bar{\check{r}}))$, which is the value of f_0. A pruning transformation replaces useless computations with $\_$. Finally, the resulting functions are optimized by eliminating the $\_$ components, adjusting the selectors, *etc.*

For the functions $\bar{\check{cmp}}$ in (5.9) and $\bar{\check{cmp}}'$ in (5.10), we obtain

$$\begin{array}{rl} \widetilde{cmp}(x) = & \textbf{let } v_1 = odd(x) \textbf{ in} \\ & \textbf{let } u_1 = \overline{sum}(v_1) \textbf{ in} \\ & \textbf{let } v_2 = even(x) \textbf{ in} \\ & \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\ & <1st(u_1) \le 1st(u_2), \\ & \quad \_,\ <1st(u_1), \_>,\ \_,\ <1st(u_2), \_>, <1st(\overline{sum}(v_2)), \_>,\ <1st(\overline{prod}(v_1)), \_>> \end{array}$$

$$\begin{array}{rl} \widetilde{cmp}'(y,\bar{\check{r}}) = & <y{+}1st(6th(\bar{\check{r}})) \le 1st(7th(\bar{\check{r}})), \\ & \quad \_, <y{+}1st(6th(\bar{\check{r}})), \_>, \_, <1st(7th(\bar{\check{r}})), \_>, <1st(3rd(\bar{\check{r}})), \_>, <y{*}1st(5th(\bar{\check{r}})), \_>> \end{array}$$

Optimizing these two functions yields the final functions $\widetilde{cmp}$ and $\widetilde{cmp}'$, which appear in Figure 5.2.

5.3 Discussion

Auxiliary information is intended to be used for incremental computation and is maintained incrementally, so at the step of discovering it, we should not be limited by the time complexity of computing it from scratch; this is why time considerations were dropped in Step A.2. However, to make the overall approach effective, we must consider the cost of computing and maintaining the auxiliary information. Here, we simply require that the candidate auxiliary information be computed at least as fast as the original program, i.e., $t(\check{f_0}(x, \bar{r})) \le t(f_0(x))$ for $\bar{r} = \bar{f_0}(x)$, which can be checked after Step A.3. We guarantee this condition by simply dropping pieces of candidate auxiliary information for which it can not be confirmed.

Suppose Step B.1 projects out the original value using $1st$. With the above condition, in a similar way to Section 4.1, we can show that, if $f_0(x) = r$, then

$$1st(\widetilde{f_0}(x)) = r \quad \text{and} \quad t(\widetilde{f_0}(x)) \le t(f_0(x)) \tag{5.11}$$

and if $f_0(x \oplus y) = r'$ and $\widetilde{f_0}(x) = \tilde{r}$, then

$$1st(\widetilde{f_0}'(x, y, \tilde{r})) = r', \;\; \widetilde{f_0}'(x, y, \tilde{r}) = \widetilde{f_0}(x \oplus y), \;\; \text{and} \;\; t(\widetilde{f_0}'(x, y, \tilde{r})) \leq t(f_0(x \oplus y)). \tag{5.12}$$

i.e., the functions $\widetilde{f_0}$ and $\widetilde{f_0}'$ preserve the semantics and compute asymptotically at least as fast. Similarly, due to the transformations used in Steps B.2 and B.3, $\widetilde{f_0}(x)$ may terminate more often than $f_0(x)$, and $\widetilde{f_0}'(x, y, \tilde{r})$ may terminate more often than $f_0(x \oplus y)$.

The two-phase method consists of six separate steps. Each component performs relatively independent analyses and transformations. Thus, the method is modular. This facilitates integrating other techniques into this framework and re-using these components for other optimizations. In particular, techniques for deriving incremental programs in Chapter 3 and caching intermediate results in Chapter 4 are used for discovering auxiliary information in this chapter, even though discovering auxiliary information is regarded as extending the methods for the former two. These three parts naturally complement one another to form a comprehensive principled approach to incremental computation.

Transformation and analysis techniques. The comprehensive approach for incremental computation discussed in this chapter is made possible by combining a number of program transformation and analysis techniques. We summarize them here.

First, Step A.1 uses the transformations in Stage I of the method in Chapter 4 to cache all intermediate results. In addition, an embedding analysis is developed to help the extension transformation avoid caching redundant information. Step A.2 uses a slightly revised version of the incrementalization method in Chapter 3 to expose candidate auxiliary information. Then, to collect the candidate auxiliary information, Step A.3 modifies the extension transformation to make use of the results of a forward dependency analysis.

Our forward dependency analysis is equivalent to binding-time analysis [JSS85, Lau88]. However, the application here is different from that in partial evaluation [JGS93]. In partial evaluation, the goal is to obtain a residual program that is specialized on a given set of static arguments and takes only the dynamic arguments, while here, the goal is to construct a program that computes only on the arguments corresponding to the "static" part without considering those corresponding to the "dynamic" part. In this aspect, the resulting program here is similar to the slice obtained from forward slicing [Wei84]. However, our forward dependency analysis is different from the forward slicing analysis in that forward dependency analysis finds parts of a program that depend *only* on certain information, while forward slicing finds parts of a program that depend *possibly* on certain information. Another distinguishing aspect is that the resulting program here returns all intermediate results on the arguments of interest as well.

Step B.1 merges candidate auxiliary information with intermediate results, which may make use of a collection of existing transformation techniques like tupling [Fea82, Pet84,Chi93]. Step B.2 uses the incrementalization method in Chapter 3, just as

Stage II of the method in Chapter 4. Finally, Step B.3 prunes the resulting programs using the backward dependency analysis just as in Stage III of the method in Chapter 4. This saves time and space by computing, using, and maintaining only useful intermediate results and auxiliary information.

Multi-pass discovery of auxiliary information. The function $\widetilde{f_0}$ can sometimes be computed even faster by maintaining still more auxiliary information, in particular, for incrementally computing the auxiliary information in it. For example, suppose we consider multiplication to be much more expensive than addition. Suppose we want to compute $f(x) = x{*}x{*}x$ incrementally under $x' = x+1$. If we do not use auxiliary information, we get

$$f'(x,r) = \ r + 3{*}x{*}x + 3{*}x + 1$$

such that, if $f(x) = r$, then $f'(x,r) = f(x+1)$. If we use auxiliary information $3{*}x{*}x$ and $3{*}x$, then we get $\tilde{f}(x) = < x{*}x{*}x,\ 3{*}x{*}x,\ 3{*}x >$ and

$$\tilde{f}'(x,\tilde{r}) = < 1st(\tilde{r}) + 2nd(\tilde{r}) + 3rd(\tilde{r}) + 1,\ 2nd(\tilde{r}) + 2{*}3rd(\tilde{r}) + 3,\ 3rd(\tilde{r}) + 3 >$$

such that, if $\tilde{f}(x) = \tilde{r}$, then $\tilde{f}'(x,\tilde{r}) = \tilde{f}(x+1)$. While $f(x+1)$ uses two multiplications and $f'(x,r)$ uses three, $\tilde{f}'(x,\tilde{r})$ uses only one. But we can use the auxiliary information $2{*}3rd(\tilde{r})$, which equals $6{*}x$, to compute the auxiliary information $3{*}x{*}x$, and obtain $\tilde{f}(x) = < x{*}x{*}x,\ 3{*}x{*}x,\ 3{*}x,\ 6{*}x >$ and

$$\tilde{f}'(x,\tilde{r}) = < 1st(\tilde{r}){+}2nd(\tilde{r}){+}3rd(\tilde{r}){+}1,\ 2nd(\tilde{r}){+}4th(\tilde{r}){+}3,\ 3rd(\tilde{r}){+}3,\ 4th(\tilde{r}){+}6 >$$

such that, if $\tilde{f}(x) = \tilde{r}$, then $\tilde{f}'(x,\tilde{r}) = \tilde{f}(x+1)$. Now $\tilde{f}'(x,\tilde{r})$ uses no multiplications.

To obtain such auxiliary information of auxiliary information, we can iterate the above approach. However, since we guarantee only that the resulting incremental program computes at least as fast as computing from scratch, such iteration may not terminate. To ensure termination, we iterate either until all subcomputations in the incremental computation that depend on x do not cost too much, or until an imposed maximum number of iterations is reached. Again, further study is needed in time and space analyses and the trade-off between them to help decide whether or not to use certain auxiliary information.

Other auxiliary information. There are cases where the auxiliary information discovered using the above approach is not sufficient for efficient incremental computation. In these cases, use of the auxiliary information typically involves special high-level properties of the particular problem at hand. For example, in SETL, several kinds of maps are maintained for functions that are not continuous for certain input changes [Pai83]. In dynamic graph algorithms, several special data structures are employed: dynamic trees [ST83], topology trees [Fre85], and sparsification trees [EGIN92].

Ideally, we can collect classes of special parameterized data structures as auxiliary information parameterized with certain data types. Then, we can systematically extend a program to compute such auxiliary information and maintain it incrementally.

In the worst case, we can code manually discovered auxiliary information to obtain a program $\widetilde{f_0}$, and then use our systematic approach to derive an incremental version of $\widetilde{f_0}$ that incrementally computes new outputs using the auxiliary information and also maintains the auxiliary information.

5.4 Examples

The running example on list processing illustrates the application of our approach to solving explicit incremental problems for, e.g., interactive systems and reactive systems. Other applications include optimizing compilers and transformational programming. First, given a simplifier based on properties of primitive operations like distributivities, the class of auxiliary information this method discovers includes all that is needed in classical strength-reduction rules used in optimizing compilers. Second, this class of auxiliary information covers much of what is needed in transformational programming. The benefit of our approach is evident in that it allows one to derive efficient incremental programs in a systematic way and, therefore, helps avoid unnecessary errors during the derivation.

This section presents an example for each of these two applications. The examples are based on problems in VLSI design and graph algorithms, respectively.

5.4.1 Binary integer square root

This example is from [OLHA94], where a specification of a non-restoring binary integer square root algorithm is transformed into a VLSI circuit design and implementation. In that work, a strength-reduced program was manually discovered and then proved correct using Nuprl [C+86]. Here, we show how our method can automatically derive the strength reductions. This is of particular interest in light of the recent Pentium chip flaw, since the current technology for proving correctness of a chip is still being argued [Gla95].

The initial specification of the algorithm is given in Figure 5.7. Given a binary integer n of l bits, where $n \geq 0$ and $l \geq 1$, it computes the binary integer square root m using the non-restoring method [Flo63,OLHA94], which is exact for perfect squares and off by at most 1 for other integers. In hardware, multiplications and exponentials

---

$m := 2^{l-1}$
for $i := l-2$ **downto** 0 **do**
  $p := n - m^2$;
  if $p > 0$ **then**
    $m := m + 2^i$
  else if $p < 0$ **then**
    $m := m - 2^i$

Figure 5.7: Specification of binary integer square root algorithm

---

are much more expensive than additions and shifts (doublings or halvings), so the goal is to replace the former by the latter.

To simplify the presentation, we jump to the heart of the problem, namely, computing $n - m^2$ and 2^i incrementally in each iteration under the change $m' = m \pm 2^i$ and $i' = i - 1$. Let function f_0 be

$$f_0(n, m, i) = \quad pair(n - m^2, 2^i)$$

where $pair$ is a constructor with selectors $fst(a, b) = a$ and $snd(a, b) = b$, and input change operation $\oplus$ be

$$\langle n', m', i' \rangle = \langle n, m, i \rangle \oplus \langle\ \rangle = \langle n,\ m \pm 2^i,\ i - 1 \rangle$$

A.1. We cache all intermediate results of f_0 and obtain

$$\bar{f}_0(n, m, i) = \quad \textbf{let}\ v = m^2\ \textbf{in}\ < pair(n - v,\ 2^i),\ v >$$

A.2. We transform $\bar{f}_0$ under $\oplus$ and obtain

$$\begin{aligned} &\bar{f}_0'(n, m, i, \bar{r}) \\ =\ \ & \textbf{let}\ v = (m \pm 2^i)^2\ \textbf{in}\ < pair(n - v,\ 2^{i-1}),\ v > \\ =\ \ & \textbf{let}\ v = m^2 \pm 2{*}m{*}2^i + (2^i)^2\ \textbf{in}\ < pair(n - v,\ 2^i/2),\ v > \\ =\ \ & \textbf{let}\ v = 2nd(\bar{r}) \pm 2{*}m{*}snd(1st(\bar{r})) + (snd(1st(\bar{r})))^2\ \textbf{in}\ < pair(n-v,\ snd(1st(\bar{r}))/2),\ v > \end{aligned}$$

A.3. We collect candidate auxiliary information and obtain

$$\check{f}_0(n, m, i, \bar{r}) = \ < 2{*}m{*}snd(1st(\bar{r})),\ (snd(1st(\bar{r})))^2 > \tag{5.13}$$

B.1. We merge the collected candidate auxiliary information with $\bar{f}_0$ and obtain $\Pi_0(\bar{\check{r}}) = 1st(\bar{\check{r}})$ and

$$\bar{\check{f}}_0(n, m, i) = \quad \textbf{let}\ v{=}m^2\ \textbf{in}\ \textbf{let}\ u{=}2^i\ \textbf{in}\ <pair(n{-}v, u),\ v,\ 2{*}m{*}u,\ u^2>$$

B.2. We derive an incremental version of $\bar{\check{f}}_0$ under $\oplus$ and obtain

$$\begin{aligned} &\bar{\check{f}}_0{}'(n, m, i, \bar{\check{r}}) \\ =\ \ & \textbf{let}\ v = (m \pm 2^i)^2\ \textbf{in}\ \textbf{let}\ u = 2^{i-1}\ \textbf{in}\ < pair(n - v, u),\ v,\ 2{*}(m \pm 2^i){*}u,\ u^2 > \\ =\ \ & \textbf{let}\ v = m^2 {\pm} 2{*}m{*}2^i + (2^i)^2\ \textbf{in}\ \textbf{let}\ u = 2^i/2\ \textbf{in}\ <pair(n{-}v, u),\ v,\ 2{*}m{*}u{\pm}2{*}2^i{*}u,\ u^2> \\ =\ \ & \textbf{let}\ v = 2nd(\bar{\check{r}}) \pm 3rd(\bar{\check{r}}) + 4th(\bar{\check{r}})\ \textbf{in}\ \textbf{let}\ u = snd(1st(\bar{\check{r}}))/2\ \textbf{in} \\ & < pair(fst(1st(\bar{\check{r}})) \mp 3rd(\bar{\check{r}}) - 4th(\bar{\check{r}}),\ u),\ v,\ 3rd(\bar{\check{r}})/2 \pm 4th(\bar{\check{r}}),\ 4th(\bar{\check{r}})/4 > \end{aligned}$$

B.3. We prune functions $\bar{\check{f}}_0$ and $\bar{\check{f}}_0{}'$ and obtain

$$\tilde{f}_0(n, m, i) = \quad \textbf{let}\ u = 2^i\ \textbf{in}\ < pair(n - m^2, u),\ \_,\ 2{*}m{*}u,\ u^2 >$$

$$\begin{aligned} &\tilde{f}_0{}'(n, m, i, \bar{\check{r}}) \\ =\ \ & <pair(fst(1st(\bar{\check{r}})){\mp}3rd(\bar{\check{r}}){-}4th(\bar{\check{r}}),\ snd(1st(\bar{\check{r}}))/2),\ \_,\ 3rd(\bar{\check{r}})/2{\pm}4th(\bar{\check{r}}),\ 4th(\bar{\check{r}})/4> \end{aligned}$$

Eliminating the _ components, we obtain

$$\tilde{f}_0(n, m, i) = \quad \textbf{let}\ u = 2^i\ \textbf{in}\ < pair(n - m^2, u),\ 2{*}m{*}u,\ u^2 > \tag{5.14}$$

$$\begin{aligned} &\tilde{f}_0{}'(n, m, i, \tilde{r}) \\ =\ \ & < pair(fst(1st(\tilde{r})){\mp}2nd(\tilde{r}){-}3rd(\tilde{r}),\ snd(1st(\tilde{r}))/2),\ 2nd(\tilde{r})/2{\pm}3rd(\tilde{r}),\ 3rd(\tilde{r})/4 > \end{aligned} \tag{5.15}$$

Thus, the expensive multiplications and exponentials in each iteration are completely replaced by additions and shifts. Following our systematic approach, we even discover that an extra shift is done in [OLHA94]. Thus, such systematic transformational approach is not only greatly desired for automating designs and guaranteeing correctness, but also also for helping reduce the cost.

5.4.2 Path sequence problem

This example is from [Bir84]. Given a directed acyclic graph, and a string whose elements are vertices in the graph, the problem is to compute the length of the longest subsequence in the string that forms a path in the graph. We focus on the second half of the example, where an exponential-time recursive solution is improved (incorrectly in [Bir84], correctly in [Bir85]).

A function llp is defined to compute the desired length, as below. The input string is given explicitly as the argument, and the input graph is given as a predicate arc such that $\mathrm{arc}(a, b)$ is true if and only if there is an edge from vertex a to vertex b in the graph. A primitive function max is used to return the larger number of its two arguments.

$$
\begin{array}{lcl}
llp(l) & : & \text{max length of subsequence in string } l \text{ that is a path in a graph} \\
llp(l) & = & \textbf{if } null(l) \textbf{ then } 0 \\
 & & \textbf{else } \max(llp(cdr(l)), 1{+}f(car(l), cdr(l))) \\
f(n,l) & = & \textbf{if } null(l) \textbf{ then } 0 \\
 & & \textbf{else if } \mathrm{arc}(n, car(l)) \textbf{ then} \\
 & & \quad \max(f(n, cdr(l)), 1{+}f(car(l), cdr(l))) \\
 & & \textbf{else } f(n, cdr(l))
\end{array}
$$

Figure 5.8: Example function definition of llp

First, we formulate the problem as computing llp incrementally under the input change operation $l \oplus i = cons(i, l)$.

A.1. We cache all intermediate results of llp and obtain

$$
\begin{array}{ll}
\overline{llp}(l) = & \textbf{if } null(l) \textbf{ then } <0, \_, \_> \\
 & \textbf{else let } v_1 = \overline{llp}(cdr(l)) \textbf{ in} \\
 & \quad \textbf{let } v_2 = \bar{f}(car(l), cdr(l)) \textbf{ in} \\
 & \quad <\max(1st(v_1), 1{+}1st(v_2)),\ v_1,\ v_2> \\
\bar{f}(n,l) = & \textbf{if } null(l) \textbf{ then } <0, \_, \_> \\
 & \textbf{else let } v_1 = \bar{f}(n, cdr(l)) \textbf{ in} \\
 & \quad \textbf{if } \mathrm{arc}(n, car(l)) \textbf{ then} \\
 & \quad\quad \textbf{let } v_2 = \bar{f}(car(l), cdr(l)) \textbf{ in} \\
 & \quad\quad <\max(1st(v_1), 1{+}1st(v_2)),\ v_1,\ v_2> \\
 & \quad \textbf{else } <1st(v_1),\ v_1,\ \_>
\end{array}
\tag{5.16}
$$

A.2. We define $\overline{llp}^{\backprime}(i,l,\bar r)$ to transform $\overline{llp}(cons(i,l))$, with $\overline{llp}(l)=\bar r$:

$$\begin{array}{ll} & \text{1. unfold } \overline{llp}(cons(i,l)),\text{ simplify} \\ = & \textbf{let } v_1=\overline{llp}(l) \textbf{ in} \\ & \textbf{let } v_2=\bar f(i,l) \textbf{ in} \\ & <\max(1st(v_1),1{+}1st(v_2)),v_1,v_2> \end{array}$$

$$\begin{array}{ll} & \text{2. replace applications, separate cases} \\ = & \textbf{let } v_1=\bar r \textbf{ in} \\ & \textbf{let } v_2=\textbf{if } null(l) \textbf{ then } <0,\_,\_> \\ & \qquad\quad \textbf{else } \bar f^{\backprime}(i,l,3rd(\bar r)) \textbf{ in} \\ & <\max(1st(v_1),1{+}1st(v_2)),\ v_1,\ v_2> \end{array}$$

$$\begin{array}{ll} & \text{3. lift conditions, simplify} \\ = & \textbf{if } null(l) \textbf{ then} \\ & \qquad <1,<0,\_,\_>,<0,\_,\_>> \\ & \textbf{else let } v_2=\bar f^{\backprime}(i,l,3rd(\bar r)) \textbf{ in} \\ & \qquad <\max(1st(\bar r),1{+}1st(v_2)),\ \bar r,\ v_2> \end{array}$$

where we define $\bar f^{\backprime}(i,l,\bar r_1)$ to transform $\bar f(i,l)$, with $l\neq nil$ and $\bar f(car(l),cdr(l))$ $=\bar r_1$:

$$\begin{array}{ll} & \text{1. unfold } \bar f(i,l),\text{ simplify} \\ = & \textbf{let } v_1=\bar f(i,cdr(l)) \textbf{ in} \\ & \textbf{if } \mathrm{arc}(i,car(l)) \textbf{ then} \\ & \qquad \textbf{let } v_2=\bar f(car(l),cdr(l)) \textbf{ in} \\ & \qquad <\max(1st(v_1),1{+}1st(v_2)),v_1,v_2> \\ & \textbf{else } <1st(v_1),\ v_1,\ \_> \end{array}$$

$$\begin{array}{ll} & \text{2. separate cases, replace applications} \\ = & \textbf{let } v_1= \textbf{if } null(cdr(l)) \textbf{ then} \\ & \qquad\qquad <0,\_,\_> \\ & \qquad\quad \textbf{else } \bar f^{\backprime}(i,cdr(l),\bar f(car(cdr(l)), \\ & \qquad\qquad\qquad cdr(cdr(l)))) \textbf{ in} \\ & \textbf{if } \mathrm{arc}(i,car(l)) \textbf{ then} \\ & \qquad \textbf{let } v_2=\bar r_1 \textbf{ in} \\ & \qquad <\max(1st(v_1),1{+}1st(v_2)),v_1,v_2> \\ & \textbf{else } <1st(v_1),\ v_1,\ \_> \end{array}$$

$$\begin{array}{ll} & \text{3. lift conditions, simplify} \\ = & \textbf{if } null(cdr(l)) \textbf{ then} \\ & \qquad \textbf{if } \mathrm{arc}(i,car(l)) \textbf{ then} \\ & \qquad\qquad <1,<0,\_,\_>,<0,\_,\_>> \\ & \qquad \textbf{else } <0,<0,\_,\_>,<0,\_,\_>> \\ & \textbf{else let } v_1=\bar f^{\backprime}(i,cdr(l),\bar f(car(cdr(l)), \\ & \qquad\qquad\qquad cdr(cdr(l)))) \textbf{ in} \\ & \qquad \textbf{if } \mathrm{arc}(i,car(l)) \textbf{ then} \\ & \qquad\qquad <\max(1st(v_1),1{+}1st(\bar r_1)),v_1,\bar r_1> \\ & \qquad \textbf{else } <1st(v_1),\ v_1,\ \_> \end{array}$$

A.3. We collect candidate auxiliary information in $\overline{llp}^{\backprime}$. It comprises the result of the function application $f(car(cdr(l)),cdr(cdr(ll)))$ and the (recursive) applications that contain this result. All of them depend only on the argument l. Thus, we obtain

$$\begin{array}{rl} l\check{l}p(l)= & \textbf{if } null(l) \textbf{ then } <\_> \\ & \textbf{else } <\check f(l)> \\ \check f(l)= & \textbf{if } null(cdr(l)) \textbf{ then } <\_,\_> \\ & \textbf{else } <\bar f(car(cdr(l)),cdr(cdr(l))),\ \check f(cdr(l))> \end{array} \qquad (5.17)$$

B.1. We merge the candidate auxiliary information $l\check{l}p$ with $\overline{llp}$. We first define

$$\overline{l\check{l}p}(l)=\ \textbf{let } \bar r=\overline{llp}(l) \textbf{ in let } \check r=l\check{l}p(l) \textbf{ in } <\bar r,\check r> \qquad \text{and} \qquad \Pi_0(\bar{\check r})=1st(1st(\bar{\check r}))$$

and then transform $\overline{l\check{l}p}(l)$. We omit the details of the transformation but just notice that if we rewrite $l\check{l}p$ in (5.17) as

$$\begin{array}{rl} l\check{l}p(l)= & \textbf{if } null(l) \textbf{ then } <\_> \\ & \textbf{else } <\check f_1(cdr(l))> \\ \check f_1(l)= & \textbf{if } null(l) \textbf{ then } <\_,\_> \\ & \textbf{else } <\bar f(car(l),cdr(l)),\ \check f_1(cdr(l))> \end{array} \qquad (5.18)$$

then it is obvious that the auxiliary information $\bar{f}(car(l), cdr(l))$ in (5.18) should be computed together with the intermediate results of $\bar{f}(n, l)$ in (5.16) regardless of whether $\mathrm{arc}(n, car(l))$ is true or false. Thus, we obtain

$$
\begin{array}{ll}
\check{\overline{llp}}(l) = & \textbf{if } null(l) \textbf{ then } <0, \_, \_> \\
& \textbf{else let } v_1 = \check{\overline{llp}}(cdr(l)) \textbf{ in} \\
& \quad \textbf{let } v_2 = \check{\vec{f}}(car(l), cdr(l)) \textbf{ in} \\
& \quad <\max(1st(v_1), 1 + 1st(v_2)),\ v_1,\ v_2> \\
\check{\vec{f}}(i, l) = & \textbf{if } null(l) \textbf{ then } <0, \_, \_> \\
& \textbf{else let } u_1 = \check{\vec{f}}(i, cdr(l)) \textbf{ in} \\
& \quad \textbf{let } u_2 = \check{\vec{f}}(car(l), cdr(l)) \textbf{ in} \\
& \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
& \qquad <\max(1st(u_1), 1 + 1st(u_2),\ u_1,\ u_2> \\
& \quad \textbf{else } <1st(u_1),\ u_1,\ u_2> \\
\end{array}
$$

and $\Pi_0(\check{\vec{r}}) = 1st(\check{\vec{r}})$

B.2. We derive an incremental version of $\check{\overline{llp}}$ under $\oplus$. We define $\check{\overline{llp}}'(i, l, \bar{r})$ to compute $\check{\overline{llp}}(cons(i, l))$ incrementally, with $\check{\overline{llp}}(l) = \check{\vec{r}}$:

$$
\begin{array}{ll}
& \text{1. unfold } \check{\overline{llp}}(cons(i, l)), \text{ simplify} \\
= & \textbf{let } v_1 = \check{\overline{llp}}(l) \textbf{ in} \\
& \textbf{let } v_2 = \check{\vec{f}}(i, l) \textbf{ in} \\
& <\max(1st(v_1), 1{+}1st(v_2)), v_1, v_2>
\end{array}
$$

$$
\begin{array}{ll}
& \text{2. replace applications, separate cases} \\
= & \textbf{let } v_1 = \check{\vec{r}} \textbf{ in} \\
& \textbf{let } v_2 = \textbf{if } null(l) \textbf{ then } <0, \_, \_> \\
& \qquad\qquad \textbf{else } \vec{f}'(i, l, 3rd(\check{\vec{r}})) \textbf{ in} \\
& <\max(1st(v_1), 1{+}1st(v_2)),\ v_1,\ v_2>
\end{array}
$$

$$
\begin{array}{ll}
& \text{3. lift conditions, simplify} \\
= & \textbf{if } null(l) \textbf{ then} \\
& \qquad <1, <0, \_, \_>, <0, \_, \_>> \\
& \textbf{else let } v_2 = \vec{f}'(i, l, 3rd(\check{\vec{r}})) \textbf{ in} \\
& \qquad <\max(1st(\check{\vec{r}}), 1{+}1st(v_2)),\ \check{\vec{r}},\ v_2>
\end{array}
$$

where we define $\vec{f}'(i, l, \check{\vec{r}}_1)$ to compute $\check{\vec{f}}(i, l)$ incrementally, with $l \neq nil$ and $\check{\vec{f}}(car(l), cdr(l)) = \check{\vec{r}}_1$:

$$
\begin{array}{ll}
& \text{1. unfold } \check{\vec{f}}(i, l), \text{ simplify} \\
= & \textbf{let } v_1 = \check{\vec{f}}(i, cdr(l)) \textbf{ in} \\
& \textbf{let } v_2 = \check{\vec{f}}(car(l), cdr(l)) \textbf{ in} \\
& \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
& \qquad <\max(1st(v_1), 1{+}1st(v_2)), v_1, v_2> \\
& \textbf{else } <1st(v_1),\ v_1,\ v_2>
\end{array}
$$

$$
\begin{array}{ll}
& \text{2. separate cases, replace applications} \\
= & \textbf{let } v_1 = \textbf{if } null(cdr(l)) \textbf{ then} \\
& \qquad\qquad <0, \_, \_> \\
& \qquad \textbf{else } \vec{f}'(i, cdr(l), 3rd(\check{\vec{r}}_1)) \textbf{ in} \\
& \textbf{let } v_2 = \check{\vec{r}}_1 \textbf{ in} \\
& \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
& \qquad <\max(1st(v_1), 1{+}1st(v_2)), v_1, v_2> \\
& \textbf{else } <1st(v_1),\ v_1,\ v_2>
\end{array}
$$

$$
\begin{array}{ll}
& \text{3. lift conditions, simplify} \\
= & \textbf{if } null(cdr(l)) \textbf{ then} \\
& \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
& \qquad <1, <0, \_, \_>, <0, \_, \_>> \\
& \quad \textbf{else } <0, <0, \_, \_>, <0, \_, \_>> \\
& \textbf{else let } v_1 = \vec{f}'(i, cdr(l), 3rd(\check{\vec{r}}_1)) \textbf{ in} \\
& \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
& \qquad <\max(1st(v_1), 1{+}1st(\check{\vec{r}}_1)), v_1, \check{\vec{r}}_1> \\
& \quad \textbf{else } <1st(v_1),\ v_1,\ \check{\vec{r}}_1>
\end{array}
$$

B.3. We prune functions $\check{\bar{llp}}$ and $\check{\bar{llp}}'$, eliminate the underscore components, and obtain

$$
\begin{array}{rl}
\widetilde{llp}(l) = & \textbf{if } null(l) \textbf{ then } <0> \\
& \textbf{else let } v_2 = \tilde{f}(car(l), cdr(l)) \textbf{ in} \\
& \quad <\max(llp(cdr(l)), 1{+}1st(v_2)),\ v_2> \\
\tilde{f}(i,l) = & \textbf{if } null(l) \textbf{ then } <0> \\
& \textbf{else let } u_2 = \tilde{f}(car(l), cdr(l)) \textbf{ in} \\
& \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
& \quad\quad <\max(f(i, cdr(l)), 1{+}1st(u_2)),\ u_2> \\
& \quad \textbf{else } <f(i, cdr(l)),\ u_2>
\end{array}
\tag{5.19}
$$

$$
\begin{array}{rl}
\widetilde{llp}'(i,l,\tilde{r}) = & \textbf{if } null(l) \textbf{ then } <1, <0>> \\
& \textbf{else let } v_2 = \tilde{f}'(i, l, 2nd(\tilde{r})) \textbf{ in} \\
& \quad <\max(1st(\tilde{r}), 1{+}1st(v_2)),\ v_2> \\
\tilde{f}'(i,l,\tilde{r_1}) = & \textbf{if } null(cdr(l)) \textbf{ then} \\
& \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then } <1, <0>> \\
& \quad \textbf{else } <0, <0>> \\
& \textbf{else let } v_1 = \tilde{f}'(i, cdr(l), 2nd(\tilde{r_1})) \textbf{ in} \\
& \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
& \quad\quad <\max(1st(v_1), 1{+}1st(\tilde{r_1})),\ \tilde{r_1}> \\
& \quad \textbf{else } <1st(v_1),\ \tilde{r_1}>
\end{array}
\tag{5.20}
$$

While computing $llp(cons(i,l))$ from scratch takes exponential time, computing $\widetilde{llp}'(i,l,\tilde{r})$ takes $O(n)$ time, where n is the length of l, since $\widetilde{llp}'(i,l,\tilde{r})$ calls $\tilde{f}'$, which goes through the list l once.

Finally, we use the derived functions for $\widetilde{llp}'$ to compute the original function llp. Note that $llp(l) = 1st(\widetilde{llp}(l))$ and, if $\widetilde{llp}(l) = \tilde{r}$, then $\widetilde{llp}'(i,l,\tilde{r}) = \widetilde{llp}(cons(i,l))$. Using the definition of $\widetilde{llp}'$ in (5.20) in this last equation, we obtain:

$$
\begin{array}{rl}
\widetilde{llp}(cons(i,l)) = & \textbf{if } null(l) \textbf{ then } <1, <0>> \\
& \textbf{else let } \tilde{r} = \widetilde{llp}(l) \textbf{ in} \\
& \quad \textbf{let } v_2 = \tilde{f}'(i, l, 2nd(\tilde{r})) \textbf{ in} \\
& \quad <\max(1st(\tilde{r}), 1 + 1st(v_2)),\ v_2>
\end{array}
$$

Using this equation and the base case $\widetilde{llp}(nil) = <0>$, we obtain a new definition of $\widetilde{llp}$:

$$
\begin{array}{rl}
\widetilde{llp}(l) = & \textbf{if } null(l) \textbf{ then } <0> \\
& \textbf{else if } null(cdr(l)) \textbf{ then } <1, <0>> \\
& \textbf{else let } \tilde{r} = \widetilde{llp}(cdr(l)) \textbf{ in} \\
& \quad \textbf{let } v_2 = \tilde{f}'(car(l), cdr(l), 2nd(\tilde{r})) \textbf{ in} \\
& \quad <\max(1st(\tilde{r}), 1 + 1st(v_2)),\ v_2>
\end{array}
\tag{5.21}
$$

where $\tilde{f}'$ is defined in (5.20). This new $\widetilde{llp}$ takes $O(n^2)$ time, since it calls $\tilde{f}'$ only $O(n)$ times.

5.5 Related work

Work related to our analysis and transformation techniques has been discussed in Section 5.3. Here, we take a closer look at related work on discovering auxiliary information for incremental computation.

Interactive systems and *reactive systems* often adopt incremental algorithms to achieve fast response time [DGHKL84,Rei84a,BS86,HN86,BC88,RT88,Kai89,BGV92]. Since explicit incremental algorithms are hard to write and appropriate auxiliary information is hard to discover, the general approach in this chapter provides a helpful systematic method for developing particular incremental algorithms. For example, for the dynamic incremental attribute evaluation algorithm in [RTD83], the characteristic graph is a kind of auxiliary information that would be discovered following the general principles underlying our approach. For static incremental attribute evaluation algorithms [Kas80,Kat84], where no auxiliary information is needed, the approach can cache intermediate results and maintain them automatically, as described in Chapter 4.

Strength reduction [All69,CK77,ACK81,SKR91] is a traditional compiler optimization technique that aims at computing each iteration incrementally based on the result of the previous iteration. Basically, a fixed set of strength-reduction rules for primitive operators like times and plus are used. Our method can be viewed as a principled strength reduction technique not limited to a fixed set of rules: it can be used to reduce strength of computations where no given rules apply and, furthermore, to derive or justify such rules when necessary, as shown in the integer square root example.

Finite differencing, proposed by Paige *etc.* [PS77,Pai81,PK82], can be regarded as a generalization of strength reduction to set-theoretic expressions for systematic program development. Basically, rules are manually developed for differentiating set expressions. For continuous expressions, our method can derive such rules directly using properties of primitive set operations. However, set expressions can be discontinuous, in which case corresponding dynamic expressions need to be discovered and rules for maintaining them derived. These dynamic expressions are composed of several kinds of maps and are a certain kind of auxiliary information. How to discover them is still a problem that needs to be studied, but once discovered, our method can at least be used to derive rules that maintain this information. Also, in general, Paige's rules apply only to very-high-level languages like SETL; our method applies also to normal high-level languages like Lisp.

Maintaining and strengthening loop invariants has been advocated by Dijkstra, Gries, and others [Dij76,Gri81,Rey81,Gri84] for almost two decades as a standard strategy for developing loops. In order to produce efficient programs, loop invariants need to be maintained by the derived programs in an incremental fashion. To make a loop more efficient, the strategy of strengthening a loop invariant, often by introducing fresh variables, is proposed [Gri84]. This corresponds to discovering appropriate auxiliary information and deriving incremental programs that maintain such information. Work on loop invariants stressed mental tools for programming, rather than mechanical assistance, so no systematic procedures were proposed.

Induction and generalization [BM79,MW93] are the logical foundations for recursive calls and iterative loops in deductive program synthesis [MW80] and constructive logics [C$^+$86]. These corpora have for the most part ignored the efficiency of the programs derived, and the resulting programs "are often wantonly wasteful of time and space" [MW92]. In contrast, the approach in this thesis is particularly concerned with the efficiency of the derived programs. Moreover, we can see that induction, whether course-of-value induction [Kle52], structural induction [Bur69,BM79], or well-founded induction [BM79,MW93], enables derived programs to use results of previous iterations in each iteration, and generalization [BM79,MW93] enables derived programs to use appropriate auxiliary information by strengthening induction hypotheses, just like strengthening loop invariants. The approach in this thesis may be used for systematically constructing induction steps [Kle52] and strengthening induction hypotheses.

The *promotion and accumulation strategies* are proposed by Bird [Bir84,Bir85] as general methods for achieving efficient transformed programs. Promotion attempts to derive a program that defines $f(cons(a,x))$ in terms of $f(x)$, and accumulation generalizes a definition by including an extra argument. Thus, promotion can be regarded as deriving incremental programs, and accumulation as identifying appropriate intermediate results or auxiliary information. Bird used two nice examples to illustrate the general strategies, with the help of succinct notations. However, we can discern no systematic steps being followed in [Bir84]. As demonstrated with the path sequence problem, our approach can be regarded as a systematic formulation of the promotion and accumulation strategies. As such, it helps avoid the kind of errors reported and corrected in [Bir85].

Other work on transformational programming for improving program efficiency, including the extension techniques in [Der83], the transformation of recursive functional programs in the CIP project [Bro84,BMPP89,Par90], and the finite differencing of functional programs in the semi-automatic program development system KIDS [Smi90,Smi91], can also be further automated with our systematic approach. Such a general systematic approach to program improvement is important to the area of program development from specification since it helps separate efficiency concerns from development of executable programs.

So far, we have presented not just a method for discovering auxiliary information, but a comprehensive approach for efficient incremental computation. The approach forms the basis of a general methodology for efficient computation and thus program improvement, which is one of the most important issues in program development and maintenance.

Chapter 6

CACHET: An interactive program transformation system based on the incremental attribution paradigm

To help apply the systematic approach for deriving incremental programs and further establish its feasibility, automatic and semi-automatic tools need to be built. A prototype system, CACHET, has been implemented for semi-automatic derivation of incremental programs using the analyses and transformations described in the previous chapters.

This chapter describes the design and implementation of CACHET as an interactive program transformation system based on the incremental attribute evaluation framework.

6.1 Introduction

Program transformation systems are important tools that implement various program manipulations that preserve program semantics and improve program performance [HL78,Fea82,PS83,Fea87,Red88,BMPP89,Par90,Smi90,Pai94].

Interactive program transformation. Program transformation systems are often required to be interactive, for at least the following two reasons. First, the goal of a program transformation system is often so ambitious that no fully automatic transformation can succeed; interaction allows convenient semi-automatic transformation. Second, the study of transformation techniques itself often consists of trying different transformations and requires much tedious program rewriting; interactive invocation of these transformations provides control during experimentation.

The usability of an interactive program transformation system depends greatly on its interface. However, designing and implementing such an interface, especially with various program manipulation functions, is a heavy task.

Incremental program analysis. For any non-trivial program transformation, substantial program analysis is needed. The analyses determine various dependencies, types, unique identifiers, *etc.* The results of the analyses are used to decide, or to help a user to decide, what transformations to apply and where and how to apply them. After each transformation step, the analyses need to be done again on the transformed program. Such interleaving of analyses and transformations continues until certain criteria for stopping are satisfied.

Each program transformation step typically changes only a small portion of the program being transformed; thus one should be able to update the results of program analyses to reflect the change. This is the problem of incremental program analysis under program transformation. Solving this problem is indispensable for speeding up the program analyses and thus the overall program transformations.

However, until now, all program transformation systems we are aware of use some *ad hoc* inference engine to do the analyses, store the results of the analyses in some database, and use some rewrite engine to do the transformations. The simple storage of the results of the analyses makes it hard to incrementalize the analyses under the transformations. This creates a performance bottleneck that is more severe for more automated systems [Pai94].

Using an attribute-grammar-based programming environment. Interactive program transformation and incremental program analysis naturally lead one to consider an attribute-grammar-based programming environment. The Synthesizer Generator is a commercially available system that generates such environments [RT88].

With such a tool, the syntax of programs can be described using a context-free grammar, and properties of programs can be described using attribute equations. In such a *declarative* framework, program analysis is performed by program tree attribution, program transformation directly mutates the program tree, and incremental analysis is conducted by incremental attribute evaluation after each transformation step.

CACHET. CACHET is an incremental-attribution-based interactive program transformation system for programs written in a first-order functional language. It is implemented using the Synthesizer Generator, with extensions to support complex tree transformations. Incremental program analysis is performed by incremental attribute evaluation, provided automatically by the Synthesizer Generator.

Attribute grammars and incremental attribute evaluation methods have been well addressed in the literature [DJL88] for describing static semantics of programs. They are not the subject of this chapter. Instead, this chapter describes how to adopt a traditional programming environment and make it suitable for performing complex tree transformations, interleaving transformations with external inputs, such as user inputs, and making more use of attribution mechanisms.

CACHET has special functionality for deriving incremental programs. It has been used to derive numerous incremental programs, including most of the examples in Chapters 3, 4, and 5. It has also been of great help in studying transformations for caching intermediate results and discovering auxiliary information.

The rest of the chapter is organized as follows. Section 6.2 gives an overview of the desired features, the problems to be solved, and the suggested solutions. Section 6.3 describes the implementation techniques used for CACHET. Section 6.4 contains a sample derivation of an incremental program using CACHET. Section 6.5 discusses related work. Section 6.6 discusses future work.

6.2 System design

This section discusses desired features, problems to be solved, and suggested solutions for adopting a traditional programming environment for program transformations.

6.2.1 Complex tree transformation

To enable program transformations, the major power that needs to be added to a traditional attribute-grammar-based framework is a metalanguage for complex tree transformations.

The mechanisms needed for tree transformations are mainly pattern matching and pattern instantiation. Languages designed specifically for specifying complex tree transformations [Hec88], which may include sophisticated pattern matching languages with, for example, second-order patterns [HL78], can greatly facilitate programming various transformations.

Transformations should be able to access attributes associated with the tree, since they are often conditioned on the results of program analyses, which are performed by tree attribution. Providing direct access to tree nodes and associated attributes at non-local places can save programming effort as well as run-time space, since otherwise extra attributes are needed to propagate information along the tree.

With pattern matching and instantiation, and with access to attributes, we can program various transformations, such as fold-unfold, simplification, specialization, and transformations enabled by equality analysis. Some transformations require more complicated treatment; in particular, the function introduction with generalization in Section 3.3 involves suspending transformation on the current tree and preparing a new subtree for recursive transformations.

Finally, a metalanguage for complex tree transformations should include a *rewrite* engine that can apply a set of transformations repeatedly to a subtree in a certain traversal order. With such an engine, repeated invocation of transformations in certain fixed patterns can be easily automated, saving both programmers' and users' effort. Higher-order term rewrite [Pau83] offers a framework for defining such rewrites. What needs to be provided is the ability to automatically interleave such repeated rewrite with incremental attribute evaluation.

6.2.2 External input as annotation

Program transformations are often semi-automatic and involve interaction with users or other external facilities, such as theorem provers. Input from such external sources is called *external input*. The role of external input is to provide transformations with information that does not depend solely on the program tree or is too inconvenient or too expensive to compute completely from the program tree.

As program transformation generalizes symbolic, and thus normal, executions of programs, external input includes dynamic information that can be set during program executions, e.g., the breakpoints in a debugger or the instruction pointer in an interpreter. Such dynamic semantics [Kai89] is needed for run-time support, symbolic debuggers, interpreters, as well as interactive program transformation systems.

External input scattered in the middle of program execution or transformation is a new concept lacking in traditional declarative attribute-grammar-based environments. We describe a corresponding new notion called *annotation* that fits well into the attribute-grammar-based framework for program transformation, preserving the declarative nature.

Annotations. Annotations should not be part of the program tree, since they do not represent terms of the subject language; nor should they be conventional attributes, since they are not determined solely by the program tree, as conventional attributes are. However, annotations should be associated with a particular position in the tree. Moreover, they should be accessible for defining attributes, since external input provides information to the analyses that guide the transformations, and the analyses are done by attribution.

To implement annotations, we can put them directly in the tree, and thus attributes can be defined solely on the tree, as before. However, some distinction between annotations and subject language terms is needed to facilitate programming with tree pattern matching on the subject language.

An annotation may be more conveniently implemented as a special kind of attribute whose existence and, perhaps, value are determined by external input, as opposed to completely determined by the term tree or the attribution of the term tree. Of course, a user can define the value of an annotation using anything, including attributes.

The term *annotation* is used by Reiss in his FIELD environment [Rei90a,Rei90b] as the primary mechanism for interacting with the source file in an editor; for example, a breakpoint in the debugger is associated with an annotation in the editor. In the current system CACHET, annotations are used to store expensive attributes whose values are computed only upon user request.

6.2.3 Tree attribution mechanism

In addition to incremental attribute evaluation, as introduced in Section 6.1, a number of other tree attribution capabilities are desired to facilitate program transformations.

Circular attribute evaluation. Some program analyses are based on fixed point iteration. If such analyses are done by attribute evaluation, then circular attribute evaluation methods are needed. Although solutions to this problem have been proposed [Far86,Jon90], they are rarely implemented due to complications and inefficiency. Here, we briefly describe how circular attribute evaluation can be simulated using tree transformations and annotations.

Basically, one defines a circular attribute as a function of the program tree and an annotation, where the annotation stores the value of the circular attribute from the previous iteration, starting from bottom. A user can compare the value of the attribute and the annotation and, if they are not the same, set the annotation to be the value of the attribute, which then triggers the incremental attribution. This can be repeated until the value of the circular attribute and the annotation are equal. This iteration can be automated with rewrite, saving the user from intervening in each iteration.

Modular attribute evaluation. Heavy program transformations are usually composed of separate phases, where each phase conducts relevant, and often different and smaller, program analyses and performs lighter transformations. Several approaches have been proposed for modular attribute specification [GG84,DC90,Far92,FMY92, BG94] and modular attribute evaluation [GG84,FMY92].

Modular specification improves the readability of attribute grammars and allows more convenient modular evaluation. Modular evaluation provides the flexibility of turning on and off attribution modules as necessary, which results in evaluators that are speedier and more storage-efficient [FMY92]. They are particularly useful for phase-based transformations. In particular, for phases not involving circular attributes, efficient evaluators for non-circular attribute grammars can be generated.

6.2.4 Replay

We want to consider replaying transformations. A minimal approach is to record the *history* or *script* of external input [Red88]. Powerful metalanguages can help reduce the recording work [Fea82]; for example, with a metalanguage that allows rewrite, we can record one rewrite in place of a sequence of transformations involved in the rewrite. Replay is important not only for helping to understand the whole transformation, but also for incremental transformation under changes to the input program. Whether it is feasible to achieve such incrementality in practice still needs to studied. An alternative to the direct tree manipulation framework for the purpose of replay is discussed in Section 6.6.

6.3 Implementation

A prototype system, CACHET, based on the design principles in the previous section, has been implemented. It uses the Synthesizer Generator [RT88], a system for generating language-based editors, and consists of about 18,000 lines of code written in SSL, the Synthesizer Generator language for specifying editors. CACHET is interactive: its flexible editor interface is generated automatically by the Synthesizer Generator.

6.3.1 Building in transformation

Source-to-source transformations are operations built in to CACHET. The Synthesizer Generator provides the functionality of defining *transforms* with first-order pattern matching, tree mutation, and access to attributes. It has been easily used to implement all but one of the basic program transformations, including fold-unfold, simplification, lifting a subexpression, and transformations that may be enabled by equality analyses. Some complex combinations of basic transformations, such as extensive simplification and specialization, have also been coded directly as transforms.

The one basic transformation not so easily implemented is function introduction with generalization, where, in the middle of transforming a subtree, we need to establish a new tree, switch over to transform the new tree recursively, and switch back to the original subtree when the recursion returns. A new *input* facility has been added to the Synthesizer Generator to recursively invoke new program transformation buffers for this purpose.

Currently, the derivations are semi-automatic, since they are composed mostly of basic transformations that are invoked manually. Manual invocation is used mainly for two reasons. First, the Synthesizer Generator allows only subtree replacement and does not currently have a rewrite engine for a fully-automatic exhaustive application of basic transformations, in particular, for applicative-order reduction, as specified by the incrementalization approach in Chapter 3. Second, we want an interactive environment to study various transformations; thus manual invocation is suitable most of the time. Also, at present, we are only using a simple equality reasoning engine, not a full-blown theorem prover.

Mechanisms for defining combinations of transformations, especially rewrite mechanisms, are being added to the Synthesizer Generator to further automate derivations. We also plan to enrich the transformation language with second-order pattern matching and easy non-local access to make it a more convenient tool for describing complex transformations.

6.3.2 Simulating annotation

Annotations are currently implemented as special parts in the program tree that are, by default, not displayed together with terms in the subject language, though they can be displayed upon request by the user.

One major issue that needs to be addressed is the validity of annotations under program tree transformations and other editing operations like cut and paste. The current implementation provides special transforms to manipulate annotations: annotations whose validities are not limited to the context in the program tree can be elevated to the root of the tree; annotations that are valid only within its context can simply be eliminated at the user's request.

We chose this current implementation strategy because it does not require new features in the Synthesizer Generator. But we plan to extend the Synthesizer Generator to implement annotations as special attributes, provide mechanisms to specify the validity conditions of annotations, and automate the elimination, elevation, or possibly other treatments of annotations. This would make the Synthesizer Generator a more suitable tool for describing dynamic semantics with user interaction and other external input.

6.3.3 Using attribution

Currently, attributes in CACHET are used mainly for propagating global information, collecting context information, analyzing dependencies, and reasoning about equalities. The values of these attributes can be displayed any time, as facilitated by the Synthesizer Generator [RT88], to help a user understand the derivation. Of course, these attributes are evaluated incrementally after each transformation step.

Methods for circular attribute evaluation still need to be implemented, either by extending the Synthesizer Generator or using the simulation proposed in the previous section. So far, the need for modular attribute specification and implementation is not strong, since we have implemented only the derivation approach in Chapter 3 and the transformations for caching all intermediate results in Chapter 4. But we expect that it would be very helpful when we implement the full-blown approaches for exploiting intermediate results as in Chapter 4 and auxiliary information as in Chapter 5, since they are strongly phase-oriented.

We believe that these implementation issues form a very promising area. Program tree attribution provides a declarative framework for program analyses, which are needed to guide powerful program transformations. Techniques for managing the interactions among tree transformation, external input, and incremental attribution can be used to generate powerful and incremental transformation systems.

6.4 Viewing an example derivation of an incremental program

CACHET is designed specifically for deriving incremental programs. It has been used to derive numerous incremental programs, including most of the examples in Chapters 3, 4, and 5.

The programs transformed by CACHET are written in a first-order functional language, where expressions in function definitions are composed of variables, data constructions, primitive function applications, user-defined recursive function applications, conditional expressions, and binding expressions.[1] An example is given in the back window of Figure 6.1. It defines a function `sort`, which does selection sort, and two auxiliary functions `least` and `rest`. The input change operation is adding

[1]The syntax for binding expressions implemented in CACHET, **let** $v = e_1$ **in** e_2 **end**, is slightly different from the syntax used in previous chapters, **let** $v = e_1$ **in** e_2.

a new element `i` to an old input `x` to form a new input `cons(i,x)`. We will derive an incremental program `sort'` that sorts `cons(i,x)` by inserting `i` into the sorted list `sort(x)`.

The basic idea for incrementalization is to transform `sort(cons(i,x))` and replace subcomputations whose values are retrievable from the value of `sort(x)` with corresponding retrievals. The derivation introduces incremental functions to compute function applications, puts them in a *definition set*, and uses them to replace the original applications. To obtain the definition of such an introduced function, the derivation unfolds the corresponding function application, collects context information, and simplifies subexpressions. It also finds subcomputations, in their respective contexts, whose values are retrievable from the cached result of the previous computation, puts them in the corresponding *cache sets*,[2] and replaces occurrences of such subcomputations with corresponding retrievals.

Given a set of `FUNCTION DEFINITIONS`, the `FUNCTION TO BE EVALUATED`, the `OLD INPUT TO THE FUNCTION`, and the `NEW INPUT TO THE FUNCTION`, the `EXPRESSION TO BE TRANSFORMED` is initialized by a transform `init-tran-exp`. We select the expression to be transformed, shown by the underlined expression in the back window of Figure 6.1. Relevant transforms for the current selection are displayed in buttons at the bottom of the window.

Transforms. A transform ending with * is a basic transformation that preserves correctness. In particular, a transform ending with `$*` uses the cached result of the previous computation. A transform ending with `!` is a combination of correctness-preserving basic transformations, and one ending with `?` is a basic transformation that may not preserve correctness (enabled for experimentation).

Transforms starting with `.` deal with annotations. In particular, transform `.f_e(fun-intr-repl)*` introduces a new function to compute the current function application or replaces the current function application with a previously introduced function; any change to the definition set leaves the new set as an annotation at the current selection. Transform `.AUX-specialize$` uses an auxiliary specializer to extend, at the current selection, the set of subcomputations whose results are retrievable from the cached result of the previous computation; the resulting cache set is left as an annotation at the current selection. Other transforms starting with `.` but with no special ending symbols manipulate annotations by elevation or elimination.

Names of transforms attempt to be illustrative. Names starting with `c`, `p`, and `f` are for applications of constructors, primitive functions, and user-defined functions, respectively. Names starting with `if` and `let` are for conditional and binding expressions, respectively. Subscript `_e` denotes a transformation like simplification, and `_r` denotes a replacement with a cached result. The strings in parentheses are for further illustration, e.g., `(de-con)` denotes destructing a construction using a selector, `(unfold-funEnv)` denotes unfolding a function in the given set of functions,

[2] Rather than associating a cache set for each unfolded function application and associating elements of the cache set with their valid information sets, as described in Chapter 3, the implemented cache sets are associated with occurrences of expressions in the unfolded application, so that their valid information sets are just the information sets associated with the occurrences.

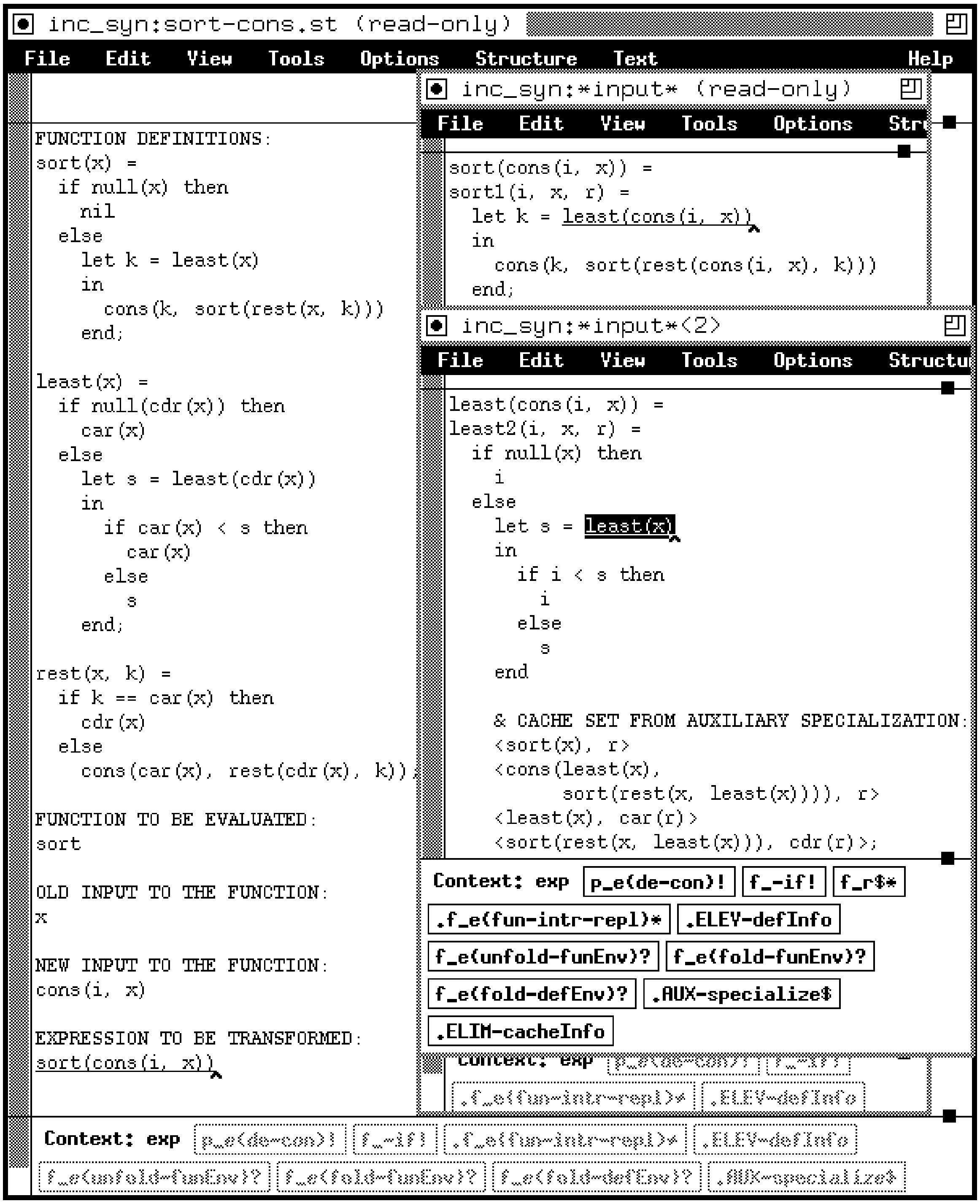

Figure 6.1: Derivation of the example

and `(fold-defEnv)` denotes folding into an introduced function in the definition set. Subscript `_n` refers to the *n*th subexpression; `-if` and `-let` lift the condition and the binding, respectively, of the *n*th subexpression out of the enclosing expression; omitting a subscript `n` implies lifting all conditions or bindings.

Function introduction. With the underlined selection in the back window of Figure 6.1, clicking on transform `.f_e(fun-intr-repl)*` suspends transformation in the current window, pops up a new window, the middle window of Figure 6.1, and switches over for recursive transformations that will yield a function `sort1` to replace the application `sort(cons(i,x))`. Note that a fresh identifier `r` is used as parameter for the cached result of `sort(x)`.

Unfolding and simplification. To obtain a definition of `sort1`, in the middle window of Figure 6.1, we unfold the application `sort(cons(i,x))` and apply simplifications `p_e(null-cons)*` (transforming `null(cons(i,x))` to `false`) and `if_e(cond-false)*` (transforming `if false then e1 else e2` to `e2`). We obtain what is shown in the middle window. Then, recursively, we introduce a function `least2`, in the front window of Figure 6.1, to compute `least(cons(i,x))`. After unfolding and applying simplification `p_e(de-con)!` (performing all destructions of constructions), we obtain the upper part of the front window.

Annotation of cache set and auxiliary specialization. The lower part of the front window shows an *annotation*, namely, the cache set at the branch where `null(x)` is false. It is the result of applying auxiliary specialization to `sort(x)` with `null(x)` being false at that branch, and is obtained by clicking `.AUX-specialize$` at the bottom of the window and specializing `sort(x)` in another window (killed upon returning the displayed cache set). The cache set indicates that the value of `sort(x)` is `r`, the value of `cons(least(x),sort(rest(x,least(x))))` (specialized `sort(x)` when `null(x)` is false, by definition of `sort`) is `r`, the value of `least(x)` is `car(r)`, *etc.*

Replacement. With the displayed cache set, when we select the underlined and highlighted expression `least(x)` in the front window, we find that rule `f_r$*` applies. Clicking on it replaces `least(x)` with `car(r)`. Similarly, the boolean expression `null(x)` can be replaced with `null(r)`, essentially because `null(x)` is true if and only if `null(sort(x))` is true.

Function replacement. The above replacements yield the definition of function `least2(i,x,r)` shown on the left of Figure 6.2.[3] Then, we return to resume the transformation for `sort1`, and we replace the application `least(cons(i,x))` with `least2(i,x,r)`.

Annotation of definition set. A definition set is now annotated at selection `least2(i,x,r)`, as displayed in Figure 6.3. The tags at the bottom of each definition indicate whether the definition is done, whether it is recursive, and whether it uses a cache argument. While the definition of `least2` is done and not recursive; the definition of `sort1` is *yet* not done and not recursive, and thus it is kept

[3]We skip the dead code elimination here, even though the parameter `x` of `least2(i,x,r)` is dead and can be eliminated. This does not affect the subsequent derivation of this example.

```
if null(r) then
  i
else
  let s = car(r)
    in
      if i < s then
        i
      else
        s
    end
```

```
if s == i then
  x
else
  cons(i, rest(x, s))
```

Figure 6.2: Intermediate function definitions

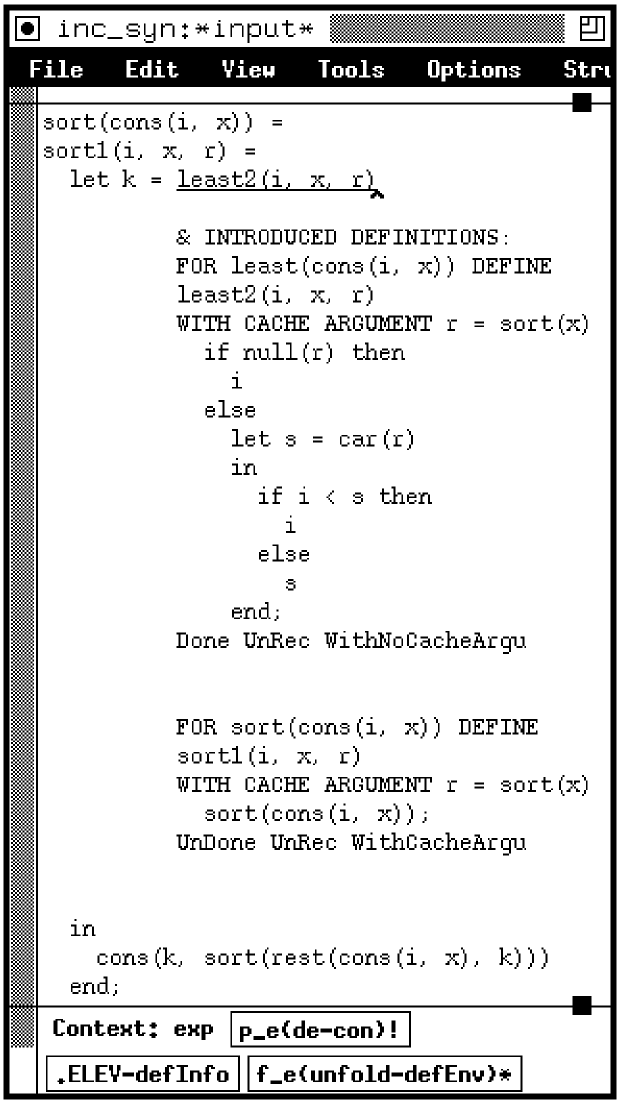

Figure 6.3: Derivation of the example (continued 1)

as `sort(cons(i,x))`. An annotation of a definition set at a subexpression can be elevated to its root by selecting the root expression and clicking on transform `.ELEV-defInfo`; for example, the displayed definition set can be elevated to its outside binding expression.

Application `least2(i,x,r)` is unfolded in place, since `least2` is not recursively defined. We then lift the conditionals and binding in the definition of `least2` out of the enclosing expression, simplify the resulting binding expressions, and obtain the result shown on the left of Figure 6.4.

```
sort1(i, x, r) =
  if null(r) then
    cons(i, sort(rest(cons(i,x),i)))
  else
    let s = car(r)
    in
      if i < s then
        cons(i, sort(rest(cons(i,x),i)))
      else
        cons(s, sort(rest(cons(i,x),s)))
    end;
```

```
sort1(i, x, r) =
  if null(r) then
    cons(i, r)
  else
    let s = car(r)
    in
      if i < s then
        cons(i, r)
      else
        if s == i then
          cons(s, sort(x))
        else
          cons(s, sort(cons(i,rest(x,s))))
    end;
```

Figure 6.4: Derivation of the example (continued 2)

Then, similarly, we introduce a function to compute `rest(cons(i,x))` in the first branch, replace it, and unfold to obtain just `x`.[4] With this, the enclosing expression becomes `sort(x)` and can just be replaced by `r`. We elevate the definition set containing the function just introduced so that the next occurrence of `rest(cons(i,x))` can use it and obtain `x` directly. Again, the enclosing `sort(x)` can be replaced with `r`. For `rest(cons(i,x),s)` in the last branch, introducing a function, replacing, and unfolding, we obtain the result shown on the right of Figure 6.2. Lifting the condition out of the enclosing applications of `sort` and `cons`, we obtain the result shown on the right of Figure 6.4.

Next, application `sort(x)` is replaced by `r`, and the two branches with conditions `i<s` and `s==i` are merged, yielding `if i<=s then cons(i,r)`. For the last branch, `sort(cons(i,rest(x,s)))` is replaced with recursive call `sort1(i,rest(x,s), sort(rest(x,s)))`. With the knowledge of the context information `s` = `car(r)` = `least(x)`, and using the cache set displayed in the front window of Figure 6.1, we see that `sort(rest(x,s))` = `sort(rest(x,least(x)))` = `cdr(r)`. Thus, we obtain the result shown on the left of Figure 6.5.

[4]This is different from the derivation in Chapter 3 since it does not use the context information that `x` is `nil`. But it allows us to show that the next application of `rest` can use the function thus introduced.

Dead code elimination. To complete the example, we need to eliminate dead code in the above definition. In particular, the second parameter of `sort1` is dead and should be eliminated. This is not implemented yet. It can be done using traditional compiler technologies, after which we would obtain the result shown on the right of Figure 6.5, which is exactly an insertion. Finally, the original application `sort(cons(i,x))` can be replaced by `sort'(i,r)`, where `r` = `sort(x)`.

<pre>sort1(i, x, r) =
  if null(r) then
    cons(i, r)
  else
    let s = car(r)
    in
      if i <= s then
        cons(i, r)
      else
        cons(s, sort1(i,rest(x,s),cdr(r)))
    end;</pre>
	<pre>sort'(i, r) =
	  if null(r) then
	    cons(i, r)
	  else
	    let s = car(r)
	    in
	      if i <= s then
	        cons(i, r)
	      else
	        cons(s, sort'(i, cdr(r)))
	    end;</pre>

Figure 6.5: Derivation of the example (continued 3)

The derivation above is performed by manually selecting the subexpression and clicking on one of the enabled transforms. After we implement the rewrite engine, we will be able to automate the derivation using an applicative-order rewrite, as specified by the derivation approach in Chapter 3. However, to see all the interesting intermediate results during the derivation, manual invocation is appropriate.

6.5 Related work

Program transformation systems and the approaches and techniques used therein are described in a number of surveys, e.g., [PS83,Fea87]. Eminent systems among them and recent systems include APTS [Pai94], KIDS [Smi90,Smi91], CIP [BMPP89, Par90], Focus [Red88], and ZAP [Fea82].

Compared to these systems, the most important and unique characteristic of CACHET is its use of attribute grammars. This has at least two advantages. First, a program transformation system based on program analysis is a complex constraint system; the attribute grammar paradigm provides a declarative framework where constraints can be specified and consistency can be maintained in a clean way. Second, incremental program analysis is important for speeding up the overall program transformation process; using attribute evaluation for program analysis allows us to take advantage of known techniques for incremental attribute evaluation. To our knowledge, the BMF-editor [VVF90] is the only program transformation system that uses the incremental attribution paradigm, although incremental semantic analysis is desirable in all these systems, especially for the more automatic approaches in APTS

[Pai94]. The BMF-editor emphasizes dynamic transformations but does not explicitly address the problem of external inputs.

Since CACHET is implemented using a language-based editor generator, it has a flexible interactive user interface, as is provided by the existing technologies of programming environments. Among the systems above, KIDS [Smi90,Smi91] seems to be the only one with such a flexible user interface. Of course, implementing CACHET as a programming environment also facilitates the integration of program derivation and validation with interactive editing, compiling, debugging, and execution.

CACHET can benefit from a stronger metalanguage, such as that pioneered by ZAP [Fea82], and certain replay functionality, such as that in Focus [Red88]. As discussed in Sections 6.2 and 6.3, a metalanguage for powerful tree transformation has been designed and is being implemented for CACHET. How to integrate replay and incremental replay in an attribute grammar framework with annotation is a problem to be studied.

How does CACHET compare to traditional programming environments that do dataflow analysis and code generation? First, if such an environment is viewed as a program transformation system based on program analysis, then CACHET is more general in that it allows interleaving program transformations with annotations of external inputs. Techniques that address such dynamic program semantics have been lacking in traditional attribute-grammar-based programming environments [Kai89]. For example, OPTRAN [LMW88] is an attribute-grammar-based system extended with rewrite mechanisms. However, it is still a batch-oriented system mainly for compiler applications rather than general program transformations.

Another difference is that traditional attribute-grammar-based programming environments perform code generation by attribution [GG84,RT88,BG94], while CACHET transforms programs by direct manipulation. How to do program transformation by attribution in the presence of annotation is related to incremental replay, as discussed in Section 6.6.

Finally, all of the above systems are for transformations from specifications to programs, or from programs to more efficient programs, whereas CACHET has the special functionality of deriving incremental programs. This functionality provides a general solution to the finite differencing problem, which must be addressed in program derivation from specification and program improvement in general [PK82, BMPP89,Par90,Pai90,Smi90,Smi91]

6.6 Future work

A number of problems have been suggested that need to be further studied. We discuss them here.

Tree rewrite. First, the Synthesizer Generator is being interfaced with Scheme, which will allow us to implement a more powerful metalanguage with rewrite engines in Scheme. However, arbitrary rewrite by itself may not terminate. Interleaving rewrite with attribution may introduce additional circularity. Actually, the circular attribute evaluation method proposed in Section 6.2 makes use of such circularity. How to control circularity and guarantee termination is a direction for future study.

Annotation. Annotation provides a *declarative* framework for describing dynamic semantics of programs that has been lacking. However, formal description of annotation in the context of attribute-grammars is needed. An example of a question to be answered is: while attributes are associated with non-terminals and attribute equations with productions, should annotations be associated with productions and/or non-terminals?

Regarding the validity of annotations under tree transformation or other editing operations, we anticipate a number of issues. Should annotations be associated with tree nodes or locations in the tree? When two valid annotations are present at one place as a result of tree rewrite, should one overwrite the other or should they be merged; if the former, which overwrites which; if the latter, how? Should invalid annotation be tagged as invalid or simply removed? One way to answer these questions is to treat annotations like attributes, and thus treat the validity as attribute re-evaluation but perhaps simpler. Other approaches may consider features like grouping some annotations together to decide their validity.

Finally, we need to study incremental algorithms that combine annotation validation with attribute evaluation. For example, if we treat validity as attribute re-evaluation, then we need to first check the well-definedness of the attribute grammar when repeated tree rewrite is allowed. In this case, we need to analyze even the external functions that define the annotations.

Circular attribute evaluation. We need to study and compare the solution to circular attribution in this chapter with previous ones, and weigh the pros and cons of using annotation. In particular, all these methods are based on dynamic dependencies between attributes, which have substantial interpretive overhead in implementation. A static evaluation method is lacking.

Incremental replay. A potential alternative for accommodating replay is to change our basic framework and conduct transformation by attribution instead of direct tree manipulation. Thus, tree attribution is used not only for semantics analysis, but also for recording transformed versions of the program.

Attribution has been traditionally used in programming environments for code generation [RT88]; it has also been proposed for general program transformation, including phase-based transformation. Approaches include attribute coupled grammars [GG84], higher-order attribute grammars [VSK89], composable attribute grammars [FMY92], and simple tree attributions [BG94]. Incremental attribute evaluation algorithms for these frameworks could be used for incremental code generation. However, these frameworks do not address external input. We need to extend them and study how annotations should be incorporated with incremental attribute evaluation for incremental program transformation.

Programming effort. Finally, we want to avoid the duplication in programming effort in attribution rules and transformation rules. Notice that both kinds of rules are based on pattern-matching the expressions in the subject language, except that attribution rules are typically grouped together by productions, i.e., expression patterns, while transformation rules are grouped together by functionalities. A few frameworks have been proposed for specifying attribution rules grouped around functionalities and, in particular, attribute grammars can be generated from such specifications [DC90]. Mechanisms that can also compose functions from attribution rules would be very helpful.

Chapter 7

Conclusion

Widespread applications of incremental computation have demonstrated the importance of the subject; but its extensive treatment in the literature has been unsatisfactory: too simple or too diverse classification and analysis, *ad hoc* comparison with each other, unclear applicability to various application domains, *etc.* Therefore, there has been a need of general theories and methods for incremental computation. Although there have been many attempts to provide such general approaches [Ear76, Pai81,PK82,Pai84,HT86,CP89,PT89,FT90,Smi90,Smi91,YS91,SH91,Sun91,Hoo92, van92,Fie93], none has incorporated many others. After all, the subject still suffered from the lack of a general framework and a generally applicable systematic approach.

What is also unsatisfactory is the treatment of the relationship between incremental computation and program efficiency improvement in general. It is well known that the program optimization technique strength reduction utilizes the principle of incremental computation. However, does incremental computation provide just one of many kinds of program optimizations? Then why could Paige derive extremely efficient programs using finite differencing—strength reduction generalized for SETL [PS77,PK82,Pai83,Pai86,PT87,CP89,Pai94]? Is it all a result of SETL being a very-high-level language? After all, finite differencing appears only as a set of syntactic rules; its underlying semantics has not been explicitly stated.

This thesis describes a general framework that addresses fundamental aspects of incremental computation. A general systematic approach based on this framework is given for deriving incremental programs from any given programs f and input change operations $\oplus$, by analyzing and transforming $f(x \oplus y)$. The generality of the approach makes it directly applicable to any program efficiency improvement by allowing program iterations to be computed using appropriate incremental programs. This exposes the underlying fact that incremental computation is the essence of improving the efficiency of computations.

We believe that the whole approach establishes a fundamental methodology for program improvement in software development and maintenance. As a matter of fact, the intended usage of the approach covers a spectrum of applications. At the one extreme, the approach provides an off-line methodology for algorithm design and program improvement. At the other extreme, a fully-automatable subset of the analyses and transformations offers a comprehensive "strength reduction" optimization for language compilers.

7.1 Summary

Three major ideas in this thesis form the core of the semantics-based approach. They are incrementalization, cache-and-prune, and looking for auxiliary information in the transformed incremental computation.

The incrementalization in Chapter 3 explores the most fundamental aspect of incremental computation. Based on program states, equality reasoning helps recognize repeated identical subcomputations, and an auxiliary specializer helps explore more opportunities for such recognition. Potential incrementality is thus discovered; then, it is realized by replacements using retrievals from old results. This semantics-based systematic approach sets this work apart from previous work.

The cache-and-prune approach in Chapter 4 opens up the potential of using all intermediate results as candidate information for incremental computation. Making use of the incrementalization, it is only left to prune out intermediate results that are not useful. Although the idea is natural and simple, it leads to a general systematic approach for program improvement via caching so that all previous techniques simply fall out of it.

As an approach to overcoming the non-existence of auxiliary information in the original programs, Chapter 5 makes use of the transformation for incrementalization to expose subcomputations that depend on the old input but were not performed by the old computation. After collecting such candidate auxiliary information, the method of cache-and-prune is utilized to determine useful auxiliary information as well as useful intermediate results.

Since every non-trivial computation proceeds by iteration, efficient computation requires computing each iteration incrementally based on the result of the previous iteration. Thus, the three aspects above not only form a comprehensive approach to incremental computation, but also explore the essence of improving the efficiency of computations. Examples in Chapters 4 and 5 have shown that this leads to significant program improvement in a systematic way that was not known before.

The prototype implementation CACHET, described in Chapter 6, allows semi-automatic derivation of incremental programs for a simple functional language. It not only shows that effective tools can be built based on the systematic approach, but has also provided great assistance during the advancement of the approach itself. The implementation of program analyses and transformations using the attribute grammar framework has proved to be an area of study in itself as well.

7.2 A general model for incremental computation techniques

This section summarizes the contribution of this work to a general model $\mathcal{M}$ of incremental computation techniques, namely, any such technique can be regarded as one that takes a program f and an input change operation $\oplus$, both written in some language $\mathcal{L}$, and derives two programs $\tilde{f}$ and $\tilde{f}'$, where $\tilde{f}$ extends f to return some extra information (intermediate results and auxiliary information), and $\tilde{f}'$ incrementally computes f using the information and maintains the corresponding information. Such a model $\mathcal{M}$ incorporates all three classes of work in incremental computation, introduced in Chapter 2, for the following reasons.

The development of particular incremental algorithms in the first class is a special case of the model $\mathcal{M}$, where f and $\oplus$ are fixed according to particular problems, and $\tilde{f}$ and $\tilde{f}'$ are derived manually. An incremental execution framework in the second class is a special case of the model $\mathcal{M}$ that is general in that it automatically incrementalizes an application program f, but has poor specializability in that any change by operation $\oplus$ to program f is handled in the way prescribed by the framework (and often no explicit f' is derived). Work in the third class is not only general, but also specialized to any program f and operation $\oplus$; however, so far, systematic methods focus on special cases of the model $\mathcal{M}$ where the language $\mathcal{L}$ is limited to a very-high-level language. What has been missing is an effective approach for deriving incremental programs from non-incremental ones written in a standard language.

This thesis presents such a systematic approach for deriving incremental programs from non-incremental programs written in a functional language. It begins the study of a general model for incremental computation along unique lines distinct from all other approaches. Although this problem is, in general, very hard, we have shown that an effective approach can be developed to derive incremental programs by effectively combining particular program analysis and transformation techniques.

Although the approach is presented in terms of a first-order functional language with strict semantics, the underlying principles are general and applies to other standard languages as well, e.g., higher-order functional languages, functional languages with lazy semantics, and imperative languages with complex data structures and side effects. An example has been given in Section 4.6 where the cache-and-prune principle is used to improve imperative programs with arrays for the local neighborhood problems in image processing. Further application of the principles to languages with these features is a subject for future study. Of course, special program analysis and transformation techniques related to these language features must be exploited, and they may complicate the derivation issues in one way or another, just as when partial evaluation techniques are developed to cope with such language features. On the other hand, these other language features allow some algorithms to be coded more naturally and incremental versions derived to be more efficient, making incremental computation techniques more complete.

By studying these general techniques, we aim to better understand the essence of efficient incremental computation. We also aim to establish a general framework in which different ideas on incremental computation can be integrated. By specializing the general techniques to different applications, we will be able to obtain particular incremental algorithms, particular incremental computation techniques, and particular incremental computation languages. Their applications could include most problems discussed in the literature [RR93].

7.3 Future work

Several of the ideas presented in this thesis, including the transformations for incrementalization and caching all intermediate results, have been implemented in CACHET, a prototype system for deriving incremental programs from non-incremental programs. The incremental programs developed using CACHET demonstrate the potential of the approach for systematically producing efficient programs. However, the present system is only a prototype suitable for such experimental studies and is far from being a finished project. Continuing development of CACHET is needed to increase its power and to construct a production system for a full programming language.

In a larger context, our research is directed towards being able to produce efficient incremental programs for different programming languages, not just any particular language, let along the simple language treated in this thesis. The proposed approach is semantics-based: it explores the fundamental aspects of incremental computation that are independent of any particular language. However, without further research on a generic mechanism to embed the approach, even if the principles underlying the approach are broadly applicable, different analysis and transformation techniques would have to be studied for different languages. Thus, one topic for further research is to design a generic engine that implements such a semantics-based approach, so that we can build front ends, and possibly back ends, for different languages and obtain efficient incremental programs by using the generic incrementalization engine.

We believe that an important direction that should be taken by future research on incremental computation is toward methodologies that make use of high-level properties of special application domains and support the design and implementation of efficient algorithms and systems for these domains. For example, one can imagine a specialized methodology for graphical interfaces [Rei84b,Van88,Lar92,DRO92,GT92], or one for document processing [van86,ALBB92,Mee94]. One of the richest and most challenging domain is incremental compilation, where many program analysis and transformation techniques are employed [RTD83,Rei84a,SDB84,Zad84,RP88,Bur90, AN91,PS92,JP94]. The methods developed in this dissertation make a contribution to this line of research by providing a principled approach for exploring incrementality.

Bibliography

[ACK81] F. E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, chapter 3, pages 79–101. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[AHR+90] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, January 1990.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[ALBB92] Toshiro Wakayama Allen L. Brown, Jr. and Howard A. Blair. A reconstruction of context-dependent document processing in SGML. In *Proceedins of EP '92: the International Conference on Electronic Publishing, Document Manipulation and Typography*, 1992.

[All69] Frances E. Allen. Program optimization. In *Annual Review of Automatic Programming*, volume 5, pages 239–307. Pergamon: Elmsford, New York, 1969.

[AN91] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *Proceedings of the 5th ACM Conference on FPCA*, volume 523 of *Lecture Notes in Computer Science*, pages 379–405, Cambridge, Massachusetts, August 1991. Springer-Verlag, Berlin.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[BC88] P. Borras and D. Clément. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, Boston, Massachusetts, November 1988. Published as SIGPLAN Notices, 24(2).

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[Ber92] Arthur Michael Berman. *Lower and Upper Bounds for Incremental Algorithms*. Ph.D. dissertation, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, October 1992.

[BG94] John Boyland and Susan L. Graham. Composing tree attributions. In *Conference Record of the 21th Annual ACM Symposium on POPL*, pages 375–388, Portland, Oregon, January 1994.

[BGV92] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The *Pan* language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.

[Bir80] Richard S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[Bir84] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

[Bir85] Richard S. Bird. Addendum: The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 7(3):490–492, July 1985.

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

[BMPP89] Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper. Formal program construction by transformations—computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[BP92] Bard Bloom and Robert Paige. Computing ready simulations efficiently. In A. Zwarico and S. Purushothaman, editors, *Proceedings of the 1st North American Process Algebra Workshop*, Workshops in Computer Science Series, pages 119–134. Springer-Verlag, Berlin, 1992.

[BPR90] Arthur M. Berman, M. C. Paull, and Barbara G. Ryder. Proving relative lower bounds for incremental algorithms. *Acta Informatica*, 27:665–683, 1990.

[Bro84] Manfred Broy. Algebraic methods for program construction: The project CIP. In Peter. Pepper, editor, *Program Transformation and Programming Environments*, volume 8 of *NATO Advanced Science Institutes Series F: Computer and System Sciences*, pages 199–222. Springer-Verlag, Berlin, 1984. Proceedings of the NATO Advanced Research Workshop on Program Transformation and Programming Environments, directed by F. L. Bauer and H. Remus, Munich, Germany, September 1983.

[BS86] Rofl Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[Bur69] R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[Bur90] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Car84] Robert Cartwright. Recursive programs as definitions in first order logic. *SIAM Journal on Computing*, 13(2):374–408, May 1984.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, Copenhagen, Denmark, June 1993.

[CHKS93] S. Ceri, M. A. W. Houtsma, A. M. Keller, and P. Samarati. Achieving incremental consistency among autonomous replicated databases. *IFIP Transactions A [Computer Science and Technology]*, A-25:223–237, 1993.

[CK77] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[CK93] Wei-Ngan Chin and Siau-Cheng Khoo. Tupling functions with multiple recursion parameters. In Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors, *Proceedings of the 3rd International Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, Berlin, September 1993.

[Coh83] N. H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[CP92] Chia-Hsiang Chang and Robert Paige. From regular expressions to dfa's using compressed nfa's. In *Proceedings of the 3rd Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 88–108, Tucson, Arizona, April 1992. Springer-Verlag, Berlin.

[CP89] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, September 1988/89.

[CPT92] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, November 1992. Special issue of best papers selected from A. Arnold, editor, Proceedings of CAAP '90: the 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990.

[CS70] John Cocke and John T. Schwartz. Programming Languages and Their Compilers; Preliminary Notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[DC90] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, April 1990. Special issue on procedural programming.

[Der83] Nachum Dershowitz. *The Evolution of Programs*, volume 5 of *Progress in Computer Science*. Birkhäuser, Boston, 1983.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structure editor: The Mentor experience. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140. McGraw-Hill, New York, 1984.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1988.

[DRO92] Jr. Dan R. Olsen. *User Interface Management Systems: Models and Algorithms*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[Ear76] Jay Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1:321–342, 1976.

[EGIN92] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. In *Proceedings of the 33rd Annual IEEE Symposium on FOCS*, Pittsburgh, Pennsylvania, October 1992.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, July 1986. Published as SIGPLAN Notices, 21(7).

[Far92] Charles Farnum. Pattern-based tree attribution. In *Conference Record of the 19th Annual ACM Symposium on POPL*, pages 211–222, January 1992.

[Fea82] Martin S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

[Fea87] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In *Program Specification and Transformation*, pages 165–195. North-Holland, 1987. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz, FRG, April 1986.

[Fie91] John Henry Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, 1991. Also appeared as Technical Report TR 91-1243, November 1991.

[Fie93] John Field. A graph reduction approach to incremental term rewriting. In *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, Montreal, June 1993. Springer-Verlag, Berlin.

[Flo63] Ivan Flores. *The Logic of Computer Arithmetic.* Prentice-Hall International Series in Electrical Engineering. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[FMB90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *Conference Record of the 19th Annual ACM Symposium on POPL*, pages 223–234, January 1992.

[FN88] Y. Futamura and K. Nogi. Generalized partial evaluation. In *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, Amsterdam, 1988.

[Fon77] Amelia C. Fong. Generalized common subexpressions in very high level languages. In *Conference Record of the 4th Annual ACM Symposium on POPL*, pages 48–57, Los Angeles, California, January 1977.

[Fon79] Amelia C. Fong. Inductively computable constructs in very high level languages. In *Conference Record of the 6th Annual ACM Symposium on POPL*, pages 21–28, San Antonio, Texas, January 1979.

[Fre85] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, November 1985.

[FT90] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LFP*, pages 307–322, 1990.

[FU76] Amelia C. Fong and Jeffrey D. Ullman. Inductive variables in very high level languages. In *Conference Record of the 3rd Annual ACM Symposium on POPL*, pages 104–112, Atlanta, Georgia, January 1976.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 157–170, June 1984. Published as SIGPLAN Notices, 19(6).

[Gla95] James Glanz. Mathematical logic flushes out the bugs in chip designs. *Science*, 267:332–333, January 20, 1995.

[GM79] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, July 1979.

[Gol72] Herman H. Goldstine. Charles babbage and his analytical engine. In *The Computer from Pascal to von Neumann*, chapter 2, pages 10–26. Princeton University Press, Princeton, New Jersey, 1972.

[Gri71] David Gries. *Compiler Construction for Digital Computers*. John Wiley & Sons, New York, 1971.

[Gri81] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[Gri84] David Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1984.

[GT92] Philip Gray and Roger Took, editors. *Building Interactive Systems: Architectures and Tools*. Workshops in Computing. Springer-Verlag, London, 1992. Published in collaboration with the British Computer Society. Proceedings of two separate meetings: Architectures for Interactive Systems, University of York, March 4, 1991, and Object-Oriented Tools for User Interface Construction, University of Glasgow, April 5, 1991.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1992.

[Hal90] Robert J. Hall. Program improvement by automatic redistribution of intermediate results. Technical Report AI-TR-1251, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, Decemeber 1990.

[Hal91] Robert J. Hall. Program improvement by automatic redistribution of intermediate results: An overview. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, chapter 14, pages 339–372. AAAI Press/The MIT Press, 1991. Proceedings of the AAAI '88 Workshop on Automating Software Design.

[Hec88] Reinhold Heckmann. A functional language for the specification of complex tree transformations. In *Proceedings of the 2nd ESOP*, volume 300 of *Lecture Notes in Computer Science*, pages 175–190, Nancy, France, March 1988. Springer-Verlag, Berlin.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55, 1978.

[HN86] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[Hoo92] Roger Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN '92 Conference on PLDI*, pages 261–272, California, June 1992.

[HT86] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[Hug85] John Hughes. Lazy memo-functions. In *Proceedings of the 2nd Conference on FPCA*, volume 201 of *Lecture Notes in Computer Science*, pages 129–146, Nancy, France, September 1985. Springer-Verlag, Berlin.

[JG82] Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *Conference Record of the 9th Annual ACM Symposium on POPL*, pages 196–206, Albuquerque, New Mexico, January 1982.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[JL89] Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the 4th International Conference on FPCA*, pages 54–74, London, U.K., September 1989.

[Jon90] Larry G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, July 1990.

[JP94] Richard Johnson and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on PLDI*, pages 171–185, Orlando, Florida, June 1994.

[JSS85] Neil D. Jones, Peter Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140, Dijon, France, May 1985. Springer-Verlag, Berlin.

[Kai89] Gail E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):168–193, April 1989.

[Kas80] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.

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

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1952. Tenth reprint, Wolters-Noordhoff Publishing, Groningen and North-Holland Publishing Company, Amsterdam, 1991.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on PLDI*, pages 147–158, Orlando, Florida, June 1994.

[KS86] Robert M. Keller and M. Ronan Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, January 1986.

[Lar92] James A. Larson. *Interactive Software: Tools For Building Interactive User Interfaces*. Yourdon Press Computing Series. Yourdon Press, Englewood Cliffs, New Jersey, 1992.

[Lau88] John Launchbury. Projections for specialisation. In *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, 1988.

[Lau89] John Launchbury. *Projection Factorisations in Partial Evaluation*. Ph.D. dissertation, Department of Computing, University of Glasgow, 1989.

[Lau91] John Launchbury. Strictness and binding-time analysis: Two for the price of one. In *Proceedings of the ACM SIGPLAN '91 Conference on PLDI*, pages 80–91, Toronto, Ontario, Canada, June 1991.

[LD93] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 124–136, January 1993.

[Liu95] Yanhong A. Liu. CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 19–26, Boston, Massachusetts, November 1995. IEEE Computer Society Press.

[LMOW88] Peter Lipps, Ulrich Möncke, Matthias Olk, and Reinhard Wilhelm. Attribute (re)evaluation in OPTRAN. *Acta Informatica*, 26:213–239, 1988.

[LMW88] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. OPTRAN: A language/system for the specification of program transformation—system overview and experiences. In Dieter Hammer, editor, *Proceedings of the 2nd CCHSC Workshop: Workshop on Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65, Berlin, GDR, October 1988. Springer-Verlag, Berlin, 1989.

[LS92] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, December 1992.

[LST96] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on POPL*, St. Petersburg Beach, Florida, January 1996.

[LT95a] Yanhong A. Liu and Tim Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, pages 190–201, La Jolla, California, June 1995.

[LT95b] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, February 1995.

[MC85] D. J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the 9th IJCAI*, pages 165–172, Los Angeles, August 1985.

[Mee94] Lambert Meertens. Incremental optimum-fit line breaking. Technical Report WG 2.1 TR 720 REN-5, CWI, January 1994.

[MF81] R. Medina-Mora and P. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, SE-7(5):472–482, September 1981.

[Mic68] Donald Michie. "memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

[Mil91] Peter Bro Miltersen. On-line reevaluation of functions. Technical Report ALCOM-91-63, Aarhus University, Aarhus, Demark, May 1991.

[MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Mog88] Torben Mogensen. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the 17th Annual ACM Symposium on POPL*, pages 184–196, San Francisco, California, January 1990.

[MSVT94] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models fo incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[MW80] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[MW92] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, August 1992.

[MW93] Zohar Manna and Richard Waldinger. *The Deductive Foundations of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1993.

[Ner92] Anil Nerode. Private communication and email communication on deriving incremental programs, May–July 1992.

[Ner95] Anil Nerode. Private communication on caching intermediate results, August 1995.

[OLHA94] John O'Leary, Miriam Leeser, Jason Hickey, and Mark Aagaard. Nonrestoring integer square root: A case study in design by principled optimization. In Ramayya Kumar and Thomas Kropf, editors, *Proceedings of TPCD '94: the 2nd International Conference on Theorem Provers in Circuit Design—Theory, Practice and Experience*, volume 901 of *Lecture Notes in Computer Science*, pages 52–71, Bad Herrenalb (Black Forest), Germany, September 1994. Springer-Verlag, Berlin, 1995.

[Pai81] Robert Paige. *Formal Differentiation: A Program Synthesis Technique*, volume 6 of *Computer Science and Artificial Intelligence*. UMI Research Press, Ann Arbor, Michigan, 1981. Revision of Ph.D. dissertation, New York University, 1979.

[Pai83] Robert Paige. Transformational programming—applications to algorithms and systems. In *Conference Record of the 10th Annual ACM Symposium on POPL*, pages 73–87, January 1983.

[Pai84] Robert Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory, Vol. 2*, pages 171–210. Plenum Press, New York, March 1984.

[Pai86] Robert Paige. Programming with invariants. *IEEE Software*, pages 56–69, January 1986.

[Pai89] Robert Paige. Real-time simulation of a set machine on a RAM. In *Computing and Information, Vol. II*, pages 69–73. Canadian Scholars Press, 1989. Proceedings of ICCI '89: the International Conference on Computing and Information, Toronto, Canada, May 23-27, 1989.

[Pai90] Robert Paige. Symbolic finite differencing—part I. In *Proceedings of the 3rd ESOP*, volume 432 of *Lecture Notes in Computer Science*, pages 36–56, Copenhagen, Denmark, May 1990. Springer-Verlag, Berlin.

[Pai94] Robert Paige. Viewing a program transformation system at work. In M. Hermenegildo and J. Penjam, editors, *Proceedings of Joint 6th International Conference on PLILP and 4th International Conference on ALP*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer-Verlag, Berlin, September 1994.

[Par90] Helmut A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1990.

[Pau83] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Pet84] Alberto Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Proceedings of the ACM '84 Symposium on LFP*, Austin, Texas, August 1984.

[Pet87] Alberto Pettorossi. Strategical derivation of on-line programs. In L. G. L. T. Meertens, editor, *Program Specification and Transformation*, pages 73–88. North-Holland, Amsterdam, 1987. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz, FRG, April 1986.

[PH87] Robert Paige and Fritz Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code—a case study. *Journal of Symbolic Computation*, 4(2):207–232, 1987.

[PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PS77] Bob Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *Conference Record of the 4th Annual ACM Symposium on POPL*, pages 58–71, January 1977.

[PS83] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[PS92] Lori L. Pollock and Mary Lou Soffa. Incremental global reoptimization of programs. *ACM Transactions on Programming Languages and Systems*, 14(2):173–200, April 1992.

[PT87] Robert Paige and Robert Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, December 1987.

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on POPL*, pages 315–328, January 1989.

[PTB85] Robert Paige, Robert Tarjan, and Robert Bonic. A linear time solution to the single coarsest partition problem. *Theoretical Computer Science*, 40:67–84, 1985.

[Pug88a] William Pugh. An improved cache replacement strategy for function caching. In *Proceedings of the ACM '88 Conference on LFP*, pages 269–276, 1988.

[Pug88b] William Worthington Pugh, Jr. *Incremental Computation and the Incremental Evaluation of Functional Programs*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, August 1988.

[Ram93] G. Ramalingam. *Bounded Incremental Computation*. Ph.D. dissertation, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1993. Also appeared as Technical Report #1172, August 1993.

[Red88] Uday Reddy. Transformational derivation of programs using the Focus system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 163–172, 1988. Published as SIGSOFT Software Engineering Notes, 13(5), November 1988 and SIGPLAN Notices, 23(2), February 1989.

[Rei84a] Steven P. Reiss. An approach to incremental compilation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 144–156, Montreal, Canada, June 1984. Published as SIGPLAN Notices, 19(6).

[Rei84b] Steven P. Reiss. Graphical program development with PECAN program development systems. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, Pittsburgh, Philadelphia, April 1984.

[Rei90a] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[Rei90b] Steven P. Reiss. Interacting with the FIELD environment. *Software—Practice and Experience*, 20(S1):89–115, June 1990.

[Rep84] Thomas William Reps. *Generating Language-Based Environments*. ACM Doctoral Dissertation Award. The MIT Press, Cambridge, Massachusetts, 1984. Ph.D. dissertation, Cornell University, 1982.

[Rey81] John C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.

[Ros89] Mads Rosendahl. Automatic complexity analysis. In *Proceedings of the 4th International Conference on FPCA*, pages 144–156, London, U.K., September 1989.

[RP88] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[RR91] G. Ramalingam and Thomas Reps. On the computational complexity of incremental algorithms. Technical Report TR-1033, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, August 1991.

[RR93] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 502–510, Charleston, South Carolina, January 1993.

[RR94] G. Ramalingam and Thomas Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Conference Record of the 21th Annual ACM Symposium on POPL*, Portland, Oregon, January 1994.

[RT88] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1988.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[Sco82] Dana S. Scott. Lectures on a mathematical theory of computation. In Manfred Broy and Gunther Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292. D. Reidel Publishing Company, 1982. Lecture notes of 1981 Marktoberdorf Summer School on Theoretical Foundations of Programming Methodology, directed by F.L. Bauer, E.W. Dijkstra, and C.A.R. Hoare.

[SDB84] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental compilation in magpie. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 122–131, June 1984. Published as SIGPLAN Notices, 19(6).

[SH91] R. S. Sundaresh and Paul Hudak. Incremental computation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on POPL*, pages 1–13, January 1991.

[SKR91] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. Efficient code motion and an adaption to strength reduction. In *Proceedings of the*

4th International Joint Conference on TAPSOFT, volume 494 of *Lecture Notes in Computer Science*, pages 394–415, Brighton, U.K., 1991. Springer-Verlag, Berlin.

[SL90] Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305–321, 1990.

[Smi90] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[Smi91] Douglas R. Smith. KIDS—a knowledge-based software development system. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, chapter 19, pages 483–514. AAAI Press/The MIT Press, 1991. Proceedings of the AAAI '88 Workshop on Automating Software Design.

[SP93] Douglas R. Smith and Eduardo A. Parra. Transformational approach to transportation scheduling. In *Proceedings of the 8th Knowledge-Base Software Engineering Conference*, Chicago, Illinois, September 1993. IEEE Computer Society Press.

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[Sun91] R. S. Sundaresh. Building incremental programs using partial evaluation. In *Proceedings of the Symposium on PEPM*, pages 83–93, Yale University, June 1991. Published as SIGPLAN Notices, 26(9).

[SVT93] S. Sairam, Jeffrey Scott Vitter, and Roberto Tamassia. A complexity theoretic approach to incremental computation. In *Proceedings of STACS '93: the 10th Annual Symposium on TACS*, volume 665 of *Lecture Notes in Computer Science*, pages 640–649, Würzburg, Germany, February 1993. Springer-Verlag, Berlin.

[Tra86] Kenneth R. Traub. A compiler for the MIT tagged-token dataflow architecture. Masters dissertation, Department of Electrical Engineering and Computer Science, MIT, 1986. Appeared as Technical Report LCS TR-370, August, 1986.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[van86] J. C. van Vliet, editor. *Text Processing and Document Manipulation: Proceedings of the International Conference*, University of Nottingham, April 1986. British Computer Society, Cambridge University Press.

[Van88] Bradley T. Vander Zanden. *Incremental Constraint Satisfaction and its Application to Graphical Interfaces*. Ph.D. dissertation, Department of

Computer Science, Cornell University, Ithaca, New York, 1988. Also appeared as Technical Report TR 88-941, October 1988.

[van92] Emma A. van der Meulen. Deriving incremental implementations from algebraic sepcifications. In *Proceedings of the 2nd International Conference on Algebraic Methodology and Software Technology*, pages 277–286. Springer-Verlag, Berlin, 1992.

[VC92] A. Varma and S. Chalasani. An incremental algorithm for TDM switching assignments in satellite and terrestrial networks. *IEEE Journal on Selected Areas in Communications*, 10(2):364–377, February 1992.

[VSK89] Harald H. Vogt, Doaitse Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on PLDI*, pages 131–145, June 1989.

[VVF90] Harald Vogt, Aswin Van den Berg, and Arend Freije. Rapid development of a program transformation system with attribute grammars and dynamic trnsformations. In *Proceedings of the International Workshop on Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 101–115, Paris, France, September 1990. Springer-Verlag, Berlin.

[Web92] J. Webb. Steps towards architecture-independent image processing. *IEEE Computer*, February 1992.

[Web93] Adam Brooks Webber. *Principled Optimization of Functional Programs*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, 1993. Also appeared as Technical Report TR 93-1363, June 1993.

[Web95] Adam Webber. Optimization of functional programs by grammar thinning. *ACM Transactions on Programming Languages and Systems*, 17(2):293–330, March 1995.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–538, September 1975.

[Weg76] Ben Wegbreit. Goal-directed program transformation. *IEEE Transactions on Software Engineering*, SE-2(2):69–80, June 1976.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wel86] William M. Wells, III. Efficient synthesis of Gaussian filters by cascaded uniform filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):234–239, March 1986.

[WH87] Philip Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407, Portland, Oregon, September 1987.

[Yeh83] D. Yeh. On incremental evaluation of ordered attribute grammars. *BIT*, 23:308–320, 1983.

[Yel93] Daniel M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, July 1993.

[YK88] D. Yeh and U. Kastens. Improvements on an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Notices*, 23(12):45–50, 1988.

[YS91] Daniel M. Yellin and Robert E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991.

[Zab94] Ramin Zabih. *Individuating Unknown Objects by Combining Motion and Stereo*. Ph.D. dissertation, Department of Computer Science, Stanford University, Stanford, California, 1994.

[Zad84] Frank Kenneth Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 132–143, June 1984. Published as SIGPLAN Notices, 19(6).

[ZGMHW94] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, June 1994.

[ZW94] Ramin Zabih and John Woodfill. Non-parametric local transforms for computing visual correspondence. In Jan-Olof Eklundh, editor, *3rd European Conference on Computer Vision*, volume 801 of *Lecture Notes in Computer Science*, pages 151–158. Springer-Verlag, 1994.

Index