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

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.