Incremental Computation for Transformational Software Development

Yanhong A. Liu* Tim Teitelbaum*

March 1995

Abstract

Given a program f and an input change $\oplus$, we wish 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. Obtaining such incremental programs is an essential part of the transformational-programming approach to software development and enhancement. This paper presents a systematic approach that discovers a general class of useful auxiliary information, combines it with useful intermediate results, and obtains an efficient incremental program that uses and maintains these intermediate results and auxiliary information. We give a number of examples from list processing, VLSI circuit design, image processing, *etc.*

1 Introduction

Software engineering is the systematic approach to the development, operation, maintenance, and retirement of software [1]. The transformational-programming approach to software engineering advocates the use of formal source-to-source transformations to reduce programming labor, improve program reliability, and upgrade program performance [35, 37]. During development, semantics-preserving transformations refine correct high-level specifications and inefficient programs into executable code and more efficient programs. During maintenance, performance bottlenecks are eliminated using similar techniques.

One of the most important techniques used to improve program performance involves *incrementally* updating results of computations as their parameters change rather than computing the results from scratch. This problem has received much attention in the transformational programming literature [7, 12, 35, 37, 45], where it is commonly known as *finite differencing*. However, general techniques of *incremental computation* have far broader application throughout software, e.g., loop optimizations in optimizing compilers [2, 3, 11, 14, 46], interactive systems like editors [6, 13, 43] and programming environments [5, 9, 13, 19, 24, 25, 41, 42], dynamic systems like distributed databases [10, 29] and real-time systems [48], and image processing [49, 51, 53, 54].

Incremental Computation. Given a program f and an input change operation $\oplus$, a program f' that computes the result of $f(x \oplus y)$ efficiently by making use of the value of $f(x)$ is called an *incremental version* of f under $\oplus$. Often, some *intermediate results* used in computing the output $f(x)$ also need to be maintained for efficient incremental computation of $f(x \oplus y)$. Moreover, certain *auxiliary information* about x may need to be discovered and maintained as well.

Numerous techniques for incremental computation have been developed, e.g., [3, 4, 16, 20, 21, 24, 31, 36, 38, 40, 43, 44, 47, 52]. In [31], we give a systematic transformational approach for deriving an incremental program f' from a given program f and an input change $\oplus$. The basic idea is to identify in the computation of $f(x \oplus y)$ those subcomputations that are also performed in the computation of $f(x)$ and whose values can be retrieved from the cached result r of $f(x)$. The computation of $f(x \oplus y)$ is symbolically transformed to avoid re-performing these subcomputations by replacing them with corresponding retrievals. This efficient way of computing $f(x \oplus y)$ is captured in the definition of $f'(x, y, r)$.

*The authors gratefully acknowledges the support of the Office of Naval Research under contract No. N00014-92-J-1973.
Author's address: Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: yanhong,tt@cs.cornell.edu

Most methods in incremental computation exploit caching intermediate results of certain kinds, e.g., annotating parse trees with attributes [43], dynamically caching results of function calls [39, 40], saving results of intermediate reductions at appointed places [16], maintaining the change-detailing network of INC [52], keeping residual programs for certain input partitions [47], and maintaining intermediate results and dependencies identified by Alphonse annotations [20]. In [30], we have given a cache-and-prune method to statically transform programs to cache all kinds of intermediate results needed for incremental computation. The basic idea is to (I) extend the program f to a program $\bar{f}$ that returns all intermediate results, (II) incrementalize the program $\bar{f}$ under $\oplus$ to obtain an incremental version $\bar{f}'$, and (III) using the dependencies in $\bar{f}'$, prune the extended program $\bar{f}$ to a program $\hat{f}$ that returns only the useful intermediate results, and prune the program $\bar{f}'$ to obtain a program $\hat{f}'$ that incrementally maintains the useful intermediate results.

Some methods in incremental computation exploit certain kinds of auxiliary information, e.g., auxiliary arithmetic associated with some classical strength-reduction rules [3], dynamic mappings maintained by finite differencing rules for aggregate primitives in SETL [36] and INC [52], and auxiliary data structures for problems with certain properties like stable decomposition [40] and other decomposition properties [15]. However, until now, the systematic discovery of auxiliary information for general problems has been a subject left completely open for study. Methods for dealing with incremental programs, intermediate results, and auxiliary information together have also been needed.

This Paper. This paper presents a systematic approach that discovers a general class of auxiliary information for any incremental computation problem. More importantly, it combines discovering auxiliary information with obtaining incremental programs and caching intermediate results.

Basically, we obtain candidate auxiliary information by transforming $f(x \oplus y)$ (using a revision of [31]) and collecting all subcomputations that depend on the previous input x. Then, we merge such information with intermediate results of $f(x)$ and decide which of them are needed for efficient incremental computation and how they can be used (in a fashion similar to the cache-and-prune method in [30]). As a result, we derive a program $\tilde{f}'$ that uses the needed intermediate results and auxiliary information to compute f incrementally, while incrementally maintaining these results and information at the same time.

Our approach is modular: each component performs relatively independent analyses and transformations. This facilitates integrating other techniques into our framework and re-using our components for other optimizations.

Several examples demonstrate the power and generality of our approach for transformational software development: list processing for pure incremental computation problems, strength reduction in optimizing compilers, principled VLSI design, automating previous error-prone "eureka"-like program derivations, and efficient algorithms for image processing.

2 Approach

Auxiliary information is, by definition, useful information not computed by the original program f, so it can not be directly obtained from f. On the other hand, auxiliary information is data that would speed up the computation of $f(x \oplus y)$ given information about $f(x)$. Seeking to obtain such information systematically, we come to the idea that when computing $f(x \oplus y)$, for example in the manner of $f'(x, y, r)$, there are often subcomputations that depend only on x and r, but not on y. If the values of these subcomputations were available, then we could make f' faster.

To obtain such candidate auxiliary information, we transform the computation $f(x \oplus y)$ and collect subcomputations that do not depend on y. To decide whether this collection of candidate auxiliary information is useful for the incremental computation of f and how it can be used, we merge it with the original computation of f, derive an incremental version for the resulting program, and determine the least amount of this information needed to compute the return value of $f(x \oplus y)$ in the incremental version.

Considerations. Several refinements of the general strategy need to be addressed before the complete transformation procedure can be presented.

• Computing *intermediate results* of $f(x)$ incrementally, with *their* corresponding auxiliary information, is often crucial for the efficient incremental computation of $f(x \oplus y)$. Thus, instead of collecting candidate auxiliary information from the incremental computation of f on $x \oplus y$, we collect it in the incremental computation of $\bar{f}$ on $x \oplus y$, where $\bar{f}$ is f extended to cache all intermediate results.

• In transforming the computation of $\bar{f}(x \oplus y)$, we modify the incrementalization method of [31] to expose *more* subcomputations that do not depend on y, especially subcomputations whose values can not be retrieved from the cached result of $\bar{f}(x)$.

• Not all computations performed in the transformed computation of $\bar{f}(x \oplus y)$ are candidate auxiliary information. In particular, those that depend on y are not useful information about x. A forward dependence analysis and a collection transformation are needed to identify and collect subcomputations that depend only on certain arguments.

• Although caching intermediate results wouldn't increase the asymptotic computation time [30], we must consider the cost of maintaining the collected auxiliary information. Let $\tilde{f}'$ be the resulting incremental program that computes the value of f and maintains the needed intermediate results and auxiliary information. We use $\tilde{f}'$ only if the incremental computation is at least as fast as computing from scratch using only f. Sometimes, this can be guaranteed by simply checking that computing auxiliary information from scratch is at least as fast as computing f from scratch.

Three-Phase Transformation. The overall approach consists of three phases, corresponding to the three-stage method for caching intermediate results in [30]. However, Phase I is extended here to discover and compute candidate auxiliary information as well.

- Phase I constructs a program $\bar{\bar{f}}$ that caches all intermediate results and candidate auxiliary information for computing f incrementally under $\oplus$. It has four steps:
  - Step 1 constructs program $\bar{f}$, an extended version of f, such that $\bar{f}(x)$ returns the values of all intermediate results used in computing $f(x)$, as in [30].
  - Step 2 derives program $\bar{f}''$, an incrementalized version of $\bar{f}$ on $x \oplus y$, using a modified method of [31] to expose subcomputations that do not depend on y.
  - Step 3 builds program $\check{f}$, a collected version of $\bar{f}''$, such that $\check{f}(x, \bar{r})$ returns as candidate auxiliary information the values of intermediate results used in $\bar{f}''(x, y, \bar{r})$ depending only on x and $\bar{r}$.
  - Step 4 defines program $\bar{\bar{f}}$ and a projection Π_0, where $\bar{\bar{f}}$ combines $\bar{f}$ and $\check{f}$ to return the value of f, the intermediate results, and the candidate auxiliary information, and Π_0 projects the value of f out of the value of $\bar{\bar{f}}$.
- Phase II derives program $\bar{\bar{f}}'$, an incremental version of $\bar{\bar{f}}$ under $\oplus$, using the approach in [31].
- Phase III generates program $\tilde{f}$, a pruned version of $\bar{\bar{f}}$, such that $\tilde{f}(x)$ returns $\Pi(\bar{\bar{r}})$, where $\bar{\bar{r}}$ is the return value of $\bar{\bar{f}}(x)$, and $\Pi(\bar{\bar{r}})$ projects out $\Pi_0(\bar{\bar{r}})$ and other components of $\bar{\bar{r}}$ on which $\Pi_0(\bar{\bar{f}}'(x, y, \bar{\bar{r}}))$ depends. Phase III also generates program $\tilde{f}'$, a pruned version of $\bar{\bar{f}}'$, such that if $\bar{\bar{f}}'(x, y, \bar{\bar{r}})$ returns $\bar{\bar{r}}'$, then $\tilde{f}'(x, y, \tilde{r})$, where $\tilde{r}$ is $\Pi(\bar{\bar{r}})$ as above, returns $\Pi(\bar{\bar{r}}')$. This phase uses methods in [30].

3 Running Example

We detail our method with the aid of a running example, *cmp*, given in Figure 1 as a set of first-order recursive equations with call-by-value semantics. In general, an input change $\oplus$ to a program f 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 this example, we consider an input change to *cmp* to be $x' = x \oplus y = cons(y, x)$. For typographical convenience, we shall always use x to refer to the previous input to f, r the cached result of $f(x)$, and y the change parameter to the input x.[1]

[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 [31].

$$
\begin{array}{lll}
cmp(x) & = & sum(odd(x)) \le prod(even(x)) \qquad \text{— compare sum of odd and product of even positions}
\end{array}
$$

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

Figure 1: Running example

In this paper, we use a simple space model that assumes unlimited space is available to achieve the least asymptotic time possible. Since only asymptotic time is of concern here and we assume that all primitive functions take constant time, for this list processing example, it is sufficient to consider only the values of function applications as candidate intermediate results to be cached or auxiliary information to be added.

We use $<>$ to denote a list constructed by the transformation to bundle intermediate results and auxiliary information with the original return value, and we use nth to get the nth element of such a list. In particular, the value of an original computation is always the first element, which is retrieved by $1st$. The other elements are the corresponding intermediate results and/or auxiliary information.

At the end of our running example, we will have obtained the programs $\widetilde{cmp}$ and $\widetilde{cmp}'$ shown in Figure 2. Rather than computing cmp from scratch using $O(n)$ time, $\widetilde{cmp}'$ computes incrementally using $O(1)$ time.

$cmp(x) = 1st(\widetilde{cmp}(x))$.
For x of length n, $\widetilde{cmp}(x)$ takes time $O(n)$;
$cmp(x)$ takes time $O(n)$.

If $\widetilde{cmp}(x) = \tilde{r}$, then $\widetilde{cmp}'(y, \tilde{r}) = \widetilde{cmp}(cons(y, x))$.
For x of length n, $\widetilde{cmp}'(y, \tilde{r})$ takes time $O(1)$;
$\widetilde{cmp}(cons(y, x))$ takes time $O(n)$.

$$
\begin{array}{lll}
\widetilde{cmp}(x) & = & \textbf{let } v_1 = odd(x) \textbf{ in} \\
 & & \textbf{let } u_1 = sum(v_1) \textbf{ in} \\
 & & \textbf{let } v_2 = even(x) \textbf{ in} \\
 & & \textbf{let } u_2 = prod(v_2) \textbf{ in} \\
 & & < u_1 \le u_2,\ u_1,\ u_2,\ sum(v_2),\ prod(v_1) > \\
\widetilde{cmp}'(y, \tilde{r}) & = & < y+4th(\tilde{r}) \le 5th(\tilde{r}), \\
 & & \quad y+4th(\tilde{r}),\ 5th(\tilde{r}),\ 2nd(\tilde{r}),\ y*3rd(\tilde{r}) >
\end{array}
$$

Figure 2: Resulting programs

4 Analysis and Transformation Methods

This section presents the analysis and transformation techniques used for incrementalization with intermediate results and auxiliary information, as outlined in Section 2.

4.1 Phase I. Caching Intermediate Results/Discovering Auxiliary Information

This phase constructs a program that extends the original program to return all intermediate results and candidate auxiliary information, as well as the original value.

4.1.1 Step 1. Caching All Intermediate Results

Step 1 extends the program f to cache all intermediate results using the local, structure-preserving *extension* transformation $\mathcal{E}xt$ in [30]. Basically, $\mathcal{E}xt$ extends a computation to return the values of all nontrivial intermediate computations, i.e., it introduces names for the values of intermediate computations, and builds up data structures for these values together with the values of the original computation.

Example. For the program cmp, after caching all intermediate results, we obtain the program $\overline{cmp}$ below, where $cmp(x) = 1st(\overline{cmp})$, and $\_$ is a dummy constant that just occupies a slot.

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

4.1.2 Step 2. Exposing Auxiliary Information by Incrementalization

Step 2 transforms $\bar{f}(x \oplus y)$ to expose subcomputations depending on x but not y using transformations similar to the *incrementalization* in [31] for obtaining the incremental version $\bar{f}'(x, y, \bar{r})$. However, our present motivation differs in that we are not interested here in whether the values of subcomputations can be efficiently retrieved from the cached value of $\bar{f}$. We are only interested in finding potentially useful candidates for auxiliary information. Thus, efficiency considerations are dropped here and are picked up later. Also, a recursive function application may be replaced by a call to a derived incremental function regardless of whether a cache retrieval can be used as argument for the cache parameter r. This leads to the exposure of more subcomputations depending only on x.

Example. For the program $\overline{cmp}$ in (1), no recursive function application needs to be treated specially. After incrementalizing $\overline{cmp}(cons(y, x))$, with $\overline{cmp}(x) = \bar{r}$:

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

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

2. transform four applications

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

3. replace applications of $\overline{even}$ and $\overline{odd}$

$$
\begin{array}{ll}
= & \textbf{let } v_1' = 4th(\bar{r}) \textbf{ in} \\
& \textbf{let } u_1' = \overline{sum}(1st(v_1')) \textbf{ in} \\
& \textbf{let } v_2' = 2nd(\bar{r}) \textbf{ in} \\
& \textbf{let } u_2 = \overline{prod}(1st(v_2')) \textbf{ in} \\
& < y + 1st(u_1') \le 1st(u_2), \\
& \quad < cons(y, 1st(v_1')), v_1' >, \\
& \quad < y + 1st(u_1'), u_1' >, \\
& \quad < 1st(v_2'), v_2' >,\ u_2 >
\end{array}
$$

we obtain the program $\overline{cmp}''$ such that, if $\overline{cmp}(x) = \bar{r}$, then $\overline{cmp}''(y, \bar{r}) = \overline{cmp}(cons(y, x))$, as follows:

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

where $\overline{sum}$ and $\overline{prod}$ are defined as in (1).

4.1.3 Step 3. Collecting Candidate Auxiliary Information

Step 3 collects intermediate results of $\bar{f}''(x, y, \bar{r})$ that depend only on x and $\bar{r}$ using a *collection* transformation $\mathcal{C}ol$. $\mathcal{C}ol$ is similar to the transformation $\mathcal{E}xt$ in that both collect intermediate results. The difference is that $\mathcal{E}xt$ collects *all* intermediate results, whereas $\mathcal{C}ol$ collects *only those* that depend only on x and $\bar{r}$.

First, we use a forward dependence analysis to identify subcomputations of $\bar{f}''(x, y, \bar{r})$ that depend only on x and $\bar{r}$. Basically, we define a set of recursive data flow equations to specify the dependence relations for the program $\bar{f}''(x, y, \bar{r})$, and, starting with the arguments x and $\bar{r}$, iterate to compute the least fixed point of these equations.

Then, we use transformation $\mathcal{C}ol$ to collect the results of these computations. While, for the transformation $\mathcal{E}xt$, what a subexpression originally computes *always* needs to be preserved, for the transformation $\mathcal{C}ol$, what a subexpression originally computes is preserved *only when* it is used to compute the results of intermediate computations that do not depend on y. Therefore, we can use an extra argument to denote whether the original value of a subcomputation is possibly used. When this is true, $\mathcal{C}ol$ acts like $\mathcal{E}xt$; otherwise, $\mathcal{C}ol$ directly collects the intermediate results without having to build the value of the original computation into the final return value.

Our dependence analysis is similar to binding-time analysis for partial evaluation [23, 28] if we regard arguments x and $\bar r$ of $\bar f''(x, y, \bar r)$ as static and y as dynamic. However, binding time analysis helps obtain a residual program that takes only the dynamic one as arguments, while forward dependence analysis helps obtain a program that computes only on the static arguments. Our resulting program is similar to the resulting slice obtained from forward slicing [50]. However, forward slicing finds parts of a program that depend *possibly* on certain information, while our analysis finds parts of a program that depend *only* on certain information.

Example. For the program $\overline{cmp}''$ in (2), we start with second argument and identify that the two applications of $\overline{sum}$ and $\overline{prod}$ are candidate auxiliary information. Thus, we collect the values of these applications and obtain the program $\check{cmp}$:

$$\begin{array}{rl} \check{cmp}(\bar r) = & \textbf{let } v_1 = \overline{sum}(1st(4th(\bar r))) \textbf{ in} \\ & \textbf{let } v_2 = \overline{prod}(1st(2nd(\bar r))) \textbf{ in} \\ & < v_1, v_2 > \end{array} \tag{3}$$

where $\overline{sum}$ and $\overline{prod}$ are defined as in (1).

4.1.4 Step 4. Combining Intermediate Results and Candidate Auxiliary Information

Step 4 merges the programs $\bar f$ and $\check f$. It first defines a program $\bar{\bar f}$ to be the pair of $\bar f$ and $\check f$:

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

and the projection Π_0 to be $\{1^{st}1^{st}\}$. Then it optimizes $\bar{\bar f}$ together with Π_0. The goal is to merge the computation of auxiliary information $\check f$ naturally into the computation of intermediate results $\bar f$, as opposed to two disjoint parallel computations. This involves transformations used in partial evaluation [22, 33], including introducing functions to compute function applications, unfolding, simplifying primitive function applications, driving, replacing recursive applications with introduced functions, etc. The only constraints on the optimization are that $\Pi_0(\bar{\bar f}(x))$ always projects out the value of $f(x)$ and that the values of all components of $\bar f$ and $\check f$ are embedded in the final return value of $\bar{\bar f}$. Thus, we can re-arrange the return components into the most straightforward form.

Example. For the programs $\overline{cmp}$ in (1) and $\check{cmp}$ in (3), we first define

$$\overline{\overline{cmp}}(x) = \ \textbf{let } \bar r = \overline{cmp}(x) \textbf{ in let } \check r = \check{cmp}(\bar r) \textbf{ in } < \bar r, \check r >$$

and projection $\{1^{st}1^{st}\}$, and then we optimize $\overline{\overline{cmp}}(x)$:

1. unfold $\overline{\overline{cmp}}$ and then $\overline{cmp}$ and $\check{cmp}$

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

2. lift **let**'s in the binding of $\bar r$, replace $4th(\bar r)$ by v_2, $2nd(\bar r)$ by v_1, lift **let**'s in the binding of $\check r$

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

3. unfold bindings for $\bar r$, $\check r$, u_1', u_2'

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

and, simplifying the return value and Π_0, we obtain the program $\overline{\overline{cmp}}$, as below, and projection $\{1^{st}\}$.

$$\begin{array}{rl} \overline{\overline{cmp}}(x) = & \textbf{let } v_1 = \overline{odd}(x) \textbf{ in} \\ & \textbf{let } u_1 = \overline{sum}(1st(v_1)) \textbf{ in} \\ & \textbf{let } v_2 = \overline{even}(x) \textbf{ in} \\ & \textbf{let } u_2 = \overline{prod}(1st(v_2)) \textbf{ in} \\ & < 1st(u_1) \le 1st(u_2), v_1, u_1, v_2, u_2, \ \overline{sum}(1st(v_2)), \ \overline{prod}(1st(v_1)) > \end{array} \tag{4}$$

We can see that computing the auxiliary information $c\breve{m}p$ is at least as fast as computing cmp. Thus, the incremental version obtained at the end is guaranteed to be at least as fast as computing from scratch.

4.2 Phase II. Incrementalization

Phase II derives a program $\bar{\bar{f}}'$, an incremental version of $\bar{\bar{f}}$ under $\oplus$. Basically, one may identify subcomputations in the expanded $\bar{\bar{f}}(x \oplus y)$ whose values can be retrieved from the cached result $\bar{\bar{r}}$ of $\bar{\bar{f}}(x)$, replace them by corresponding retrievals, and capture the resulting way of computing $\bar{\bar{f}}(x \oplus y)$ in the incremental version $\bar{\bar{f}}'(x, y, \bar{r})$. Such a derivation method is given in [31], and, depending on the power one expects from the derivation, the method can be made semi-automatic or fully-automatic.

Example. For the program $\bar{\bar{cmp}}$ in (4), after deriving an incremental version of it under $\oplus$:

1. unfold $\bar{\bar{cmp}}(cons(y,x))$

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

2. transform the applications

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

3. replace applications by retrievals

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

we obtain the program $\bar{\bar{cmp}}'$:

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

4.3 Phase III. Pruning

Phase III prunes the programs $\bar{\bar{f}}$ and $\bar{\bar{f}}'$ to compute and maintain only intermediate results and auxiliary information that are useful for computing the value of f incrementally under $\oplus$. Basically, a backward dependence analysis is used to identify subcomputations of $\bar{\bar{f}}'$ whose values are used in computing the value of f, a pruning transformation is used to replace unneeded computations with $\_$, and finally, the resulting programs are optimized by eliminating the $\_$ components [30].

Example. For the programs $\bar{\bar{cmp}}$ in (4) and $\bar{\bar{cmp}}'$ in (5), after pruning, we obtain

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

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

Finally, we optimize the two programs $\bar{\bar{cmp}}$ and $\bar{\bar{cmp}}'$ together, and we obtain the programs $\widetilde{cmp}$ and $\widetilde{cmp}'$:

$$
\begin{aligned}
\widetilde{cmp}(x) = \;& \textbf{let } v_1 = odd(x) \textbf{ in} \\
& \textbf{let } u_1 = sum(v_1) \textbf{ in} \\
& \textbf{let } v_2 = even(x) \textbf{ in} \\
& \textbf{let } u_2 = prod(v_2) \textbf{ in} \\
& < u_1 \le u_2,\ u_1,\ u_2,\ sum(v_2),\ prod(v_1) >
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
\widetilde{cmp}'(y, \tilde{r}) = \;& < y + 4th(\tilde{r}) \le 5th(\tilde{r}), \\
& \quad y + 4th(\tilde{r}),\ 5th(\tilde{r}),\ 2nd(\tilde{r}),\ y * 3rd(\tilde{r}) >
\end{aligned}
\tag{7}
$$

5 Discussion

Multi-Pass Discovery of Auxiliary Information. The program $\tilde{f}$ can sometimes be computed even faster by caching still more auxiliary information, in particular, for incrementally computing the other auxiliary information. For example, suppose multiplication is 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

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

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

$$\begin{aligned}\tilde{f}'(x,\tilde{r}) &= < x{*}x{*}x + 3{*}x{*}x + 3{*}x + 1,\ 3{*}x{*}x + 6{*}x + 3,\ 3{*}x + 3 > \\ &= < 1st(\tilde{r}) + 2nd(\tilde{r}) + 3rd(\tilde{r}) + 1,\ 2nd(\tilde{r}) + 2{*}3rd(\tilde{r}) + 3,\ 3rd(\tilde{r}) + 3 >\end{aligned}$$

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 $3{*}x{*}x$. We obtain $\tilde{f}(x) = < x{*}x{*}x,\ 3{*}x{*}x,\ 3{*}x,\ 6{*}x >$ and

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

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, 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.

To help decide whether or not to use certain auxiliary information, and how much auxiliary information to use, time analysis is crucial. We also need to take into consideration the space consumption and the trade-off between time and space, which is particularly true for space-consuming computations. Further study is needed in these areas.

Implementation. A prototype system, CACHET, has been implemented for most of the transformations used in our approach. The implementation will be described elsewhere.

6 Examples

6.1 Binary Integer Square Root

In [32], a specification of binary integer square root algorithm is transformed into a VLSI circuit. We show how our method can automate the derivation of the finite differencing transformations that are manually discovered and verified in [32]. This is of particular interest in the context of the recent Pentium chip flaw [18].

The initial specification of the l-bit binary integer square root algorithm uses the non-restoring method [17, 32], which is exact for perfect squares and off by at most 1 for other integers:

$$\begin{array}{l} m := 2^{l-1} \\ \textbf{for } i := l-2 \textbf{ downto } 0 \textbf{ do} \\ \quad p := n - m^2; \\ \quad \textbf{if } p > 0 \textbf{ then} \\ \quad\quad m := m + 2^i \\ \quad \textbf{else if } p < 0 \textbf{ then} \\ \quad\quad m := m - 2^i \end{array} \qquad (8)$$

To simplify the presentation, we jump to the heart of the problem, namely, computing $n - m^2$ and 2^i incrementally in each iteration when the change is $m' = m \pm 2^i$ and $i' = i - 1$. Note that here multiplications and exponentials are much more expensive than additions and shifts (doublings or halvings). Let the program f be

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

where $pair$ is a constructor with selectors $fst(a,b)=a$ and $snd(a,b)=b$. Let the input change be

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

Phase I. Step 1. Cache all intermediate results of f.

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

Step 2. Incrementalize $\bar{f}$ under the input change $\oplus$.

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

Step 3. Collect intermediate results of $\bar{f}$ that depend on the old input.

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

Step 4. Combine the collected candidate auxiliary information $\check{f}$ with $\bar{f}$.

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

Phase II. Derive an incremental version of $\bar{\bar{f}}$ under $\oplus$.

$$\begin{aligned} \bar{\bar{f}}'(n,m,i,\bar{\bar{r}}) = \quad & \textbf{let } v = (m \pm 2^i)^2 \textbf{ in let } u = 2^{i-1} \textbf{ in } \; < pair(n-v,u), \; v, \; u, \; 2{*}(m \pm 2^i){*}u, \; u^2 > \\ = \quad & \textbf{let } v = m^2 \pm 2{*}m{*}2^i + (2^i)^2 \textbf{ in let } u = 2^i/2 \textbf{ in } \; < pair(n-v,u), \; v, \; u, \; 2{*}m{*}u \pm 2{*}2^i{*}u, \; u^2 > \\ = \quad & \textbf{let } v = 2nd(\bar{\bar{r}}) \pm 4th(\bar{\bar{r}}) + 5th(\bar{\bar{r}}) \textbf{ in let } u = snd(1st(\bar{\bar{r}}))/2 \textbf{ in} \\ & < pair(fst(1st(\bar{\bar{r}})) \mp 4th(\bar{\bar{r}}) - 5th(\bar{\bar{r}}), \; u), \; v, \; u, \; 4th(\bar{\bar{r}})/2 \pm 5th(\bar{\bar{r}}), \; 5th(\bar{\bar{r}})/4 > \end{aligned}$$

Phase III. Prune the programs $\bar{\bar{f}}$ and $\bar{\bar{f}}'$.

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

$$\bar{\bar{f}}'(n,m,i,\bar{\bar{r}}) = \; < pair(fst(1st(\bar{\bar{r}})) \mp 4th(\bar{\bar{r}}) - 5th(\bar{\bar{r}}), \; snd(1st(\bar{\bar{r}}))/2), \; \_, \; \_, \; 4th(\bar{\bar{r}})/2 \pm 5th(\bar{\bar{r}}), \; 5th(\bar{\bar{r}})/4 >$$

Eliminating the underscores for the two programs, we obtain

$$\tilde{f}(n,m,i) = \quad \textbf{let } u = 2^i \textbf{ in } \; < pair(n-m^2,u), \; 2{*}m{*}u, \; u^2 > \tag{9}$$

$$\tilde{f}'(n,m,i,\tilde{r}) = \; < pair(fst(1st(\tilde{r})) \mp 2nd(\tilde{r}) - 3rd(\tilde{r}), \; snd(1st(\tilde{r}))/2), \; 2nd(\tilde{r})/2 \pm 3rd(\tilde{r}), \; 3rd(\tilde{r})/4 > \tag{10}$$

Thus, the expensive multiplications and exponentials in each iteration are completely replaced by additions and shifts. Following our systematic approach, we have even eliminated an extra shift done in [32].

6.2 Local Neighborhood Problems

In image processing, computing information about local neighborhoods is common [49, 51, 53, 54]. A simple but typical example is the local summation problem [51, 53]: given an n-by-n image, compute, for each pixel, the local sum of its m-by-m neighborhood. The naive algorithm takes $O(n^2m^2)$ time, while an efficient algorithm using dynamic programming takes $O(n^2)$ time. We show how to obtain such an efficient algorithm systematically following our approach.

The naive algorithm, call it f, is straightforward:

```
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
```
(11)

For simplicity, initializations for the array margins are ignored.

6.2.1 Inner Loop

To improve f, we first improve the inner loop, call it g, with any fixed index i for the outer loop. Our approach obtains an incremental program for the program g_1, defined below, under input change $j_2' = j_2+1$, and uses the incremental program for each iteration of g.

```
for j := 0 to j2 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
```

Phase I. Caching all intermediate results and candidate auxiliary information, we obtain the program $\bar{\bar{g_1}}$ (same as $\bar{g_1}$) as follows:

1. save results of the innermost loop

```
for j := 0 to j2 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 j2 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
```

Phase II. Incremetalizing $\bar{\bar{g_1}}$ on input j_2+1, we obtain the program $\bar{\bar{g_1}}'$ as follows:

1. unfold $\bar{\bar{g_1}}$ on new index j_2+1

```
for j := 0 to j2 + 1 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
```

2. save computing iterations 1 to j_2

```
sum := 0;
for k := 0 to m do
   row := 0;
   for l := 0 to m do
      row := row + a[i+k, j2+1+l]
   c[i, j2+1, k] := row
   sum := sum + c[i, j2+1, k]
b[i, j2+1] := sum
```

3. $c[i, j_2+1, k] = c[i, j_2, k] - a[i+k, j_2] + a[i+k, j_2+1+m]$

```
sum := 0;
for k := 0 to m do
   row := 0;
   for l := 0 to m do
      row := row + a[i+k, j2+1+l]
   c[i, j2+1, k] := c[i, j2, k] - a[i+k, j2] + a[i+k, j2+1+m]
   sum := sum + c[i, j2+1, k]
b[i, j2+1] := sum
```

Phase III. Pruning the program $\bar{\bar{g_1}}'$, we obtain the program $\tilde{g_1}'$ as follows:

```
sum := 0;
for k := 0 to m do
   c[i, j2+1, k] := c[i, j2, k] - a[i+k, j2] + a[i+k, j2+1+m]
   sum := sum + c[i, j2+1, k]
b[i, j2+1] := sum
```

Finally, an optimized version of g is obtained by adding the outside loop for j from 0 to n to $\tilde{g_1}'$ and replacing j_2+1 by j (and thus j_2 by $j-1$):

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

Now, using the optimized g in f, we obtain an optimized program f as follows:

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

6.2.2 Outer Loop

We then improve the outer loop of the new program f above. Our approach obtains an incremental program for the program f_1, defined below, under input change $i_1' = i_1 + 1$, and uses the incremental program for each iteration of the new f.

```
for i := 0 to i1 do
  for j := 1 to n do
    sum := 0;
    for k := 0 to m do
      c[i,j,k] := c[i,j-1,k] - a[i+k,j-1] + a[i+k,j+m]
      sum := sum + c[i,j,k]
    b[i,j] := sum
```

Phase I. Caching all intermediate results and discovering candidate auxiliary information, we obtain the program $\bar{\bar{f}}_1$ same as above, since all its intermediate results have been cached and no auxiliary information is discovered.

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

1. unfold $\bar{\bar{f}}_1$ on new index $i_1 + 1$

```
for i := 0 to i1+1 do
  for j := 0 to n do
    sum := 0;
    for k := 0 to m do
      c[i,j,k] := c[i,j-1,k] - a[i+k,j-1] + a[i+k,j+m]
      sum := sum + c[i,j,k]
    b[i,j] := sum
```

2. save computing iterations 1 to i_1

```
for j := 0 to n do
  sum := 0;
  for k := 0 to m do
    c[i1+1,j,k] := c[i,j-1,k] - a[i+k,j-1] + a[i+k,j+m]
    sum := sum + c[i1+1,j,k]
  b[i1+1,j] := sum
```

3. $b[i_1+1, j] = b[i_1, j] - c[i_1, j, 0] + c[i_1+1+m, j, 0]$

```
for j := 0 to n do
  sum := 0;
  for k := 0 to m do
    c[i,j,k] := c[i,j-1,k] - a[i+k,j-1] + a[i+k,j+m]
    sum := sum + c[i,j,k]
  b[i,j] := b[i1,j] - c[i1,j,0] + c[i1+1+m,j,0]
```

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

1. prune unneeded code

```
for j := 0 to n do
  c[i,j,0] := c[i,j-1,0] - a[i+0,j-1] + a[i+0,j+m]
  b[i,j] := b[i1,j] - c[i1,j,0] + c[i1+1+m,j,0]
```

2. prune unneeded array dimension

```
for j := 0 to n do
  c[i,j] := c[i,j-1] - a[i,j-1] + a[i,j+m]
  b[i,j] := b[i1,j] - c[i1,j] + c[i1+1+m,j]
```

Finally, an optimized version of f is obtained by adding the outside loop for i from 0 to i_1 to $\tilde{f}_1'$ and replacing $i_1 + 1$ by i (and thus i_1 by $i - 1$):

```
for i := 1 to n do
  for j := 1 to n do
    c[i,j] := c[i,j-1] - a[i,j-1] + a[i,j+m]
    b[i,j] := b[i-1,j] - c[i-1,j] + c[i+m,j]
```
(12)

In summary, 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.

6.3 Path Sequence Problem

This example is taken from [7]. 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 where an exponential-time recursive solution is improved (incorrectly in [7] but corrected in [8]).

A program *llp* is defined to compute the desired length. The input string is given explicitly as the argument, and the input graph is given as a predicate arc such that 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. The program *llp*, as given, takes exponential time:

$$\begin{array}{ll} llp(l) = & \textbf{if } null(l) \textbf{ then } 0 \\ & \textbf{else } \max(llp(cdr(l)), 1 + f(car(l), cdr(l))) \end{array} \qquad \begin{array}{ll} f(n,l) = & \textbf{if } null(l) \textbf{ then } 0 \\ & \textbf{else if } \mathrm{arc}(n, car(l)) \textbf{ then} \\ & \quad \max(f(n, cdr(l)), 1 + f(car(l), cdr(l))) \\ & \textbf{else } f(n, cdr(l)) \end{array}$$

We formulate the problem as computing *llp* incrementally under the input change $l \oplus i = cons(i, l)$. We ignore the detail of the derivation due to the space limit and give the resulting program as follows:

$$\begin{array}{ll} \widetilde{llp}(l) = & \textbf{if } null(l) \textbf{ then } <0> \\ & \textbf{else let } v_2 = \tilde{f}(car(l), cdr(l)) \textbf{ in} \\ & \quad <\max(llp(cdr(l)), 1 + 1st(v_2)),\ v_2> \end{array} \qquad \begin{array}{ll} \tilde{f}(i,l) = & \textbf{if } null(l) \textbf{ then } <0> \\ & \textbf{else let } u_2 = \tilde{f}(car(l), cdr(l)) \textbf{ in} \\ & \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\ & \quad\quad <\max(f(i, cdr(l)), 1 + 1st(u_2)),\ u_2> \\ & \quad \textbf{else } <f(i, cdr(l)),\ u_2> \end{array} \tag{13}$$

$$\begin{array}{ll} \widetilde{llp}'(i,l,\tilde{r}) = & \textbf{if } null(l) \textbf{ then } <1, <0>> \\ & \textbf{else let } v_2 = \tilde{f}'(i, l, 2nd(\tilde{r})) \textbf{ in} \\ & \quad <\max(1st(\tilde{r}), 1+1st(v_2)),\ v_2> \end{array} \qquad \begin{array}{ll} \tilde{f}'(i,l,\tilde{r_1}) = & \textbf{if } null(cdr(l)) \textbf{ then} \\ & \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then } <1, <0>> \\ & \quad \textbf{else } <0, <0>> \\ & \textbf{else let } u_1 = \tilde{f}'(i, cdr(l), 2nd(\tilde{r_1})) \textbf{ in} \\ & \quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\ & \quad\quad <\max(1st(u_1), 1+1st(\tilde{r_1})),\ \tilde{r_1}> \\ & \quad \textbf{else } <1st(u_1),\ \tilde{r_1}> \end{array} \tag{14}$$

Since $\widetilde{llp}'(i,l,\tilde{r})$ calls $\tilde{f}'$ to go through the list l once for each element, computing $\widetilde{llp}'(i,l,\tilde{r})$ takes $O(n)$ time, where n is the length of l, but computing $llp(cons(i,l))$ from scratch takes exponential time.

Now, we return to the original program. Note that we have $llp(l) = 1st(\widetilde{llp}(l))$ 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 (14) in this last equation, we redefine $\widetilde{llp}$ as follows:

$$\begin{array}{ll} \widetilde{llp}(cons(i,l)) = & \textbf{if } null(l) \textbf{ then } <1, <0>> \\ & \textbf{else let } \tilde{r} = \widetilde{llp}(l) \textbf{ in} \\ & \quad \textbf{let } v_2 = \tilde{f}'(i, l, 2nd(\tilde{r})) \textbf{ in} \\ & \quad <\max(1st(\tilde{r}), 1 + 1st(v_2)),\ v_2> \end{array} \tag{15}$$

If $l = nil$, then $\widetilde{llp}(l) = <0>$, otherwise, letting $l = cons(i, l_1)$ and using (15), we obtain a new definition of $\widetilde{llp}$:

$$\begin{array}{ll} \widetilde{llp}(l) = & \textbf{if } null(l) \textbf{ then } <0> \\ & \textbf{else if } null(cdr(l)) \textbf{ then } <1, <0>> \\ & \textbf{else let } \tilde{r} = \widetilde{llp}(cdr(l)) \textbf{ in} \\ & \quad \textbf{let } v_2 = \tilde{f}'(car(l), cdr(l), 2nd(\tilde{r})) \textbf{ in} \\ & \quad <\max(1st(\tilde{r}), 1 + 1st(v_2)),\ v_2> \end{array} \tag{16}$$

where $\tilde{f}'$ is defined as in (14). Since this new program $\widetilde{llp}$ calls $\tilde{f}'$ only $O(n)$ times, it takes $O(n^2)$ time.

7 Related Work

Work related to our analysis and transformation techniques used has been discussed while presenting them. Related work on incremental computation that exploits intermediate results and auxiliary information is summarized in Section 1. Here, we take a closer look at related work in discovering auxiliary information for incremental computation.

Interactive systems and reactive systems often use various incremental algorithms to achieve fast response time [5, 6, 9, 13, 19, 24, 41, 42]. Explicit incremental algorithms are hard to write and appropriate auxiliary information is hard to discover. Our approach provides a general and systematic approach for developing particular incremental algorithms. For example, for the dynamic incremental attribute evaluation algorithm in [43], the characteristic graph is auxiliary information that would be discovered following our principle. For static incremental attribute evaluation algorithms [26, 27], where no auxiliary information is needed, our approach can cache intermediate results and maintain them automatically [30].

Strength reduction [3, 11, 46] 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 multiplication and addition are used on induction variables and region constants. Our method can be viewed as a principled strength reduction technique not limited to a fixed set of pre-known rules, even allowing such rules to be derived and justified when necessary, as shown in the integer square root example.

Finite differencing, proposed by Paige [34, 35, 36], can be regarded as a generalization of strength reduction to set-theoretic expressions for systematic program development. Basically, a set of 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, where corresponding dynamic expressions need to be discovered and rules for maintaining them derived. These dynamic expressions are a certain kind of auxiliary information. How to discover them is a problem that remains to be studied, but once discovered, our method can be used to derive rules that maintain this information. Also, in general, Paige's rules apply only to very high-level languages like set expressions; our method applies also to standard languages like Lisp.

The *promotion and accumulation strategies* are proposed by Bird [7, 8] 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 formulated as deriving incremental programs, and accumulation as identifying appropriate intermediate results or auxiliary information. However, we can discern no systematic steps being followed in Bird's derivation; such derivations are error-prone [8]. As demonstrated with the path sequence problem, our approach can be regarded as a systematic formulation of the promotion and accumulation strategies.

Other work in transformational program development, including the *extension* technique [12], the differencing strategy in CIP [37], and the finite differencing of functional programs in KIDS [45], can be further automated with our systematic approach. These techniques are indispensable for developing efficient programs.

References

[1] *IEEE Standard Glossary of software engineering terminology.* IEEE Standard 729. 1983.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[3] F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 79–101. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[4] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, January 1990.

[5] R. Bahlke and G. 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.

[6] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The *Pan* language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.

[7] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

[8] R. S. Bird. Addendum: The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 7(3):490–492, July 1985.

[9] 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).

[10] 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.

[11] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[12] N. Dershowitz. *The evolution of programs*, volume 5 of *Progress in Computer Science*. Birkhäuser, Boston, 1983.

[13] 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.

[14] J. Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1:321–342, 1976.

[15] 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.

[16] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LFP*, pages 307–322, 1990.

[17] I. Flores. *The logic of computer arithmetic*. Prentice-Hall international series in electrical engineering. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[18] J. Glanz. Mathematical logic flushes out the bugs in chip designs. *Science*, 267:332–333, January 20 1995.

[19] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[20] R. Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN '92 Conference on PLDI*, pages 261–272, California, June 1992.

[21] F. Jalili and J. H. Gallier. Building friendly parsers. In *Conference Record of the 9th Annual ACM Symposium on POPL*, pages 196–206, Albuquerque, New Mexico, January 1982.

[22] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[23] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, May 1985. Springer-Verlag. LNCS 202.

[24] G. E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):168–193, April 1989.

[25] S. M. Kaplan and G. E. Kaiser. Incremental attribute evaluation in distributed language-based environments. In *Conference Record of the ACM Symposium on PODC*, Calgary, Canada, August 1986.

[26] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.

[27] T. Katayama. Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345–369, July 1984.

[28] J. Launchbury. Projections for specialisation. In *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, 1988.

[29] 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.

[30] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on on PEPM*, La Jolla, California, June 1995.

[31] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.

[32] J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In *Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design — Theory, Practice and Experience*, Bad Herrenalb (Black Forest), Germany, September 1994. Springer-Verlag. LNCS.

[33] F. G. Pagan. *Partial computation and the construction of language processors.* Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[34] B. 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.

[35] R. Paige. Transformational programming – applications to algorithms and systems. In *Conference Record of the 10th Annual ACM Symposium on POPL*, pages 73–87, January 1983.

[36] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[37] H. A. Partsch. *Specification and Transformation of Programs - A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[38] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Transactions on Programming Languages and Systems*, 14(2):173–200, April 1992.

[39] W. Pugh. An improved cache replacement strategy for function caching. In *Proceedings of the ACM '88 Conference on LFP*, pages 269–276, 1988.

[40] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on POPL*, pages 315–328, January 1989.

[41] S. 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 144–156, Montreal, Canada, June 1984.

[42] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1988.

[43] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[44] 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.

[45] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[46] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *Proceedings of the 4th International Joint Conference on TAPSOFT*, pages 394–415, Brighton, UK, 1991. Springer-Verlag. LNCS 494.

[47] R. S. Sundaresh and P. Hudak. Incremental computation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on POPL*, pages 1–13, January 1991.

[48] 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.

[49] J. Webb. Steps towards architecture-independent image processing. *IEEE Computer*, February 1992.

[50] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[51] W. 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.

[52] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991.

[53] R. Zabih. Individuating unknown objects by combining motion and stereo. Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, California, 1994.

[54] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In J.-O. Eklundh, editor, *3rd European Conference on Computer Vision*, pages 151–158. Springer-Verlag, 1994. LNCS 801.