Discovering Auxiliary Information for Incremental Computation

Yanhong A. Liu* Scott D. Stoller† Tim Teitelbaum*
Department of Computer Science, Cornell University, Ithaca, New York 14853
{yanhong,stoller,tt}@cs.cornell.edu

Abstract

This paper presents program analyses and transformations that discover a general class of auxiliary information for any incremental computation problem. Combining these techniques with previous techniques 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. Applications of the approach include strength reduction in optimizing compilers and finite differencing in transformational programming.

1 Introduction

Importance of incremental computation. In essence, every program computes by fixed-point iteration, expressed as recursive functions or loops. This is why loop optimizations are so important. A loop body can be regarded as a program f parameterized by an induction variable x that is incremented on each iteration by a change operation $\oplus$. Efficient iterative computation relies on effective use of state, i.e., computing the result of each iteration using stored results of previous iterations. This is why strength reduction [2] and related techniques [48] are crucial for performance.

Given a program f and an input change operation $\oplus$, a program f' that computes $f(x \oplus y)$ efficiently by using the result of the previous computation of $f(x)$ is called an *incremental version* of f under $\oplus$. Sometimes, information other than the result of $f(x)$ needs to be maintained and used for efficient incremental computation of $f(x \oplus y)$. We call a function that computes such information an *extended version* of f. Thus, the goal of computing loops efficiently corresponds to constructing an extended version of a program f and deriving an incremental version of the extended version under an input change operation $\oplus$.

In general, incremental computation aims to solve a problem on a sequence of inputs that differ only slightly from one another, making use of the previously computed output in computing a new output, instead of computing the new output from scratch. Incremental computation is a fundamental issue relevant throughout computer software, e.g., optimizing compilers [1, 2, 15, 20, 60], transformational program development [7, 17, 47, 49, 59], and interactive systems [4, 5, 9, 19, 27, 33, 53, 54]. Numerous techniques for incremental computation have been developed, e.g., [2, 3, 22, 28, 29, 30, 41, 48, 51, 52, 55, 58, 61, 64].

Deriving incremental programs. We are engaged in an ambitious effort to *derive* incremental extended programs automatically (or semi-automatically) from non-incremental programs written in standard programming languages. This approach contrasts with many other approaches that aim to *evaluate* non-incremental programs incrementally. We have partitioned the problem (thus far) into three subproblems:

- P1. Exploiting the *result*, i.e., the return value, of $f(x)$.
- P2. Caching, maintaining, and exploiting *intermediate results* of the computation $f(x)$.
- P3. Discovering, computing, maintaining, and exploiting *auxiliary information* about x, i.e., information not computed by $f(x)$.

Our current approaches to problems P1 and P2 are described in [41] and [40], respectively. In this paper, we address issue P3 for the first time and contribute:

- A novel proposal for finding auxiliary information.
- A comprehensive methodology for deriving incremental programs that addresses all three subproblems.

Some approaches to incremental computation have exploited specific kinds of auxiliary information, e.g., auxiliary arithmetic associated with some classical strength-reduction rules [2], dynamic mappings maintained by finite differencing rules for aggregate primitives in SETL [48] and INC [64], and auxiliary data structures for problems with certain properties like stable decomposition [52]. However, until now,

*The author gratefully acknowledges the support of the Office of Naval Research under contract No. N00014-92-J-1973.

†Supported in part by NSF/DARPA Grant No. CCR-9014363, NASA/DARPA Grant NAG-2-893, and AFOSR Grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

****

Appears in Proceedings of POPL'96: the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 21-24, 1996.

systematic discovery of auxiliary information for arbitrary programs has been a subject completely open for study.

Auxiliary information is, by definition, useful information about x that is *not* computed by $f(x)$. Where, then, can one find it? The key insight of our proposal is:

> **A.** Consider, as candidate auxiliary information for f, all intermediate computations of an incremental version of f that depend only on x; such an incremental version can be obtained using some techniques we developed for solving P1 and P2.[1]

How can one discover which pieces of candidate auxiliary information are useful and how they can be used? We propose:

> **B.** Extend f with all candidate auxiliary information, then apply some techniques used in our methods for P1 and P2 to obtain an extended version and an incremental extended version that together compute, exploit, and maintain only useful intermediate results and useful auxiliary information.

Thus, on the one hand, one can regard the method for P3 in this paper as an extension to methods for P1 and P2. On the other hand, one can regard methods for P1 and P2 (suitably revised for their different applications here) as aids for solving P3. The modular components complement one another to form a comprehensive principled approach for incremental computation and therefore also for efficient iterative computation generally. Although the entire approach seems complex, each module or step is simple.

We summarize here the essence of our methods:

P1. In [41], we gave a systematic transformational approach for deriving an incremental version f' of a program f under 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 $f'(x, y, r)$.

P2. In [40], we gave a method, called *cache-and-prune*, for statically transforming programs to cache all intermediate results useful 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 $\bar{f}'$ of $\bar{f}$ using our method for P1, and (III) analyze the dependencies in $\bar{f}'$, then 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 only the useful intermediate results.

P3. This paper presents a two-phase method that discovers a general class of auxiliary information for any incremental computation problem. The two phases correspond to A and B above. For Phase A, we have developed an embedding analysis that helps avoid including redundant information in an extended version, and we have exploited a forward dependence analysis that helps identify candidate auxiliary information. All the program analyses and transformations used in this method are combined with considerations for caching intermediate results, so we obtain incremental extended programs that exploit and maintain intermediate results as well as auxiliary information.

We illustrate our approach by applying it to problems in list processing, VLSI design, and graph algorithms.

The rest of this paper is organized as follows. Section 2 formulates the problem. Section 3 discusses discovering candidate auxiliary information. Section 4 describes how candidate auxiliary information is used. Two examples are given in Section 6. Finally, we discuss related work and conclude in Section 7.

2 Formulating the problem

We use a simple first-order functional programming language, with expressions given by the following grammar:

$e ::= v$	variable
$\mid\ c(e_1, ..., e_n)$	constructor application
$\mid\ p(e_1, ..., e_n)$	primitive function application
$\mid\ f(e_1, ..., e_n)$	function application
$\mid$ **if** e_1 **then** e_2 **else** e_3	conditional expression
$\mid$ **let** $v = e_1$ **in** e_2	binding expression

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

$$f(v_1, ..., v_n) = e \qquad (1)$$

and a function f_0 that is to be evaluated with some input $x = \langle x_1, ..., x_n \rangle$. Figure 1 gives some example definitions. The semantics of the language is strict.

An input change operation $\oplus$ to a function 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 to the function *cmp* of Figure 1 may be defined by $x' = x \oplus y = cons(y, x)$.

We use an asymptotic cost model for measuring time complexity and write $t(f(v_1, ..., v_n))$ to denote the asymptotic time of computing $f(v_1, ..., v_n)$. Thus, assuming all primitive functions take constant time, it is sufficient to consider only the values of function applications as candidate information to cache. Of course, maintaining extra information takes extra space. Our primary goal is to improve the asymptotic running time of the incremental computation. 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 use the approach in [41] 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.[2] For example, for the function *sum* of Figure 1 and input change operation $x \oplus y = cons(y, x)$, the function *sum'* in Figure 2 is derived.

[1] We use techniques developed for solving P1 and P2, instead of just P1, so that the candidate auxiliary information includes auxiliary information useful for efficiently maintaining the intermediate results.

[2] 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 [41].

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

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

Figure 1: Example function definitions

In order to use also intermediate results of $f_0(x)$ to compute $f_0(x \oplus y)$ possibly faster, we use the approach in [40] to cache useful intermediate results of f_0 and obtain a program that incrementally computes the return value and maintains these intermediate results. For example, for the function cmp of Figure 1 and input change operation $x \oplus \langle y_1, y_2 \rangle = cons(y_1, cons(y_2, x))$, the intermediate results $sum(odd(x))$ and $prod(even(x))$ are cached, and the functions $\widehat{cmp}$ and $\widehat{cmp}'$ in Figure 2 are obtained.

However, auxiliary information other than the intermediate results of $f_0(x)$ is sometimes needed to compute $f_0(x \oplus y)$ quickly. For example, for the function cmp of Figure 1 and input change operation $x \oplus y = cons(y, x)$, the values of $sum(even(x))$ and $prod(odd(x))$ are crucial for computing $cmp(cons(y, x))$ incrementally but are not computed in $cmp(x)$. Using the method in this paper, we can derive the functions $\widetilde{cmp}$ and $\widetilde{cmp}'$ in Figure 2 that compute these pieces of auxiliary information, use them in computing $cmp(cons(y, x))$, and maintain them as well. $\widetilde{cmp}'$ computes incrementally using only $O(1)$ time. We use this example as a running example.

Notation. We use <> to construct tuples that bundle intermediate results and auxiliary information with the original return value of a function. The selector nth returns the nth element of such a tuple.

We use x to denote the previous input to f_0; r, the cached result of $f_0(x)$; y, the input change parameter; x', the new input $x \oplus y$; and f_0', an incremental version of f_0 under $\oplus$. We let $\bar{f}_0$ return all intermediate results of f_0, and let $\check{f}_0$ return candidate auxiliary information for f_0 under $\oplus$. We use $\bar{\check{f}}_0$ to denote a function that returns all intermediate results and candidate auxiliary information; $\bar{\check{r}}$, the cached result of $\bar{\check{f}}_0(x)$; and $\bar{\check{f}}_0'$, an incremental version of $\bar{\check{f}}_0$ under $\oplus$. Finally, we use $\widetilde{f}_0$ to denote a function that returns only the useful intermediate results and auxiliary information; $\widetilde{r}$, the cached result of $\widetilde{f}_0(x)$; and $\widetilde{f}_0'$, a function that incrementally maintains only the useful intermediate results and auxiliary information. Note that (useful) intermediate results include the original return value. Figure 3 summarizes the notation.

function	return value	denoted as	incremental function
f_0	original value	r	f_0'
$\bar{f}_0$	all i.r.	$\bar{r}$	
$\check{f}_0$	candidate a.i.		
$\bar{\check{f}}_0$	all i.r. & candidate a.i.	$\bar{\check{r}}$	$\bar{\check{f}}_0'$
$\widetilde{f}_0$	useful i.r. & useful a.i.	$\widetilde{r}$	$\widetilde{f}_0'$

Figure 3: Notation

3 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 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 $\bar{f}_0^{\backslash}$ that exposes candidate auxiliary information. Step 3 constructs a function $\check{f}_0$ that computes only the candidate auxiliary information in $\bar{f}_0^{\backslash}$.

3.1 Step A.1: Caching all intermediate results

Extending f_0 to cache all intermediate results uses the transformations in Stage I of [40]. It first performs a straightforward *extension transformation* to embed all intermediate results in the final return value and then performs administrative simplifications.

Certain improvements to the extension transformation are suggested, although not given, in [40] to avoid caching redundant intermediate results, i.e., values of function applications that are already embedded in the values of their enclosing computations, since these omitted values can be re-

If $sum(x) = r$, then $sum'(y, r) = sum(cons(y, x))$.
For x of length n, $sum'(y, r)$ takes time $O(1)$;
$sum(cons(y, x))$ takes time $O(n)$.

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

If $\widehat{cmp}(x) = \hat{r}$, then $\widehat{cmp}'(y_1, y_2, \hat{r}) = \widehat{cmp}(cons(y_1, cons(y_2, x)))$.
For x of length n, $\widehat{cmp}'(y_1, y_2, \hat{r})$ takes time $O(1)$;
$\widehat{cmp}(cons(y_1, cons(y_2, x)))$ takes time $O(n)$.

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

If $\widetilde{cmp}(x) = \widetilde{r}$, 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 $O(n)$.

$$
\begin{array}{lcl}
sum'(y, r) & = & y + r \\
\widehat{cmp}(x) & = & \textbf{let } v_1 = sum(odd(x)) \textbf{ in} \\
 & & \textbf{let } v_2 = prod(even(x)) \textbf{ in} \\
 & & < v_1 \le v_2,\ v_1,\ v_2 > \\
\widehat{cmp}'(y_1, y_2, \hat{r}) & = & \textbf{let } v_1 = y_1 + 2nd(\hat{r}) \textbf{ in} \\
 & & \textbf{let } v_2 = y_2 * 3rd(\hat{r}) \textbf{ in} \\
 & & < v_1 \le v_2,\ v_1,\ v_2 > \\
\widetilde{cmp}(x) & = & \textbf{let } v_1 = odd(x) \textbf{ in} \\
 & & \textbf{let } u_1 = sum(v_1) \textbf{ in} \\
 & & \textbf{let } v_2 = even(x) \textbf{ in} \\
 & & \textbf{let } u_2 = prod(v_2) \textbf{ in} \\
 & & < u_1 \le u_2,\ u_1,\ u_2,\ sum(v_2),\ prod(v_1) > \\
\widetilde{cmp}'(y, \widetilde{r}) & = & < y + 4th(\widetilde{r}) \le 5th(\widetilde{r}), \\
 & & \quad y + 4th(\widetilde{r}),\ 5th(\widetilde{r}),\ 2nd(\widetilde{r}),\ y * 3rd(\widetilde{r}) >
\end{array}
$$

Figure 2: Resulting function definitions

trieved from the results of the enclosing applications. These improvements are 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.

We first briefly describe the extension transformation in [40]; then, we describe an embedding analysis that leads to the desired improvements to the extension transformation.

Extension transformation. Basically, for each function definition $f(v_1, ..., v_n) = e$, we construct a function definition

$$\bar{f}(v_1, ..., v_n) = \mathcal{E}xt[\![e]\!] \qquad (2)$$

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. We assume that each introduced binding uses a fresh variable name. For a constructed tuple $<>$, while we use $1st$ to return the first element, which is the original return value, we use rst to return a tuple of the remaining elements, which are the corresponding intermediate results here. We use an infix operation @ to concatenate two tuples. For transforming a conditional expression, the transformation $\mathcal{P}ad[\![e]\!]$ generates a tuple of $\_$'s of length equal to the number of the function applications in e, where $\_$ is a dummy constant that just occupies a spot. The length of the tuple generated by $\mathcal{P}ad[\![e]\!]$ can easily be determined statically. The use of $\mathcal{P}ad$ ensures that each possible intermediate result appears in a fixed position independent of value of the Boolean expression.

Administrative simplifications are performed on the resulting functions to simplify tuple operations for passing intermediate results, unwind binding expressions that become unnecessary as a result of simplifying their subexpressions, and lift bindings out of enclosing expressions whenever possible to enhance readability.

The following improvements can be made to the above brute-force caching of all intermediate results. First, 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. 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:

$$
\begin{array}{l}
\text{if } Mf(f,i) = true, \text{ then there exists an expression } f_i^{-1} \\
\quad \text{such that, if } u = f(v_1, ..., v_n), \text{ then } v_i = f_i^{-1}(u) \\
\text{if } Me(e,v) = true, \text{ then there exists an expression } e_v^{-1} \\
\quad \text{such that, if } u = e, \text{ then } v = e_v^{-1}(u)
\end{array} \qquad (3)
$$

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. 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, $if_{e_2 e_3}^{e_1}$ denotes $true$ if the value of e_1 can be determined statically or inferred from the value

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

Figure 4: Definition of $\mathcal{E}xt$

$$
\begin{array}{lcl}
Me(u,v) & = & \begin{cases} true & \text{if } v = u \\ false & \text{otherwise} \end{cases} \\
Me(c(e_1,...,e_n),v) & = & Me(e_1,v) \ \vee \ ... \ \vee \ Me(e_n,v) \\
Me(p(e_1,...,e_n),v) & = & \big(\exists p_1^{-1} \wedge Me(e_1,v)\big) \ \vee \ ... \ \vee \ \big(\exists p_n^{-1} \wedge Me(e_n,v)\big) \\
Me(f(e_1,...,e_n),v) & = & \big(Mf(f,1) \wedge Me(e_1,v)\big) \ \vee \ ... \ \vee \ \big(Mf(f,n) \wedge Me(e_n,v)\big) \\
Me(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, v) & = & if^{e_1}_{e_2e_3} \wedge \big(e_1 \vdash Me(e_2,v)\big) \ \wedge \ \big(e_1 \vdash Me(e_3,v)\big) \\
Me(\textbf{let } u = e_1 \textbf{ in } e_2, v) & = & Me(e_2,v) \ \vee \ \big(Me(e_1,v) \wedge Me(e_2,u)\big)
\end{array}
$$

Figure 5: Definition of Me

of **if** e_1 **then** e_2 **else** e_3, and $false$ otherwise. For example, $if^{e_1}_{e_2e_3}$ is true if e_1 is T (for true) or F (for false), 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 $if^{e_1}_{e_2e_3}$ is true.

We can easily show by induction that the safety requirements (3) 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 pointwise extension of 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 $Mtag(e)$ with each subexpression e of e_f, indicating whether the value of e is embedded in the value of 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 6; the tags of other subexpressions of e_f are defined to be $false$. These tags can be computed directly once the above embedding relations are computed.

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

$$
Mall(f) = \bigwedge_{\substack{\text{all function applications} \\ g(e_1,...,e_n) \text{ occurring in } e_f}} Mtag(g(e_1,...,e_m)) \wedge Mall(g) \tag{4}
$$

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 (4) 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.

If there is a function not all of whose intermediate results are embedded in its return value, then an extended

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

Figure 6: Definition of $Mtag$

function for it needs to be defined as in (2). 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:

$$\begin{array}{ll} \mathcal{E}xt[\![f(e_1,...,e_n)]\!] & \\ = \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} & \\ \quad \textbf{let } v = f(1st(v_1),...,1st(v_n)) \textbf{ in} & (5) \\ \quad < v > @ \; rst(v_1) \; @ \; ... \; @ \; rst(v_n) \; @ \; < v > & \end{array}$$

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:

$$\begin{array}{ll} \mathcal{E}xt[\![f(e_1,...,e_n)]\!] & \\ = \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} & (6) \\ \quad < f(1st(v_1),...,1st(v_n)) > @ \; rst(v_1) \; @ \; ... \; @ \; rst(v_n) & \end{array}$$

To summarize, the transformation $\mathcal{E}xt$ remains the same as in Figure 4 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 (6); 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); otherwise define $\mathcal{E}xt[\![f(e_1,...,e_n)]\!]$ as in Figure 4. Note that 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 1, this improved extension transformation yields the following functions:

$$\begin{array}{lll} \overline{cmp}(x) = & \textbf{let } v_1 = odd(x) \textbf{ in} & \\ & \textbf{let } u_1 = \overline{sum}(v_1) \textbf{ in} & \\ & \textbf{let } v_2 = even(x) \textbf{ in} & \\ & \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} & \\ & < 1st(u_1) \leq 1st(u_2), v_1, u_1, v_2, u_2 > & \\ \overline{sum}(x) = & \textbf{if } null(x) \textbf{ then } < 0, \_ > & (7) \\ & \textbf{else let } v_1 = \overline{sum}(cdr(x)) \textbf{ in} & \\ & \quad < car(x) + 1st(v_1), \; v_1 > & \\ \overline{prod}(x) = & \textbf{if } null(x) \textbf{ then } < 1, \_ > & \\ & \textbf{else let } v_1 = \overline{prod}(cdr(x)) \textbf{ in} & \\ & \quad < car(x) * 1st(v_1), \; v_1 > & \end{array}$$

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

3.2 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 [41] that derive 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 [41] are dropped here but are picked up after Step A.3, as discussed in Section 5.

In particular, in [41], 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 [41], 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 [41], we denote them by $f^{\backprime}$.

For the function $\overline{cmp}$ in (7) and input change operation $x \oplus y = cons(y, x)$, we transform the computation of $\overline{cmp}(cons(y, x))$, with $\overline{cmp}(x) = \bar{r}$:

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

2. unfold odd, $\overline{sum}$, $even$ and simplify

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

3. replace applications of $even$ and odd by retrievals

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

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

$$
\begin{array}{ll}
\overline{cmp}^{\backprime}(y, \bar{r}) = & \textbf{let } u_1' = \overline{sum}(4th(\bar{r})) \textbf{ in} \\
& \textbf{let } u_2 = \overline{prod}(2nd(\bar{r})) \textbf{ in} \\
& < y{+}1st(u_1') \leq 1st(u_2), \\
& \quad cons(y, 4th(\bar{r})),\ < y{+}1st(u_1'), u_1' >, \\
& \quad 2nd(\bar{r}),\ u_2 >
\end{array} \tag{8}
$$

3.3 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 dependence analysis. First, we use a *forward dependence analysis* to identify subcomputations of $\bar{f}_0^{\backprime}(x, y, \bar{r})$ that depend only on x and $\bar{r}$. The analysis is in the same spirit as binding-time analysis [32, 37] 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 7, where $FV(e)$ denotes the set of free variables in e and is defined as follows:

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

To compute these sets, we start with $\Sigma_{\bar{f}_0^{\backprime}}$ containing the indices of the arguments of $\bar{f}_0^{\backprime}$ 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 obtain $\Sigma_{\overline{cmp}^{\backprime}} = \{2\}$ and $\Sigma_{\overline{sum}} = \Sigma_{\overline{prod}} = \{1\}$. For every subexpression e in the definition of $\overline{cmp}^{\backprime}(y, \bar{r})$, $\bar{r} \in \Sigma_{[e]}$. For every subexpression e in the definitions of $\overline{sum}(x)$ and $\overline{prod}(x)$, $\Sigma_{[e]} = \{x\}$.

Collection transformation. Next, we 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 $\bar{f}_0^{\backprime}$ and such that $\Sigma_f \neq \emptyset$, we construct a function definition

$$
\check{f}(v_{i_1}, ..., v_{i_k}) = \mathcal{C}ol[\![e]\!] \tag{9}
$$

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 8, where $\mathcal{P}\check{a}d[\![e]\!]$ 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.

Although this forward dependence analysis is equivalent to binding time analysis, the application here is different from that in partial evaluation [31]. 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, we construct a program that computes only on the "static" arguments. In this respect, the resulting program obtained here is similar to the slice obtained from forward slicing [63]. However, our forward dependence 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. Furthermore, our resulting program also returns all intermediate results on the arguments of interest.

For the function $\overline{cmp}^{\backprime}$ in (8), collecting all intermediate results that depend only on its second parameter yields

$$
c\check{m}p(\bar{r}) = \ < \overline{sum}(4th(\bar{r})),\ \overline{prod}(2nd(\bar{r})) > \tag{10}
$$

We can see that computing $c\check{m}p(\bar{r})$ 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.

4 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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{For each function } f(v_1,...,v_n)=e_f, \text{ define } \Sigma_{[e_f]} = \{v_i \mid i \in \Sigma_f\} \text{ and, for each subexpression } e \text{ of } e_f,} \\
\quad \text{if } e \text{ is } c(e_1,...,e_n) \text{ or } p(e_1,...,e_n) & \text{then } \Sigma_{[e_1]} = ... = \Sigma_{[e_n]} = \Sigma_{[e]} & \\
\quad \text{if } e \text{ is } f_1(e_1,...,e_n) & \text{then } \Sigma_{[e_1]} = ... = \Sigma_{[e_n]} = \Sigma_{[e]} \text{ and } \Sigma_{f_1} = \{i \mid FV(e_i) \subseteq \Sigma_{[e]}\} \cap \Sigma_{f_1} & \\
\quad \text{if } e \text{ is } \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 & \text{then } \Sigma_{[e_1]} = \Sigma_{[e_2]} = \Sigma_{[e_3]} = \Sigma_{[e]} & \\
\quad \text{if } e \text{ is } \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 & \text{then } \Sigma_{[e_1]} = \Sigma_{[e]} \text{ and } \Sigma_{[e_2]} = \begin{cases} \Sigma_{[e]} \cup \{v\} & \text{if } FV(e_1) \subseteq \Sigma_{[e]} \\ \Sigma_{[e]} \setminus \{v\} & \text{otherwise} \end{cases} &
\end{array}
$$

Figure 7: Definition of Σ

$$
\begin{array}{lcll}
\mathcal{C}ol[\![v]\!] & = & <> & \\
\mathcal{C}ol[\![g(e_1,...,e_n)]\!] \quad \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)]\!] & = & \mathbf{let}\ v_1 = \mathcal{C}ol[\![e_1]\!]\ \mathbf{in}\ ...\ \mathbf{let}\ v_n = \mathcal{C}ol[\![e_n]\!]\ \mathbf{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 \\ <\breve{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 \le i_1 < ... < i_k \le n & \\
\mathcal{C}ol[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & = & \mathbf{let}\ v_1 = \mathcal{C}ol[\![e_1]\!]\ \mathbf{in} & \text{if } FV(e_1) \subseteq \Sigma_{[e_1]} \\
& & \mathbf{if}\ 1st(v_1)\ \mathbf{then}\ \mathbf{let}\ v_2 = \mathcal{C}ol[\![e_2]\!]\ \mathbf{in} & \\
& & \qquad\qquad rst(v_1) \ @ \ v_2 \ @ \ \breve{\mathcal{P}ad}[\![e_3]\!] & \\
& & \qquad\quad \mathbf{else}\ \mathbf{let}\ v_3 = \mathcal{C}ol[\![e_3]\!]\ \mathbf{in} & \\
& & \qquad\qquad rst(v_1) \ @ \ \breve{\mathcal{P}ad}[\![e_2]\!] \ @ \ v_3 & \\
& = & \mathbf{let}\ v_1 = \mathcal{C}ol[\![e_1]\!]\ \mathbf{in}\ \mathbf{let}\ v_2 = \mathcal{C}ol[\![e_2]\!]\ \mathbf{in} & \text{otherwise} \\
& & \qquad\qquad \mathbf{let}\ v_3 = \mathcal{C}ol[\![e_3]\!]\ \mathbf{in} & \\
& & \qquad\qquad v_1 \ @ \ v_2 \ @ \ v_3 & \\
\mathcal{C}ol[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!] & = & \mathbf{let}\ v_1 = \mathcal{C}ol[\![e_1]\!]\ \mathbf{in} & \text{if } FV(e_1) \subseteq \Sigma_{[e_1]} \\
& & \mathbf{let}\ v = 1st(v_1)\ \mathbf{in}\ \mathbf{let}\ v_2 = \mathcal{C}ol[\![e_2]\!]\ \mathbf{in} & \\
& & \qquad\qquad rst(v_1) \ @ \ v_2 & \\
& = & \mathbf{let}\ v_1 = \mathcal{C}ol[\![e_1]\!]\ \mathbf{in}\ \mathbf{let}\ v_2 = \mathcal{C}ol[\![e_2]\!]\ \mathbf{in} & \text{otherwise} \\
& & \qquad\qquad v_1 \ @ \ v_2 &
\end{array}
$$

Figure 8: Definition of $\mathcal{C}ol$

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 $\breve{f}_0(x, \bar{r})$ 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 $\breve{f}_0$ with $\bar{f}_0$ to form a function $\breve{\bar{f}}_0$ 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 $\breve{\bar{f}}_0$. Step 2 incrementalizes $\breve{\bar{f}}_0$ under $\oplus$ to obtain an incremental version $\breve{\bar{f}}_0'$. Step 3 prunes out of $\breve{\bar{f}}_0$ and $\breve{\bar{f}}_0'$ the intermediate results and auxiliary information that are not useful.

4.1 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 $\breve{\bar{f}}_0$ to be the pair of $\bar{f}_0$ and $\breve{f}_0$:

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

and use the projection $\Pi_0(\breve{\bar{r}}) = 1st(1st(\breve{\bar{r}}))$ to project out the original return value of f_0. However, we can do better by using a transformation to integrate the computation of $\breve{f}_0$ 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 [13] like those used in tupling tactic [21, 50, 14]. 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 $\check{f}_0(x,\bar{r})$ are embedded in the value of $\bar{\check{f}}_0(x)$. This allows re-arranging the order of the components in the return value.

For the functions $\overline{cmp}$ in (7) and $\check{cmp}$ in (10), we first define a function

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

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

$$\begin{array}{ll}
\multicolumn{2}{l}{1.\ \text{unfold } \bar{\check{cmp}}, \text{ then } \overline{cmp} \text{ and } \check{cmp}} \\
= & \mathbf{let}\ \bar{r} = \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\
& \qquad\quad \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\
& \qquad\quad \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\
& \qquad\quad \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\
& \qquad\quad < 1st(u_1) \le 1st(u_2), \\
& \qquad\qquad v_1, u_1, v_2, u_2 >\ \mathbf{in} \\
& \mathbf{let}\ \check{r} = <\overline{sum}(4th(\bar{r})), \\
& \qquad\qquad \overline{prod}(2nd(\bar{r}))>\ \mathbf{in} \\
& <\bar{r},\check{r}>
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{2.\ \text{lift bindings for } v_1,\ u_1,\ v_2,\ u_2, \text{ and simplify}} \\
= & \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\
& \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\
& \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\
& \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\
& \mathbf{let}\ \bar{r} = <1st(u_1) \le 1st(u_2), \\
& \qquad\qquad v_1, u_1, v_2, u_2 >\ \mathbf{in} \\
& \mathbf{let}\ \check{r} = <\overline{sum}(v_2), \\
& \qquad\qquad \overline{prod}(v_1)>\ \mathbf{in} \\
& <\bar{r},\check{r}>
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{3.\ \text{unfold bindings for } \bar{r} \text{ and } \check{r}} \\
= & \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\
& \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\
& \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\
& \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\
& <<1st(u_1) \le 1st(u_2), \\
& \quad v_1, u_1, v_2, u_2 >, \\
& \ \ <\overline{sum}(v_2),\ \overline{prod}(v_1)>>
\end{array}$$

Simplifying the return value and Π_0, we obtain the function

$$\begin{array}{ll}
\bar{\check{cmp}}(x) = & \mathbf{let}\ v_1 = odd(x)\ \mathbf{in} \\
& \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\
& \mathbf{let}\ v_2 = even(x)\ \mathbf{in} \\
& \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\
& < 1st(u_1) \le 1st(u_2), \\
& \quad v_1,\ u_1,\ v_2,\ u_2,\ \overline{sum}(v_2),\ \overline{prod}(v_1) >
\end{array} \qquad (11)$$

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

4.2 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 [41], as sketched in Section 1. Depending on the power expected from the derivation, the method can be made semi-automatic or fully automatic.

For the function $\bar{\check{cmp}}$ in (11) and input change operation $x \oplus y = cons(y,x)$, we derive an incremental version of $\bar{\check{cmp}}$ under $\oplus$:

$$\begin{array}{ll}
\multicolumn{2}{l}{1.\ \text{unfold } \bar{\check{cmp}}(cons(y,x))} \\
= & \mathbf{let}\ v_1 = odd(cons(y,x))\ \mathbf{in} \\
& \mathbf{let}\ u_1 = \overline{sum}(v_1)\ \mathbf{in} \\
& \mathbf{let}\ v_2 = even(cons(y,x))\ \mathbf{in} \\
& \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\
& < 1st(u_1) \le 1st(u_2), \\
& \quad v_1,\ u_1,\ v_2,\ u_2,\ \overline{sum}(v_2),\ \overline{prod}(v_1) >
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{2.\ \text{unfold } odd,\ \overline{sum},\ even,\ \overline{prod} \text{ and simplify}} \\
= & \mathbf{let}\ v_1' = even(x)\ \mathbf{in} \\
& \mathbf{let}\ u_1' = \overline{sum}(v_1')\ \mathbf{in} \\
& \mathbf{let}\ v_2 = odd(x)\ \mathbf{in} \\
& \mathbf{let}\ u_2 = \overline{prod}(v_2)\ \mathbf{in} \\
& \mathbf{let}\ u_4' = \overline{prod}(v_1')\ \mathbf{in} \\
& <y + 1st(u_1') \le 1st(u_2), \\
& \quad cons(y, v_1'),\ <y{+}1st(u_1'), u_1'>,\ v_2,\ u_2, \\
& \quad \overline{sum}(v_2),\ <y{*}1st(u_4'), u_4'>>
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{3.\ \text{replace all applications by retrievals}} \\
= & \mathbf{let}\ v_1' = 4th(\bar{\check{r}})\ \mathbf{in} \\
& \mathbf{let}\ u_1' = 6th(\bar{\check{r}})\ \mathbf{in} \\
& \mathbf{let}\ v_2 = 2nd(\bar{\check{r}})\ \mathbf{in} \\
& \mathbf{let}\ u_2 = 7th(\bar{\check{r}})\ \mathbf{in} \\
& \mathbf{let}\ u_4' = 5th(\bar{\check{r}})\ \mathbf{in} \\
& <y + 1st(u_1') \le 1st(u_2), \\
& \quad cons(y, v_1'),\ <y{+}1st(u_1'), u_1'>,\ v_2,\ u_2, \\
& \quad 3rd(\bar{\check{r}}),\ <y{*}1st(u_4'), u_4'>>
\end{array}$$

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

$$\begin{array}{ll}
\bar{\check{cmp}}'(y,\bar{\check{r}}) = & <y{+}1st(6th(\bar{\check{r}})) \le 1st(7th(\bar{\check{r}})), \\
& \quad cons(y, 4th(\bar{\check{r}})),\ <y{+}1st(6th(\bar{\check{r}})), 6th(\bar{\check{r}})>, \\
& \quad 2nd(\bar{\check{r}}),\ 7th(\bar{\check{r}}), \\
& \quad 3rd(\bar{\check{r}}),\ <y{*}1st(5th(\bar{\check{r}})), 5th(\bar{\check{r}})>>
\end{array} \qquad (12)$$

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

4.3 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 [40]. A backward dependence analysis determines the components of $\bar{\check{r}}$ and subcomputations of $\bar{\check{f}}_0{}'$ whose values are useful in computing $\Pi_0(\bar{\check{f}}_0{}'(x,y,\bar{\check{r}}))$, 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 (11) and $\bar{\check{cmp}}'$ in (12), we obtain

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

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

Optimizing these functions yields the final definitions of $\widetilde{cmp}$ and $\widetilde{cmp}'$, which appear in Figure 2:

5 Discussion

Correctness. Auxiliary information is maintained incrementally, so at the step of discovering it, we should not be concerned with 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 $\bar{r} = \bar{f}_0(x)$. This 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. Standard constructions for mechanical time analysis [57, 62] can be used, although further study is needed. Automatic space analysis and the trade-off between time and space are problems open for study.

Suppose Step B.1 projects out the original value using $1st$. With the above condition, in a similar way to [40], we can show that, if $f_0(x) = r$, then

$$1st(\widetilde{f}_0(x)) = r \quad \text{and} \quad t(\widetilde{f}_0(x)) \le t(f_0(x)) \tag{13}$$

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

$$
\begin{aligned}
1st(\widetilde{f}_0'(x, y, \widetilde{r})) = r', \quad \widetilde{f}_0'(x, y, \widetilde{r}) = \widetilde{f}_0(x \oplus y), \\
\text{and} \quad t(\widetilde{f}_0'(x, y, \widetilde{r})) \le t(f_0(x \oplus y)).
\end{aligned} \tag{14}
$$

i.e., the functions $\widetilde{f}_0$ and $\widetilde{f}_0'$ preserve the semantics and compute asymptotically at least as fast. Note that $\widetilde{f}_0(x)$ may terminate more often than $f_0(x)$, and $\widetilde{f}_0'(x, y, \widetilde{r})$ may terminate more often than $f_0(x \oplus y)$, due to the transformations used in Steps B.2 and B.3.

Multi-pass discovery of auxiliary information. The function $\widetilde{f}_0$ can sometimes be computed even faster by maintaining auxiliary information useful for incremental computation of the auxiliary information already in $\widetilde{f}_0$. We can obtain such auxiliary information of auxiliary information by iterating the above approach.

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, classes of special parameterized data structures are often used. Ideally, we can collect them as auxiliary information parameterized with certain classes of 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 an extended program $\widetilde{f}_0$, and then use our systematic approach to derive an incremental version of $\widetilde{f}_0$ that incrementally computes the new output using the auxiliary information and also maintains the auxiliary information.

6 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.

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

6.1 Strength reduction in optimizing compilers: binary integer square root

This example is from [45], 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 [16]. 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 [24].

The initial specification of the l-bit non-restoring binary integer square root algorithm [23, 45], which is exact for perfect squares and off by at most 1 for other integers, is

$$
\begin{array}{l}
m := 2^{l-1} \\
\textbf{for } i := l-2 \textbf{ downto } 0 \textbf{ do} \\
\quad p := n - m^2; \\
\quad \textbf{if } p > 0 \textbf{ then} \\
\quad\quad m := m + 2^i \\
\quad \textbf{else if } p < 0 \textbf{ then} \\
\quad\quad m := m - 2^i
\end{array} \tag{15}
$$

In hardware, multiplications and exponentials 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 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 let input change operation $\oplus$ be

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

Step A.1. We cache all intermediate results of f_0, obtaining

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

Step A.2. We transform $\bar{f}_0$ under $\oplus$, obtaining

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

Step A.3. We collect candidate auxiliary information, obtaining

$$\check{f}_0(n, m, i, \bar{r}) = \; < 2{*}m{*}snd(1st(\bar{r})),\ (snd(1st(\bar{r})))^2 > \tag{16}$$

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

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

Step B.2. We derive an incremental version of $\bar{\check{f}}_0$ under $\oplus$, obtaining

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

Step B.3. We prune the functions $\bar{\check{f}}_0$ and $\bar{\check{f}}_0{}'$, eliminating their second components and obtaining

$$\begin{array}{l} \widetilde{f}_0(n,m,i) \\ = \ \textbf{let } u = 2^i \textbf{ in } <pair(n-m^2,u),\ 2*m*u,\ u^2> \end{array} \tag{17}$$

$$\begin{array}{l} \widetilde{f}_0{}'(n,m,i,\widetilde{r}) \\ = \ <pair(fst(1st(\widetilde{r})) \mp 2nd(\widetilde{r}) - 3rd(\widetilde{r}),\ snd(1st(\widetilde{r}))/2), \\ \qquad 2nd(\widetilde{r})/2 \pm 3rd(\widetilde{r}),\ \ 3rd(\widetilde{r})/4> \end{array} \tag{18}$$

The expensive multiplications and exponentials have been completely replaced with additions and shifts. We even discover that an unnecessary shift is done in [45]. Thus, a systematic approach such as ours is desirable not only for automating designs and guaranteeing correctness, but also for reducing costs.

6.2 Transformational programming: path sequence problem

This example is 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 of the example, where an exponential-time recursive solution is improved (incorrectly in [7], correctly in [8]).

The function llp defined below computes the desired length. The input string is given explicitly as the argument to llp. The input graph is represented by a predicate arc such that $\mathrm{arc}(a,b)$ is true iff there is an edge from vertex a to vertex b in the graph. The primitive function max returns the maximum of its two arguments.

$$\begin{array}{rl} llp(l) = & \textbf{if } null(l) \textbf{ then } 0 \\ & \textbf{else } \max(llp(cdr(l)), 1{+}f(car(l),cdr(l))) \\ f(n,l) = & \textbf{if } null(l) \textbf{ then } 0 \\ & \textbf{else if } \mathrm{arc}(n,car(l)) \textbf{ then} \\ & \quad \max(f(n,cdr(l)), 1{+}f(car(l),cdr(l))) \\ & \textbf{else } f(n,cdr(l)) \end{array} \tag{19}$$

The problem is to compute llp incrementally under the input change operation $l \oplus i = cons(i,l)$. Using the approach described in this paper, we obtain

$$\begin{array}{rl} \widetilde{llp}(l) = & \textbf{if } null(l) \textbf{ then } <0> \\ & \textbf{else let } v = \widetilde{f}(car(l),cdr(l)) \textbf{ in} \\ & \quad <\max(llp(cdr(l)), 1{+}1st(v)),\ v> \\ \widetilde{f}(n,l) = & \textbf{if } null(l) \textbf{ then } <0> \\ & \textbf{else let } u = \widetilde{f}(car(l),cdr(l)) \textbf{ in} \\ & \quad \textbf{if } \mathrm{arc}(n,car(l)) \textbf{ then} \\ & \quad\quad <\max(f(n,cdr(l)), 1{+}1st(u),\ u> \\ & \quad \textbf{else } <f(n,cdr(l)),\ u> \end{array} \tag{20}$$

and

$$\begin{array}{rl} \widetilde{llp}'(i,l,\widetilde{r}) = & \textbf{if } null(l) \textbf{ then } <1,<0>> \\ & \textbf{else let } v = \widetilde{f}'(i,l,2nd(\widetilde{r})) \textbf{ in} \\ & \quad <\max(1st(\widetilde{r}), 1{+}1st(v)),\ v> \\ \widetilde{f}'(i,l,\widetilde{r_1}) = & \textbf{if } null(cdr(l)) \textbf{ then} \\ & \quad \textbf{if } \mathrm{arc}(i,car(l)) \textbf{ then } <1,<0>> \\ & \quad \textbf{else } <0,<0>> \\ & \textbf{else let } v = \widetilde{f}'(i,cdr(l),2nd(\widetilde{r_1})) \textbf{ in} \\ & \quad \textbf{if } \mathrm{arc}(i,car(l)) \textbf{ then} \\ & \quad\quad <\max(1st(v), 1{+}1st(\widetilde{r_1})),\ \widetilde{r_1}> \\ & \quad \textbf{else } <1st(v),\ \widetilde{r_1}> \end{array} \tag{21}$$

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

Finally, we use these derived functions to compute the original function llp. Note that $llp(l) = 1st(\widetilde{llp}(l))$ and, if $\widetilde{llp}(l) = \widetilde{r}$, then $\widetilde{llp}'(i,l,\widetilde{r}) = \widetilde{llp}(cons(i,l))$. Using the definition of $\widetilde{llp}'$ in (21) in this last equation, we obtain:

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

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

$$\begin{array}{rl} \widetilde{llp}(l) = & \textbf{if } null(l) \textbf{ then } <0> \\ & \textbf{else if } null(cdr(l)) \textbf{ then } <1,<0>> \\ & \textbf{else let } \widetilde{r} = \widetilde{llp}(cdr(l)) \textbf{ in} \\ & \quad \textbf{let } v = \widetilde{f}'(car(l),cdr(l),2nd(\widetilde{r})) \textbf{ in} \\ & \quad <\max(1st(\widetilde{r}), 1{+}1st(v)),\ v> \end{array} \tag{22}$$

where $\widetilde{f}'$ is defined in (21). This new $\widetilde{llp}$ takes only $O(n^2)$ time, since it calls $\widetilde{f}'$ only $O(n)$ times.

7 Related work and conclusion

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

Interactive systems and *reactive systems* often use incremental algorithms to achieve fast response time [4, 5, 9, 19, 27, 33, 53, 54]. Since explicit incremental algorithms are hard to write and appropriate auxiliary information is hard to discover, the general approach in this paper provides a systematic method for developing particular incremental algorithms. For example, for the dynamic incremental attribute evaluation algorithm in [55], 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 [34, 35], where no auxiliary information is needed, the approach can cache intermediate results and maintain them automatically [40].

Strength reduction [2, 15, 60] 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 [46, 47, 48] generalizes 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. For discontinuous set expressions, dynamic expressions need to be discovered and rules for maintaining them derived. How to discover these dynamic expressions remains to be studied, but once discovered, our method can be used to derive rules that maintain them. In general, such rules apply only to very-high-level languages like SETL; our method applies also to lower-level languages like Lisp.

Maintaining and strengthening loop invariants has been advocated by Dijkstra, Gries, and others [18, 25, 26, 56] 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 [26]. 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 [10, 44] are the logical foundations for recursive calls and iterative loops in deductive program synthesis [42] and constructive logics [16]. 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" [43]. In contrast, the approach in this paper is particularly concerned with the efficiency of the derived programs. Moreover, we can see that induction, whether course-of-value induction [36], structural induction [10, 12], or well-founded induction [10, 44], enables derived programs to use results of previous iterations in each iteration, and generalization [10, 44] enables derived programs to use appropriate auxiliary information by strengthening induction hypotheses, just like strengthening loop invariants. The approach in this paper may be used for systematically constructing induction steps [36] and strengthening induction hypotheses.

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 regarded as deriving incremental programs, and accumulation as identifying appropriate intermediate results or auxiliary information. Bird illustrates these strategies with two examples. However, we can discern no systematic steps being followed in [7]. As demonstrated with the path sequence problem, our approach can be regarded as a systematic formulation of the promotion and accumulation strategies. It helps avoid the kind of errors reported and corrected in [8].

Other work on transformational programming for improving program efficiency, including the *extension* technique in [17], the transformation of recursive functional programs in the CIP project [11, 6, 49], and the finite differencing of functional programs in the semi-automatic program development system KIDS [59], can also be further automated with our systematic approach.

In conclusion, incremental computation has widespread applications throughout computing. This paper proposes a systematic approach for discovering a general class of auxiliary information for incremental computation. It is naturally combined with incrementalization and reusing intermediate results to form a comprehensive approach for efficient incremental computation. The modularity of the approach lets us integrate other techniques in our framework and reuse our components for other optimizations.

Although our approach is presented in terms of a first-order functional language with strict semantics, the underlying principles are general and apply to other languages as well. For example, the method has been used to improve imperative programs with arrays for the local neighborhood problems in image processing [39]. A prototype system, CACHET [38], based on our approach is under development.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[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] R. S. Boyer and J. S. Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

[11] M. Broy. Algebraic methods for program construction: The project CIP. In P. 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.

[12] R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[13] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[14] W.-N. Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, Copenhagen, Denmark, June 1993.

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

[16] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[17] N. Dershowitz. *The Evolution of Programs*, volume 5 of *Progress in Computer Science*. Birkhäuser, Boston, 1983.

[18] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[19] 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.

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

[21] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

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

[23] I. Flores. *The Logic of Computer Arithmetic*. Prentice-Hall International Series in Electrical Engineering. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

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

[25] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[26] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1984.

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

[28] 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.

[29] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[30] 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.

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

[32] 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*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140, Dijon, France, May 1985. Springer-Verlag, Berlin.

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

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

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

[36] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1952. Tenth reprint, Wolters-Noordhoff Publishing, Groningen and North-Holland Publishing Company, Amsterdam, 1991.

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

[38] Y. 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*, Boston, Massachusetts, November 1995. IEEE Computer Society Press.

[39] Y. A. Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1996. To appear as Cornell Technical Report, October, 1995.

[40] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, pages 190–201, La Jolla, California, June 1995.

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

[42] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[43] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, August 1992.

[44] Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming.* Addison-Wesley, Reading, Massachusetts, 1993.

[45] J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In R. Kumar and T. 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.

[46] 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.

[47] 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.

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

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

[50] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Proceedings of the ACM '84 Symposium on LFP*, Austin, Texas, August 1984.

[51] 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.

[52] 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.

[53] S. 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).

[54] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1988.

[55] 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.

[56] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall, 1981.

[57] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the 4th International Conference on FPCA*, pages 144–156, London, U.K., September 1989.

[58] 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.

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

[60] 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*, volume 494 of *Lecture Notes in Computer Science*, pages 394–415, Brighton, U.K., 1991. Springer-Verlag, Berlin.

[61] 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.

[62] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–538, September 1975.

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

[64] 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.