Strengthening Invariants for Efficient Computation ⋆

Yanhong A. Liu [a,1], Scott D. Stoller [a], Tim Teitelbaum [b]

[a] *Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794, USA*

[b] *Department of Computer Science, Cornell University, Ithaca, NY 14853, USA*

Abstract

This paper presents program analyses and transformations for strengthening invariants for the purpose of efficient computation. Finding the stronger invariants corresponds to discovering a general class of auxiliary information for any incremental computation problem. Combining the 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

Efficient computation via 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

⋆ This work was supported in part by ONR under grants N00014-92-J-1973, N00014-99-1-0132, and N00014-99-1-0358 and by NSF under grants CCR-9503319, CCR-9711253, and CCR-9876058. This article is a revised and extended version of a paper that appeared in *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[1] Corresponding author. E-mail: liu@cs.sunysb.edu

computation relies on effective use of state, i.e., computing the result of each iteration incrementally using stored results of previous iterations. This is why strength reduction [3], finite differencing [64], and related techniques 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 $\oplus$. Often, 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 program 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 [2,3,17,23,78], transformational program development [8,20,62,65,77], and interactive systems [5,6,10,22,33,41,71,72]. Numerous techniques for incremental computation have been developed, e.g., [3,4,25,34–36,56,64,68,70,73,76,79,86].

Strengthening invariants for incrementalization. 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 call this approach *incrementalization*. We have partitioned the core of the problem 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., values not computed by $f(x)$.

Our current approaches to problems P1 and P2 are described in [56] and [54], respectively. In this paper, we address problem P3 and contribute:

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

Since auxiliary information is not computed by the original body of computation $f(x)$, adding it strengthens the invariants that hold over the iterative computation that uses $f(x)$.

Some approaches to efficient computation have exploited specific kinds of auxiliary information for strengthening invariants, e.g., auxiliary arithmetic associated with some classical strength-reduction rules [3], auxiliary maps maintained by finite differencing rules for aggregate primitives in SETL [64] and INC [86], and auxiliary data structures for problems with certain properties like stable decomposition [70]. However, 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 results 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. P2 is included here so that candidate auxiliary information useful for efficiently maintaining intermediate results is also included.

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, and 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 incrementalization and therefore also for achieving 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 [56], 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 [54], 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, graph algorithms, and other application areas.

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. A number of 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, call-by-value functional programming language. The expressions of the language are given by the following grammar:

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

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

$$f(v_1, ..., v_n) \triangleq e \tag{1}$$

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.

---

cmp compares sum of odd and product of even positions of list x.

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

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

Fig. 1. Example function definitions.

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 function *cmp* in 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 suffices 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 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 [56] 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. 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 [56]. For example, for function *sum* of Figure 1 and input change operation $x \oplus y = cons(y, x)$, function *sum'* in Figure 2 is derived.

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 [54] 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 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 functions $\widehat{cmp}$ and $\widehat{cmp}'$ in Figure 2 are obtained.

Sometimes, auxiliary information other than the intermediate results of $f_0(x)$ is needed to compute $f_0(x \oplus y)$ quickly. For example, for function cmp of Figure 1 and input change operation $x \oplus y = cons(y, x)$, the values of $sum(even(x))$ and $prod(odd(x))$, in addition to the intermediate values $sum(odd(x))$ and $prod(even(x))$, 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 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. Function $\widetilde{cmp}'$ computes incrementally using only $O(1)$ time. We use this example as a running example.

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 $O(n)$.

$$sum'(y, r) \triangleq\ y + r$$

$cmp(x) = 1st(\widehat{cmp}(x))$.
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 $O(n)$.

$$\begin{array}{l} \widehat{cmp}(x) \triangleq\ \textbf{let } v_1 = sum(odd(x)) \textbf{ in} \\ \qquad\qquad \textbf{let } v_2 = prod(even(x)) \textbf{ in} \\ \qquad\qquad < v_1 \le v_2,\ v_1,\ v_2 > \\ \widehat{cmp}'(y_1, y_2, \hat{r}) \triangleq\ \textbf{let } v_1 = y_1 + 2nd(\hat{r}) \textbf{ in} \\ \qquad\qquad\qquad \textbf{let } v_2 = y_2 * 3rd(\hat{r}) \textbf{ in} \\ \qquad\qquad\qquad < v_1 \le v_2,\ v_1,\ v_2 > \end{array}$$

$cmp(x) = 1st(\widetilde{cmp}(x))$.
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 $O(n)$.

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

Fig. 2. Resulting function definitions.

Notation. We use $<>$ to construct tuples that bundle intermediate results and auxiliary information with the original return value of a function. We use selectors $1st$, $2nd$, $3rd$, ... to select the first, second, third, ... elements 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 $\breve{f}_0$ return candidate auxiliary information for f_0 under $\oplus$. We use $\breve{\bar{f}}_0$ to denote a function that returns all intermediate results and candidate auxiliary information; $\breve{\bar{r}}$, the cached result of $\breve{\bar{f}}_0(x)$; and $\breve{\bar{f}}_0'$, an incremental version of $\breve{\bar{f}}_0$ under $\oplus$. Finally, we use $\widetilde{f_0}$ to denote a function that returns only the useful intermediate results and auxiliary information; $\tilde{r}$, the cached result of $\widetilde{f_0}(x)$; and $\widetilde{f_0}'$, a function that incrementally maintains only the useful intermediate results and auxiliary information. Note that (useful) intermediate results include the original return value. Table 1 summarizes the notation.

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

Table 1
Notation.

For convenience, we assume that bound variables have distinct names.

3 Phase A: Discovering candidate auxiliary information

Auxiliary information is, by definition, useful information not computed by the original program f_0, so it cannot 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 incremental version $f_0'(x, y, r)$, there are often subcomputations that depend only on x and r, but not on y, and whose values cannot be retrieved from the return value or intermediate results of $f_0(x)$. If the values of these subcomputations were available, then we might make f_0' faster.

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

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

3.1 Step A.1: Caching all intermediate results

Extending f_0 to cache all intermediate results uses the transformations in Stage I of [54]. It first performs a straightforward *extension transformation* to embed all intermediate results in the final return value and then performs administrative simplifications.

Certain improvements, suggested in [55] but not given in [55] or [54], can be made to the extension transformation. In particular, we can 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 retrieved 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 [54]; then, we describe an embedding analysis that leads to the desired improvements to the extension transformation.

Extension transformation. For each function definition $f(v_1, ..., v_n) \triangleq e$, we construct a function definition

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

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 3. 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. 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 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 by static inspection of e. The use of $\mathcal{P}ad$ ensures that each possible intermediate result appears in a fixed position independent of values of the conditions in conditional expressions.

$$
\begin{array}{lll}
\mathcal{E}xt[\![v]\!] & = & < v > \\[4pt]
\mathcal{E}xt[\![g(e_1, ..., e_n)]\!] \quad \text{where } g \text{ is } c \text{ or } p & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\
 & & < g(1st(v_1), ..., 1st(v_n)) > @\ rst(v_1)\ @\ ...\ @\ rst(v_n) \\[4pt]
\mathcal{E}xt[\![f(e_1, ..., e_n)]\!] & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } ... \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\
 & & \textbf{let } v = \bar{f}(1st(v_1), ..., 1st(v_n)) \textbf{ in} \\
 & & < 1st(v) > @\ rst(v_1)\ @\ ...\ @\ rst(v_n)\ @ < v > \\[4pt]
\mathcal{E}xt[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in} \\
 & & \textbf{if } 1st(v_1) \textbf{ then let } v_2 = \mathcal{E}xt[\![e_2]\!] \textbf{ in} \\
 & & \qquad\qquad < 1st(v_2) > @\ rst(v_1)\ @\ rst(v_2)\ @\ \mathcal{P}ad[\![e_3]\!] \\
 & & \qquad \textbf{else let } v_3 = \mathcal{E}xt[\![e_3]\!] \textbf{ in} \\
 & & \qquad\qquad < 1st(v_3) > @\ rst(v_1)\ @\ \mathcal{P}ad[\![e_2]\!]\ @\ rst(v_3) \\[4pt]
\mathcal{E}xt[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!] & = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in} \\
 & & \textbf{let } v = 1st(v_1) \textbf{ in let } v_2 = \mathcal{E}xt[\![e_2]\!] \textbf{ in} \\
 & & \qquad\qquad < 1st(v_2) > @\ rst(v_1)\ @\ rst(v_2)
\end{array}
$$

Fig. 3. Definition of $\mathcal{E}xt$.

It is easy to show by induction that each rule $\mathcal{E}xt[\![e]\!] = e'$ guarantees that $1st(e') = e$. Thus, $1st(\bar{f}(v_1, ..., v_n)) = f(v_1, ..., v_n)$. Essentially, $\bar{f}$ performs the same computation as f except that $\bar{f}$ builds a tree of intermediate results computed in computing f. The original value returned by any subcomputation is always a leftmost component. Each leftmost component is consumed as in the original computation to produce the original value. Values of function calls are also kept in the other components, as first established by the last $< v >$ in the rule for function applications and propagated by $@rst(v_i)$ in the other rules.

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. For example, for functions *sum* and *prod* in Figure 1, we obtain functions $\overline{sum}$ and $\overline{prod}$, respectively, in (7), where intermediate values of recursive calls to *sum* and *prod*, respectively, are returned as well as the original return values.

The following improvements, not given in [55] or [54], 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 define and compute *embedding predicates* Mf and Me, where "is embedded in" means "can be retrieved from". 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 predicates must satisfy the following safety requirements:

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

For each function definition $f(v_1,...,v_n) \triangleq e_f$, we define $Mf(f,i)=Me(e_f,v_i)$, and we define Me recursively as in Figure 4. 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, $if_{e_2e_3}^{e_1}$ denotes *true* if the value of e_1 can be determined statically or inferred from the value of **if** e_1 **then** e_2 **else** e_3, and *false* otherwise. For example, $if_{e_2e_3}^{e_1}$ 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 the value of e_1 is T, 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 $if_{e_2e_3}^{e_1}$ is *true*.

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

Fig. 4. Definition of Me.

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 $f(v_1,...,v_n) \triangleq e_f$, we associate an embedding tag $Mtag(e)$ with each occurrence of 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 5; the tags of other subexpressions of e_f are defined to be *false*. These tags can be computed directly once the above embedding

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

Fig. 5. Definition of $Mtag$.

predicates 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 $f(v_1,...,v_n) \triangleq e_f$,

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

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.

Embedding analysis is related to the transmission analysis for compile-time garbage collection [39], in that both analyze values that appear in the result of the enclosing computation. While embedding analysis determines *whether* a value appears in the result, transmission analysis determines *what components of* a value appear in certain components of the result. The domain formed with *true* and *false* for embedding analysis is finite, so it allows an efficient linear-time analysis, which is also precise for our usage here. The components in transmission analysis are described as nested patterns that form an infinite domain, causing the analysis to be imprecise or inefficient in general. Embedding analysis is also related to the propagation analysis for update optimization [81], but that analysis only determines whether a value *is the same as* the result of the enclosing computation.

Improved extension transformation. 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 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:

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

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:

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

To summarize, the transformation $\mathcal{E}xt$ remains the same as in Figure 3 except that the rule for a function application $f(e_1, ..., e_n)$ is replaced with the following: if $Mall(f) = true$ and $Mtag(f(e_1, ..., e_n)) = true$, then define $\mathcal{E}xt[\![f(e_1, ..., e_n)]\!]$ as in (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 3. 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 function *cmp* in Figure 1, this improved extension transformation yields the following functions:

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

where $\overline{cmp}$ is extended to return values of *odd* and *even*, as well as recursive calls to *sum* and *prod*. 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 cannot be retrieved from the cached result of $\bar{f}_0(x)$. It uses analyses and transformations similar to those in [56] 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 *cannot* be retrieved from $\bar{r}$. Thus, time considerations in [56] are dropped here but are picked up after Step A.3, as discussed in Section 5.

In particular, in [56], 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 [56], 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 cannot 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 [56], we denote them by $f^{`}$.

For 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 $\overline{cmp}(x) = \bar{r}$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{1. unfold } \overline{cmp}(cons(y,x))} \\
= & \textbf{let } v_1 = odd(cons(y,x)) \textbf{ in} \\
 & \textbf{let } u_1 = \overline{sum}(v_1) \textbf{ in} \\
 & \textbf{let } v_2 = even(cons(y,x)) \textbf{ in} \\
 & \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\
 & < 1st(u_1) \le 1st(u_2),\ v_1,\ u_1,\ v_2,\ u_2 > \\
\multicolumn{2}{l}{\text{2. unfold } odd,\ \overline{sum},\ even \text{ and simplify}} \\
= & \textbf{let } v_1' = even(x) \textbf{ in} \\
 & \textbf{let } u_1' = \overline{sum}(v_1') \textbf{ in} \\
 & \textbf{let } v_2 = odd(x) \textbf{ in} \\
 & \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\
 & < y+1st(u_1') \le 1st(u_2),\ cons(y, v_1'),\ <y+1st(u_1'),\ u_1'>,\ v_2,\ u_2 > \\
\multicolumn{2}{l}{\text{3. replace applications of } even \text{ and } odd \text{ by retrievals}} \\
= & \textbf{let } v_1' = 4th(\bar{r}) \textbf{ in} \\
 & \textbf{let } u_1' = \overline{sum}(v_1') \textbf{ in} \\
 & \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\
 & \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\
 & < y+1st(u_1') \le 1st(u_2),\ cons(y, v_1'),\ <y+1st(u_1'),\ u_1'>,\ v_2,\ u_2 >
\end{array}
$$

Simplification, i.e., unwinding the bindings for v_1' and v_2 here, yields the following function $\overline{cmp}'$ such that, if $\overline{cmp}(x) = \bar{r}$, then $\overline{cmp}'(y, \bar{r}) = \overline{cmp}(cons(y, x))$:

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

3.3 Step A.3: Collecting candidate auxiliary information

This step collects *candidate auxiliary information*, i.e., intermediate results of $\bar{f}_0'(x, y, \bar{r})$ that depend only on x and $\bar{r}$, and yields function $\check{f}_0(x, \bar{r})$. 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 $\bar{f}_0'(x, y, \bar{r})$ that depend only on x and $\bar{r}$. The analysis is in the same spirit as binding-time analysis [38,45] 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 $f(v_1, ..., v_n) \triangleq e_f$, we compute a set $Sf(f)$ 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 occurrence of each subexpression e of e_f, we compute a set $Se(e)$ 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 6, where $FV(e)$ denotes the set of free variables in e and is

For each function $f(v_1, ..., v_n) \triangleq e_f$, define $Se(e_f) = \{v_i \mid i \in Sf(f)\}$ and, for each subexpression e of e_f,

if e is $c(e_1, ..., e_n)$ or $p(e_1, ..., e_n)$	then $Se(e_1) = ... = Se(e_n) = Se(e)$
if e is $f_1(e_1, ..., e_n)$	then $Se(e_1) = ... = Se(e_n) = Se(e)$, $Sf(f_1) = \{i \mid FV(e_i) \subseteq Se(e)\} \cap Sf(f_1)$
if e is **if** e_1 **then** e_2 **else** e_3	then $Se(e_1) = Se(e_2) = Se(e_3) = Se(e)$
if e is **let** $v = e_1$ **in** e_2	then $Se(e_1) = Se(e)$, $Se(e_2) = \begin{cases} Se(e) \cup \{v\} & \text{if } FV(e_1) \subseteq Se(e) \\ Se(e) \setminus \{v\} & \text{otherwise} \end{cases}$

Fig. 6. Definition of Sf and Se.

defined as follows:

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

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

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

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.

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

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

where $Sf(f) = \{i_1, ..., i_k\}$ 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 $Sf(f) = \{1, ..., n\}$ 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., $FV(e) \subseteq Se(e)$. 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 7, where $\mathcal{P}\breve{a}d[\![e]\!]$ 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 $Sf(f) = \emptyset$, or

$Sf(f) = \{1,...,n\}$ 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 the original value of e, and $rst(\mathcal{C}ol[\![e]\!])$ contains values of intermediate function applications that depend only on x and $\bar{r}$; otherwise, $\mathcal{C}ol[\![e]\!]$ contains only values of intermediate function applications that depend only on x and $\bar{r}$.

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

Fig. 7. Definition of $\mathcal{C}ol$ when $FV(e) \not\subseteq Se(e)$.

Although this forward dependence analysis is equivalent to binding time analysis in partial evaluation [37], the application here is different. 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 a sense, what is computed by our resulting program corresponds to what is computed by the first of the two stages from the staging transformation [40]. The resulting program obtained here is similar to the slice obtained from forward slicing [84]. 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 function $\overline{cmp}'$ in (8), collecting all intermediate results that depend only on its second parameter yields

$$\check{cmp}(\bar{r}) \triangleq < \overline{sum}(4th(\bar{r})),\ \overline{prod}(2nd(\bar{r})) > \tag{10}$$

We can see that computing $\check{cmp}(\bar{r})$ is no slower than computing $cmp(x)$. We will see that this guarantees that incremental computation using the program obtained at the end is at least as fast as computing cmp from scratch.

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 basic idea is to merge the candidate auxiliary information with the original computation of $f_0(x)$, derive an incremental version for the merged program, and determine the least information useful for computing the value of $f_0(x \oplus y)$ in that incremental version.

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

Phase B has three steps. Step 1 merges $\check{f}_0$ with $\bar{f}_0$ to form a function $\bar{\check{f}}_0$ 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 $\bar{\check{f}}_0$. Step 2 incrementalizes $\bar{\check{f}}_0$ under $\oplus$ to obtain an incremental version $\bar{\check{f}}_0'$. Step 3 prunes out of $\bar{\check{f}}_0$ and $\bar{\check{f}}_0'$ 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 $\bar{\check{f}}_0$ to be the pair of $\bar{f}_0$ and $\check{f}_0$:

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

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

We do not describe the integration in detail. Basically, it uses traditional transformation techniques [14] like those used in tupling tactic [24,66,15]. 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 components in the return value.

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

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

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

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

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

2. lift bindings for v_1, u_1, v_2, u_2, and simplify

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

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

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

Simplifying the return value and Π_0, we obtain the function

$$
\begin{array}{rl}
\breve{\overline{cmp}}(x) \triangleq & \textbf{let } v_1 = odd(x) \textbf{ in} \\
& \textbf{let } u_1 = \overline{sum}(v_1) \textbf{ in} \\
& \textbf{let } v_2 = even(x) \textbf{ in} \\
& \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\
& < 1st(u_1) \leq 1st(u_2),\, v_1,\, u_1,\, v_2,\, u_2,\, \overline{sum}(v_2),\, \overline{prod}(v_1) >
\end{array}
\tag{11}
$$

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

4.2 Step B.2: Incrementalization

To derive an incremental version $\breve{\bar{f}}_0'$ of $\breve{\bar{f}}_0$ under $\oplus$, we can use the method in [56], as sketched in Section 1. Depending on the power expected from the derivation, the method can be made semi-automatic or fully automatic.

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

$$
\begin{array}{rl}
\multicolumn{2}{l}{\text{1. unfold } \breve{\overline{cmp}}(cons(y,x))} \\
= & \textbf{let } v_1 = odd(cons(y,x)) \textbf{ in} \\
& \textbf{let } u_1 = \overline{sum}(v_1) \textbf{ in} \\
& \textbf{let } v_2 = even(cons(y,x)) \textbf{ in} \\
& \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\
& < 1st(u_1) \leq 1st(u_2),\, v_1,\, u_1,\, v_2,\, u_2,\, \overline{sum}(v_2),\, \overline{prod}(v_1) > \\
\multicolumn{2}{l}{\text{2. unfold } odd,\, \overline{sum},\, even,\, \overline{prod} \text{ and simplify}} \\
= & \textbf{let } v_1' = even(x) \textbf{ in} \\
& \textbf{let } u_1' = \overline{sum}(v_1') \textbf{ in} \\
& \textbf{let } v_2 = odd(x) \textbf{ in} \\
& \textbf{let } u_2 = \overline{prod}(v_2) \textbf{ in} \\
& \textbf{let } u_4' = \overline{prod}(v_1') \textbf{ in} \\
& < y + 1st(u_1') \leq 1st(u_2),\, cons(y, v_1'),\, < y{+}1st(u_1'), u_1' >,\, v_2,\, u_2, \\
& \quad \overline{sum}(v_2),\, < y{*}1st(u_4'), u_4' >> \\
\multicolumn{2}{l}{\text{3. replace all applications by retrievals}} \\
= & \textbf{let } v_1' = 4th(\breve{\bar{r}}) \textbf{ in} \\
& \textbf{let } u_1' = 6th(\breve{\bar{r}}) \textbf{ in} \\
& \textbf{let } v_2 = 2nd(\breve{\bar{r}}) \textbf{ in} \\
& \textbf{let } u_2 = 7th(\breve{\bar{r}}) \textbf{ in} \\
& \textbf{let } u_4' = 5th(\breve{\bar{r}}) \textbf{ in} \\
& < y + 1st(u_1') \leq 1st(u_2),\, cons(y, v_1'),\, < y{+}1st(u_1'), u_1' >,\, v_2,\, u_2, \\
& \quad 3rd(\breve{\bar{r}}),\, < y{*}1st(u_4'), u_4' >>
\end{array}
$$

Simplification, i.e., unwinding all the bindings, yields the following incremental version $\check{\overline{cmp}}'$ such that, if $\check{\overline{cmp}}(x) = \check{\bar{r}}$, then $\check{\overline{cmp}}'(y, \check{\bar{r}}) = \check{\overline{cmp}}(cons(y, x))$:

$$\begin{aligned}\check{\overline{cmp}}'(y, \check{\bar{r}}) \triangleq\ & {<}y{+}1st(6th(\check{\bar{r}})) \leq 1st(7th(\check{\bar{r}})),\\ & cons(y, 4th(\check{\bar{r}})),\ {<}y{+}1st(6th(\check{\bar{r}})), 6th(\check{\bar{r}}){>},\ 2nd(\check{\bar{r}}),\ 7th(\check{\bar{r}}),\\ & 3rd(\check{\bar{r}}),\ {<}y{*}1st(5th(\check{\bar{r}})), 5th(\check{\bar{r}}){>>}\end{aligned} \tag{12}$$

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

4.3 Step B.3: Pruning

To prune $\check{\bar{f}}_0$ and $\check{\bar{f}}_0'$, we use the analyses and transformations in Stage III of [54]. A backward dependence analysis determines the components of $\check{\bar{r}}$ and subcomputations of $\check{\bar{f}}_0'$ whose values are useful in computing $\Pi_0(\check{\bar{f}}_0'(x, y, \check{\bar{r}}))$, 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. Improved methods for pruning are described in [48,52].

For functions $\check{\overline{cmp}}$ in (11) and $\check{\overline{cmp}}'$ in (12), $1st(\check{\overline{cmp}}'(y, \check{\bar{r}}))$ depends on $1st(6th(\check{\bar{r}}))$ and $1st(7th(\check{\bar{r}}))$, which depend on $1st(3rd(\check{\bar{r}}))$ and $1st(5th(\check{\bar{r}}))$, respectively. All other components are not needed. We obtain

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

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

Optimizing these functions, i.e., eliminating unneeded components, adjusting indexing, and simplifying tuple constructions and selections, 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 cannot be confirmed. Standard constructions for mechanical time analysis [75,83,49] can be used, although further study is needed, and it is being carried out for both time analysis [29] and space analysis [80]. The trade-off between time and space is a problem open for study.

Suppose Step B.1 projects out the original value using $1st$. With the above condition, in a similar way to [54], 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 $\widetilde{f_0}(x) = \tilde{r}$ and $f_0(x \oplus y) = r'$, then

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

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

Multi-pass discovery of auxiliary information. The program $\widetilde{f_0}$ 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. Whether to continue iteration using the approach depends on whether the desired performance improvement is achieved. Therefore, again, analysis of performance of the transformed program is important.

Other auxiliary information. There are cases where the auxiliary information discovered using the above approach is not sufficient for efficient incremental computation. For example, we do not yet know how to discover a heap data structure systematically for computing a minimum element in a set under element deletions. In these cases, classes of special parameterized data structures are often used. Ideally, we could collect them as auxiliary information parameterized with certain classes of data types. Then, we could systematically extend a program to compute such auxiliary information and maintain it incrementally. How to do this precisely is a problem open for study. In the worst case, we could code manually discovered auxiliary information to obtain an extended program $\widetilde{f_0}$, and then use our systematic approach to derive an incremental version $\widetilde{f_0}'$ that incrementally computes the new output using the auxiliary information and also maintains the auxiliary information.

Incrementalization for program efficiency improvement. To use our approach for program efficiency improvement, we need to identify expensive computations f and appropriate change operations $\oplus$.

Expensive computations are generally easy to determine, e.g., function calls, especially calls of non-linear functions, in a language that contains functions, operations on sets and maps in a very-high-level language, or multiplications and exponentiations in a hardware description language. Change operations in an iterative program are simply updates to the parameters of f by the loop body, but change operations in a recursive program, especially for non-linear recursive functions, are non-trivial to determine. Nevertheless, Liu and Stoller have studied a general method for identifying change operations [51] for recursive functions. The basic idea is to use a *minimal* input change that is in the *opposite* direction of change compared to arguments of recursive calls. Using the opposite direction of change yields an increment; using a minimal change allows maximum reuse, i.e., maximum incrementality.

Section 6 gives examples where efficient incremental programs are derived using the method in this paper based on identified expensive computations f and change operations $\oplus$.

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 first presents an example for each of these two applications, based on problems in VLSI design and graph algorithms, respectively. Then, we describe four other examples, taken from problems in games, string processing, combinatorial optimization, and image processing. They help show that this systematic method for discovering auxiliary information is powerful, that the underlying principle is general and applies to other language features, and that it has many application areas.

6.1 *Strength reduction in optimizing compilers: binary integer square root*

This example is from formal hardware design [60], where a specification of a non-restoring binary integer square root algorithm is transformed into a VLSI circuit design and implementation. There, a strength-reduced program was manually discovered and then proved correct using Nuprl [18]. Here, we show how our method can automatically derive the strength reductions. This is of particular interest in light of the Pentium chip flaw in 1994 [28].

The initial specification of the algorithm is given in (15). Given a binary integer n of l bits, where $n > 0$ (and l is usually 8, 16, ...), it computes the binary integer square root m of n using the non-restoring method [26,60], which is exact for perfect squares and off by at most 1 for other integers. The method is to first set bit $l-1$ of the answer m to 1 and then adjust bits $l-2$ to 0 in order: increase m by 2^i if m is smaller than $\sqrt{n}$, and decrease m by 2^i if m is larger than $\sqrt{n}$.

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

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

$$f_0(n,m,i) \triangleq \mathit{pair}(n - m^2, 2^i)$$

where *pair* is a constructor with selectors $\mathit{fst}(a,b) = a$ and $\mathit{snd}(a,b) = b$, 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

$$\bar{f_0}(n,m,i) \triangleq \textbf{ let } v = m^2 \textbf{ in } < pair(n-v,\ 2^i),\ v >$$

Step A.2. We transform $\bar{f_0}$ under $\oplus$, obtaining

$$\begin{aligned}\bar{f_0}'(n,m,i,\bar{r}) \triangleq\ & \textbf{let } v = 2nd(\bar{r}) \pm 2{*}m{*}snd(1st(\bar{r})) + (snd(1st(\bar{r})))^2 \textbf{ in} \\ & < pair(n-v,\ snd(1st(\bar{r}))/2),\ v >\end{aligned}$$

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

$$\check{f_0}(n,m,i,\bar{r}) \triangleq < 2{*}m{*}snd(1st(\bar{r})),\ (snd(1st(\bar{r})))^2 > \tag{16}$$

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

$$\begin{aligned}\bar{\check{f_0}}(n,m,i) \triangleq\ & \textbf{let } v = m^2 \textbf{ in let } u = 2^i \textbf{ in} \\ & <pair(n-v,u),\ v,\ 2{*}m{*}u,\ u^2>\end{aligned}$$

Step B.2. We derive an incremental version of $\bar{\check{f_0}}$ under $\oplus$, obtaining

$$\begin{aligned}\bar{\check{f_0}}'(n,m,i,\bar{\check{r}}) \triangleq\ & \textbf{let } v = 2nd(\bar{\check{r}}) \pm 3rd(\bar{\check{r}}) + 4th(\bar{\check{r}}) \textbf{ in} \\ & \textbf{let } u = snd(1st(\bar{\check{r}}))/2 \textbf{ in} \\ & < pair(fst(1st(\bar{\check{r}})) \mp 3rd(\bar{\check{r}}) - 4th(\bar{\check{r}}),\ u), \\ & \quad v,\ 3rd(\bar{\check{r}})/2 \pm 4th(\bar{\check{r}}),\ 4th(\bar{\check{r}})/4 >\end{aligned}$$

Step B.3. We prune functions $\bar{\check{f_0}}$ and $\bar{\check{f_0}}'$, eliminating their second components and obtaining

$$\widetilde{f_0}(n,m,i) \triangleq \textbf{ let } u = 2^i \textbf{ in } < pair(n-m^2,u),\ 2{*}m{*}u,\ u^2 > \tag{17}$$

$$\begin{aligned}\widetilde{f_0}'(n,m,i,\widetilde{r}) \triangleq\ & < pair(fst(1st(\widetilde{r})) \mp 2nd(\widetilde{r}) - 3rd(\widetilde{r}),\ snd(1st(\widetilde{r}))/2), \\ & 2nd(\widetilde{r})/2 \pm 3rd(\widetilde{r}),\ \ 3rd(\widetilde{r})/4 >\end{aligned} \tag{18}$$

Thus, we can initialize using (17) and perform the corresponding incremental update in the loop body using (18). The expensive multiplications and exponentials in the loop body have been completely replaced with additions, subtractions, and shifts. We even discover that an unnecessary shift is done in [60]. Thus, a systematic approach such as ours is desirable not only for automating designs and guaranteeing correctness, but also for reducing costs. Explicit incrementalization also enables further optimizations on the overall loop structure, as discussed in [47].

6.2 Promotion and accumulation in transformational programming: path sequence problem

This example was used by Bird to illustrate important program transformation strategies called promotion and accumulation [8]. 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 [8], corrected in [9]).

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 $arc(a, b)$ 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.

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

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

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

and

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

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 $\widetilde{llp}(cons(i,l)) = \widetilde{llp}'(i,l,\widetilde{r})$. Using the definition of $\widetilde{llp}'$ in (21) in this last equation, we obtain

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

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

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

where $\widetilde{f}'$ is defined in (21). This new $\widetilde{llp}$ recursively considers tails of the input sequence and, for each element at the head of a subsequence, calls $\widetilde{f}'$ to use it to extend recursively computed sequence lengths. It takes only $O(n^2)$ time, since it calls $\widetilde{f}'$ only $O(n)$ times.

6.3 Towers of Hanoi

This problem computes the sequence of steps needed to move a stack of n disks, one at a time, from one peg a, initially with no larger disks on top of smaller ones, to a second peg b via a third peg c, without ever putting a larger disk on top of a smaller one [1,67]. This can be solved using a simple recursion that has one move operation, two recursive calls, and two concatenation operations, but computing the recursive function for n disks takes $O(2^n)$ time, due to repeated recursive calls on the same arguments.

Pettorossi and Proietti use tupling [66] to give, to our knowledge, the most recent derivation of a linear-time program for this problem [67]. The transformation involves unfolding functions 9 times and eventually identifying that 6 function calls can be computed in a tuple using a recursion with step size 2. The resulting program consists of 9 moves and 18 concatenations in a recursive equation and 3 moves and 6 concatenations together in five other non-recursive equations.

Our method identifies a change operation that corresponds to step size 1, uses the method for discovering auxiliary information twice, and yields an incremental function that has 3 moves and 6 concatenations. Using this function directly to form a new recursion, we obtain a linear-time program that is one quarter the size of that in [67].

Function *hanoi* defined below computes the desired length, where *skip* and *move* are constructors, and :: is a primitive function for concatenation.

$$\begin{aligned} hanoi(n,a,b,c) \triangleq\ & \textbf{if } n \le 0 \textbf{ then } skip \\ & \textbf{else } hanoi(n-1,a,c,b) :: move(a,b) :: hanoi(n-1,c,b,a) \end{aligned} \tag{23}$$

The problem can be formulated as computing *hanoi* incrementally under the input change operation $\langle n', a', b', c' \rangle = \langle n, a, b, c \rangle \oplus \langle\rangle = \langle n+1, a, c, b \rangle$, identified using the method in [51] as the minimum input increment operation. After discovering auxiliary information twice using the method in this paper, we obtain

$$\widetilde{hanoi}(n,a,b,c) \triangleq\ < hanoi(n,a,b,c),\ hanoi(n,b,c,a),\ hanoi(n,c,a,b) > \tag{24}$$

and

$$\begin{aligned} \widetilde{hanoi}'(n,a,b,c,\widetilde{r}) \triangleq\ & \textbf{if } n+1 \le 0 \textbf{ then } < skip, skip, skip > \\ & \textbf{else } < 1st(\widetilde{r}) :: move(a,c) :: 2nd(\widetilde{r}), \\ & \qquad 3rd(\widetilde{r}) :: move(c,b) :: 1st(\widetilde{r}), \\ & \qquad 2nd(\widetilde{r}) :: move(b,a) :: 3rd(\widetilde{r}) > \end{aligned} \tag{25}$$

Clearly, computing $\widetilde{hanoi}(n+1,a,c,b)$ from scratch takes exponential time, but computing $\widetilde{hanoi}'(n,a,b,c,\widetilde{r})$ takes only $O(1)$ time. To compute the original *hanoi*, note that $hanoi(n,a,b,c) = 1st(\widetilde{hanoi}(n,a,b,c))$ and, if $\widetilde{hanoi}(n,a,b,c) = \widetilde{r}$, then $\widetilde{hanoi}(n+1,a,c,b) = \widetilde{hanoi}'(n,a,b,c,\widetilde{r})$. Using the definition of $\widetilde{hanoi}'$ in this last equation, we obtain

$$
\begin{aligned}
\widetilde{hanoi}(n+1,a,c,b) = \;& \textbf{if } n+1 \le 0 \textbf{ then } < skip, skip, skip > \\
& \textbf{else let } \widetilde{r} = \widetilde{hanoi}(n,a,b,c) \textbf{ in} \\
& \quad < 1st(\widetilde{r}) :: move(a,c) :: 2nd(\widetilde{r}), \\
& \quad\;\; 3rd(\widetilde{r}) :: move(c,b) :: 1st(\widetilde{r}), \\
& \quad\;\; 2nd(\widetilde{r}) :: move(b,a) :: 3rd(\widetilde{r}) >
\end{aligned}
$$

Replacing $n+1$ with n, and exchanging b and c, we obtain a new definition of $\widetilde{hanoi}$:

$$
\begin{aligned}
\widetilde{hanoi}(n,a,b,c) \triangleq \;& \textbf{if } n \le 0 \textbf{ then } < skip, skip, skip > \\
& \textbf{else let } \widetilde{r} = \widetilde{hanoi}(n-1,a,c,b) \textbf{ in} \\
& \quad < 1st(\widetilde{r}) :: move(a,b) :: 2nd(\widetilde{r}), \\
& \quad\;\; 3rd(\widetilde{r}) :: move(b,c) :: 1st(\widetilde{r}), \\
& \quad\;\; 2nd(\widetilde{r}) :: move(c,a) :: 3rd(\widetilde{r}) >
\end{aligned}
\tag{26}
$$

Clearly, we can compute *hanoi* using the new $\widetilde{hanoi}$, which takes $O(n)$ time.

6.4 Longest common subsequence

Suppose we want to compute the length of the longest common subsequence of two given sequences x and y of lengths n and m, respectively [19]. This can be programmed as a straightforward recursive function that takes $O(2^{n+m})$ time to compute. Using the method in this paper, in a similar fashion to the path sequence problem discussed earlier, we can derive an optimized program that runs in $O(n*m)$ time.

Function *lcs* below computes the desired length, where $x(n)$ is the nth element of sequence x, and $y(m)$ is the mth element of sequence y.

$$
\begin{aligned}
lcs(n,m) \triangleq \;& \textbf{if } n \le 0 \textit{ or } m \le 0 \textbf{ then } 0 \\
& \textbf{else if } x(n) = y(m) \textbf{ then } lcs(n-1,m-1)+1 \\
& \textbf{else } max(lcs(n,m-1), lcs(n-1,m))
\end{aligned}
\tag{27}
$$

We incrementalize *lcs* under the input change operation $\langle n', m' \rangle = \langle n, m \rangle \oplus \langle\rangle = \langle n+1, m \rangle$, identified using the method in [51] as a minimum input increment operation, and obtain

$$
\begin{array}{rl}
\widetilde{lcs}(n,m) \triangleq & \textbf{if } n \le 0 \textit{ or } m \le 0 \textbf{ then } <0> \\
& \textbf{else let } v = \widetilde{lcs}(n, m-1) \textbf{ in} \\
& \quad \textbf{if } x(n) = y(m) \textbf{ then } < lcs(n-1, m-1)+1, v > \\
& \quad \textbf{else } < max(1st(v), lcs(n-1, m)), v >
\end{array}
\tag{28}
$$

and

$$
\begin{array}{rl}
\widetilde{lcs}'(n,m,\widetilde{r}) \triangleq & \textbf{if } n+1 \le 0 \textit{ or } m \le 0 \textbf{ then } <0> \\
& \textbf{else if } n \le 0 \textbf{ then} \\
& \quad \textbf{let } v = \widetilde{lcs}'(n+1, m-1, <0>) \textbf{ in} \\
& \quad \textbf{if } x(n+1) = y(m) \textbf{ then } <1, v> \\
& \quad \textbf{else } < 1st(v), v > \\
& \textbf{else let } v = \widetilde{lcs}'(n+1, m-1, 2nd(\widetilde{r})) \textbf{ in} \\
& \quad \textbf{if } x(n+1) = y(m) \textbf{ then } < 1st(2nd(\widetilde{r}))+1, v > \\
& \quad \textbf{else } < max(1st(v), 1st(\widetilde{r})), v >
\end{array}
\tag{29}
$$

Computing $\widetilde{lcs}(n+1,m)$ from scratch takes exponential time, but computing $\widetilde{lcs}'(n,m,\widetilde{r})$ takes only $O(m)$ time. Note that $lcs(n,m) = 1st(\widetilde{lcs}(n,m))$ and, if $\widetilde{lcs}(n,m) = \widetilde{r}$, then $\widetilde{lcs}(n+1,m) = \widetilde{lcs}'(n,m,\widetilde{r})$. Using $\widetilde{lcs}'$ to form a new $\widetilde{lcs}$, we obtain

$$
\begin{array}{rl}
lcs(n,m) \triangleq & 1st(\widetilde{lcs}(n,m)) \\
\widetilde{lcs}(n,m) \triangleq & \textbf{if } n \le 0 \textit{ or } m \le 0 \textbf{ then } <0> \\
& \textbf{else let } \widetilde{r} = \widetilde{lcs}(n-1,m) \textbf{ in } \widetilde{lcs}'(n-1, m, \widetilde{r})
\end{array}
\tag{30}
$$

Function $\widetilde{lcs}$ takes $O(n*m)$ time since it calls $\widetilde{lcs}'$ $O(n)$ times.

The auxiliary information used in this problem and the path sequence problem belongs to a special class that can be discovered using a simpler method than the general method described in this paper. Basically, the auxiliary information needed corresponds to "intermediate results" that are computed in some branches but not some other branches of the original program. Computing such results also in these other branches forms the auxiliary information. Thus, the simpler method only needs to perform (I) a modified caching transformation, extending Step A.1, (II) incrementalization, as in Step B.2, and (III) pruning, as in Step B.3.

For example, for function f in the original program (19) for the path sequence problem, the result of $f(car(l), cdr(l))$ is computed in the branch where $arc(n, car(l))$ is true but not in the alternative branch, and the truth value of the predicate arc is independent of the local variables n and l. In function $\widetilde{f}$ in the resulting extended program (20), this result is computed and put in variable u regardless of the truth value of $arc(n, car(l))$, i.e., it is computed when $arc(n, car(l))$ is false as well, as the auxiliary information; this information is used and maintained in the resulting incremental program (21). The details of this simpler but restricted method and the derivation for the longest common subsequence problem are described in [51].

6.5 0-1 Knapsack

This problem computes the maximum value for a subset of n items of integer weight whose total weight does not exceed w [19]. It can be programmed as the following recursive function $knap(n, w)$, where $value(n)$ and $weight(n)$ are the value and weight, respectively, of the nth item. The running time of $knap(n, w)$ is $O(2^n)$.

$$\begin{array}{ll} knap(n,w) \triangleq & \textbf{if } n \le 0 \textit{ or } w \le 0 \textbf{ then } 0 \\ & \textbf{else if } weight(n) \ge w \textbf{ then } knap(n-1, w) \\ & \textbf{else } max(value(n) + knap(n-1, w - weight(n)),\ knap(n-1, w)) \end{array} \tag{31}$$

Using the method in this paper, we can find a piece of auxiliary information, $knap(n, w - weight(n+1))$, that is needed for computing $knap(n+1, w)$, where $weight(n+1)$ can be any integer from 0 to w. Since $w - weight(n+1)$ can also be any integer from 0 to w, taking $knap(n, k)$ for any particular value k as auxiliary information is not sufficient for efficient computation of $knap(n+1, w)$. We need $knap(n, k)$ for all k such that $0 \le k \le w$. We can extend the method in this paper to cache an array for these values, and use the rest of the method in the same way; this yields an optimized program that runs in $O(n * w)$ time. The precise formulation of this extension needs to be studied, but if we assume that the construct **for** $k := i$ **to** j **do** $a[k] := e$ assigns expression e to the kth element of array a for each k and returns a, we can incrementalize $knap$ under $\langle n', w' \rangle = \langle n, w \rangle \oplus \langle \rangle = \langle n+1, w \rangle$ and obtain

$$\begin{array}{ll} \widetilde{knap}(n,w) \triangleq & \textbf{for } k := 0 \textbf{ to } w \textbf{ do} \\ & \quad \widetilde{r}[k] := knap(n,k) \end{array} \tag{32}$$

and

$$\begin{array}{ll} \widetilde{knap}'(n,w,\widetilde{r}) \triangleq & \textbf{for } k := 0 \textbf{ to } w \textbf{ do} \\ & \quad \widetilde{r}'[k] := \begin{array}[t]{l} \textbf{if } n+1 \le 0 \textit{ or } w \le 0 \textbf{ then } 0 \\ \textbf{else if } weight(n+1) \ge w \textbf{ then } \widetilde{r}[w] \\ \textbf{else } max(value(n+1) + \widetilde{r}[w - weight(n+1)], \widetilde{r}[w]) \end{array} \end{array} \tag{33}$$

Computing $\widetilde{knap}(n+1, w)$ from scratch takes exponential time, but computing $\widetilde{knap}'(n, w, \widetilde{r})$ takes only $O(w)$ time. Note that $knap(n, w) = \widetilde{knap}(n, w)[w]$ and, if $\widetilde{knap}(n, w) = \widetilde{r}$, then $\widetilde{knap}(n+1, w) = \widetilde{knap}'(n, w, \widetilde{r})$. Using $\widetilde{knap}'$ to form a new $\widetilde{knap}$, we obtain

$$
\begin{aligned}
knap(n, w) &\triangleq \widetilde{knap}(n, w)[w] \\
\widetilde{knap}(n, w) &\triangleq \textbf{if } n \le 0 \textit{ or } w \le 0 \textbf{ then for } k := 0 \textbf{ to } w \textbf{ do } \widetilde{r}[k] := 0 \\
&\qquad \textbf{else let } \widetilde{r} = \widetilde{knap}(n-1, w) \textbf{ in } \widetilde{knap}'(n-1, w, \widetilde{r})
\end{aligned}
\tag{34}
$$

The auxiliary information used here is essentially a map and is used extensively in finite differencing of set expressions [64]. It is interesting to notice that most previous methods have used arrays as the only data structures for caching, e.g., [16,19,65], and often ended up using asymptotically more space than necessary. We use arrays only when necessary, and we found that, for most problems, arrays are not needed [51].

6.6 Local neighborhood problems

Many image processing problems compute information about local neighborhoods of objects, such as pixels, rows, and regions [82,85,87,88]. For example, an algorithm to blur a picture computes the sum of a square area for every pixel. Such programs are often easily written using loops and arrays, rather than recursive functions.

To optimize these programs using incrementalization, we identify aggregate computations on arrays performed by loops as function f, identify updates to loop variables and array subscripts as operation $\oplus$, and incrementalize f under $\oplus$ [50]; incrementalization is reduced to solving constraints on loop variables and array subscripts, for which we use Omega [69].

To increase incrementality, intermediate results and a special kind of auxiliary information for aggregate array computations are also cached and used when needed [50]. The auxiliary information there corresponds to "intermediate results" that are re-associated based on associativity of operations. For example, given an array a with n_1 rows and n_2 columns, the row-neighborhood-summation problem computes, for each row i ($0 \le i \le n_1-m$), the sum of the m-by-n_2 rectangle comprising rows i through $i+m-1$. The following straightforward program computes this in $O(n_1 n_2 m)$ time.

$$
\begin{array}{l}
\textbf{for } i := 0 \textbf{ to } n_1 - m \textbf{ do} \\
\quad s[i] := 0; \\
\quad \textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\
\quad\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad\quad s[i] := s[i] + a[i+k, l]
\end{array}
\tag{35}
$$

Using the method in [50], we cache a special kind of auxiliary information that corresponds to the innermost **for**-l-loop, which computes the sum of row $i+k$ of array a, and incrementalize the **for**-k-loop with respect to $i' = i+1$ of the **for**-i-loop, yielding

$$
\begin{array}{l}
s[0] := 0; \\
\textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\
\quad s_1[k] := 0; \\
\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad s_1[k] := s_1[k] + a[k, l]; \\
\quad s[0] := s[0] + s_1[k]; \\
\textbf{for } i := 1 \textbf{ to } n_1 - m \textbf{ do} \\
\quad s_1[i{+}m{-}1] := 0; \\
\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad s_1[i{+}m{-}1] := s_1[i{+}m{-}1] + a[i{+}m{-}1, l]; \\
\quad s[i] := s[i{-}1] - s_1[i{-}1] + s_1[i{+}m{-}1]
\end{array}
\tag{36}
$$

Using the ideas in this paper, we can discover the above auxiliary information by directly incrementalizing the **for**-k-loop with respect to $i' = i+1$ of the **for**-i-loop without caching them specially in the first place. Observe that computing the **for**-k-loop for $i' = i+1$ requires adding elements in row $i+m+1$ and subtracting elements in row i; this exposes these rows as auxiliary information. This approach is more general, since, even if we exchange the **for**-k-loop and **for**-l-loop in (35) to compute the same thing, this method yields the same additions and subtractions. However, if we use the method in [50], using the special kind of auxiliary information in the innermost **for**-k-loop yields no improvement. We believe that the basic ideas for identifying general auxiliary information in this paper also apply to discovering general auxiliary information for aggregate array computations. Detailed analyses and transformations are being developed.

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 and on strengthening invariants for efficient computation in general.

Interactive systems and *reactive systems* often use incremental algorithms to achieve fast response time [5,6,10,22,33,41,71,72]. 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 [73], 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 [42,43], where no auxiliary information is needed, the approach can cache intermediate results and maintain them automatically [54].

Strength reduction [3,17,78] 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 [61–64] 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, auxiliary expressions need to be discovered and rules for maintaining them derived. Based on careful study of the finite differencing rules [64], we believe that our basic ideas for discovering auxiliary information apply to discovering auxiliary set expressions as well, and once discovered, our method can also be used to derive rules that maintain these expressions. In general, such rules apply only to very-high-level languages like SETL; our method applies also to lower-level languages like Lisp or Java.

Maintaining and strengthening loop invariants has been advocated by Dijkstra, Gries, and others [21,31,32,74] 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 [32]. 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 [11,59] are the logical foundations for recursive calls and iterative loops in deductive program synthesis [57] and constructive logics [18]. 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" [58]. 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 [44], structural induction [11,13], or well-founded induction [11,59], enables derived programs to use results of previous iterations in each iteration, and generalization [11,59] 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 [44] and strengthening induction hypotheses.

The *promotion and accumulation strategies* are proposed by Bird [8,9] 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, no systematic steps were given in [8]. 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 [9].

Other work on transformational programming for improving program efficiency, including the *extension* technique in [20], the transformation of recursive functional programs in the CIP project [7,12,65], and the finite differencing of functional programs in the semi-automatic program development system KIDS [77], can also be further automated with our systematic approach.

In conclusion, incremental computation has widespread applications for improving the efficiency of computations throughout computing. This paper proposes a systematic approach for strengthening invariants by discovering a general class of auxiliary information for incremental computation. It is naturally combined with methods for reusing the previous result and 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 re-use our components for other optimizations.

Although our approach is presented in terms of a first-order functional language, 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 [50].

A prototype system, CACHET, for incrementalization has been under development. It was first developed as a semi-automatic transformation system [46] for deriving incremental programs that exploit return values [56]. Soon after, transformations and analyses for cache and prune [55,54], fully automated, were added. Later on, a subset of the transformations for exploiting return values were made fully automatic [89], and the analysis for pruning was drastically improved [52]. A separate module for optimizing aggregate array computations [50] was added most recently. All these are implemented using the Synthesizer Generator [72], making extensive use of its scripting language, STk [30], a dialect of Scheme. CACHET has been used in teaching several graduate courses on related topics. To better integrate existing functionalities in the system, and to facilitate implementation of improvements to these functionalities, we are currently planning on a redesign and cleanup of the system. This should also allow us to easily add new extensions [51,53].

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass., 1983.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

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

[4] 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. ACM, New York, Jan. 1990.

[5] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst.*, 8(4):547–576, Oct. 1986.

[6] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The *Pan* language-based editing system. *ACM Trans. Soft. Eng. Methodol.*, 1(1):95–127, Jan. 1992.

[7] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng.*, 15(2):165–180, Feb. 1989.

[8] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, Oct. 1984.

[9] R. S. Bird. Addendum: The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 7(3):490–492, July 1985.

[10] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. ACM, New York, Nov. 1988.

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

[12] M. Broy. Algebraic methods for program construction: The project CIP. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 199–222. Springer-Verlag, Berlin, 1984.

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

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

[15] W.-N. Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132. ACM, New York, June 1993.

[16] W.-N. Chin and M. Hagiya. A bounds inference method for vector-based memoization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 176–187. ACM, New York, June 1997.

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

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

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press/McGraw-Hill, 1990.

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

[21] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[22] 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.

[23] J. Earley. High level iterators and a method for automatically designing data structure representation. *J. Comput. Lang.*, 1:321–342, 1976.

[24] M. S. Feather. A system for assisting program transformation. *ACM Trans. Program. Lang. Syst.*, 4(1):1–20, Jan. 1982.

[25] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 307–322. ACM, New York, June 1990.

[26] I. Flores. *The Logic of Computer Arithmetic*. Prentice-Hall, Englewood Cliffs, N.J., 1963.

[27] *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, Sept. 1989.

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

[29] G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. Technical Report TR 535, Computer Science Department, Indiana University, Nov. 1999.

[30] GrammaTech, Inc. *Synthesizer Generator Scripting Language Supplemental Manual, Release 5.0*. GrammaTech, Inc., Ithaca, New York, 1996.

[31] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[32] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Sci. Comput. Program.*, 2:207–214, 1984.

[33] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Trans. Softw. Eng.*, SE-12(12):1117–1127, Dec. 1986.

[34] R. Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 261–272. ACM, New York, June 1992.

[35] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Trans. Program. Lang. Syst.*, 8(4):577–608, Oct. 1986.

[36] F. Jalili and J. H. Gallier. Building friendly parsers. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 196–206. ACM, New York, Jan. 1982.

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

[38] 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. Springer-Verlag, Berlin, May 1985.

[39] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In FPCA 1989 [27], pages 54–74.

[40] U. Jorring and W. L. Scherlis. Compilers and staging transformations. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 86–96. ACM, New York, Jan. 1986.

[41] G. E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Trans. Program. Lang. Syst.*, 11(2):168–193, Apr. 1989.

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

[43] T. Katayama. Translation of attribute grammars into procedures. *ACM Trans. Program. Lang. Syst.*, 6(3):345–369, July 1984.

[44] S. C. Kleene. *Introduction to Metamathematics.* Van Nostrand, New York, 1952. 10th reprint, Wolters-Noordhoff Publishing, Groningen and North-Holland Publishing Company, Amsterdam, 1991.

[45] J. Launchbury. Projections for specialisation. In B. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, Amsterdam, 1988.

[46] 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*, pages 19–26. IEEE CS Press, Los Alamitos, Calif., Nov. 1995.

[47] Y. A. Liu. Principled strength reduction. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 357–381. Chapman & Hall, London, U.K., 1997.

[48] Y. A. Liu. Dependence analysis for recursive data. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 206–215. IEEE CS Press, Los Alamitos, Calif., May 1998.

[49] Y. A. Liu and G. Gómez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 31–40. Springer-Verlag, June 1998.

[50] Y. A. Liu and S. D. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 262–271. IEEE CS Press, Los Alamitos, Calif., May 1998.

[51] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 288–305. Springer-Verlag, Berlin, Mar. 1999.

[52] Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. In *Proceedings of the 6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 211–231. Springer-Verlag, Berlin, Sept. 1999.

[53] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 73–82. ACM, New York, Jan. 2000.

[54] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998.

[55] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–201. ACM, New York, June 1995.

[56] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.

[57] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, Jan. 1980.

[58] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Trans. Softw. Eng.*, 18(8):674–704, Aug. 1992.

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

[60] 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 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. Springer-Verlag, Berlin, 1995.

[61] B. Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 58–71. ACM, New York, Jan. 1977.

[62] R. Paige. Transformational programming—Applications to algorithms and systems. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 73–87. ACM, New York, Jan. 1983.

[63] R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, Jan. 1986.

[64] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.

[65] H. A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development.* Springer-Verlag, Berlin, 1990.

[66] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming.* ACM, New York, Aug. 1984.

[67] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Comput. Surv.*, 28(2):360–414, June 1996.

[68] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Trans. Program. Lang. Syst.*, 14(2):173–200, Apr. 1992.

[69] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8):102–114, Aug. 1992.

[70] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328. ACM, New York, Jan. 1989.

[71] S. P. Reiss. An approach to incremental compilation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 144–156. ACM, New York, June 1984. Published as SIGPLAN Notices, 19(6).

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

[73] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, July 1983.

[74] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

[75] M. Rosendahl. Automatic complexity analysis. In FPCA 1989 [27], pages 144–156.

[76] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Trans. Program. Lang. Syst.*, 10(1):1–50, Jan. 1988.

[77] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, Sept. 1990.

[78] 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. Springer-Verlag, Berlin, 1991.

[79] R. S. Sundaresh and P. Hudak. Incremental computation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13. ACM, New York, Jan. 1991.

[80] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical Report TR 538, Computer Science Department, Indiana University, Apr. 2000.

[81] M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 184–193. IEEE CS Press, Los Alamitos, Calif., May 1998.

[82] J. A. Webb. Steps towards architecture-independent image processing. *IEEE Comput.*, 25(2):21–31, Feb. 1992.

[83] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–538, Sept. 1975.

[84] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, SE-10(4):352–357, July 1984.

[85] W. M. Wells, III. Efficient synthesis of Gaussian filters by cascaded uniform filters. *IEEE Trans. Patt. Anal. Mach. Intell.*, 8(2):234–239, Mar. 1986.

[86] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, Apr. 1991.

[87] R. Zabih. *Individuating Unknown Objects by Combining Motion and Stereo.* PhD thesis, Department of Computer Science, Stanford University, Stanford, Calif., 1994.

[88] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In J.-O. Eklundh, editor, *Proceedings of the 3rd European Conference on Computer Vision*, volume 801 of *Lecture Notes in Computer Science*, pages 151–158. Springer-Verlag, 1994.

[89] Y. Zhang and Y. A. Liu. Automating derivation of incremental programs. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, page 350. ACM, New York, Sept. 1998.