Efficient Computation via Incremental Computation

Yanhong A. Liu*

July 1998

Abstract

Incremental computation takes advantage of repeated computations on inputs that differ slightly from one another, computing each output efficiently by exploiting the previous output. This paper gives an overview of a general and systematic approach to *incrementalization*. Given a program f and an operation $\oplus$, the approach yields an incremental program that computes $f(x \oplus y)$ efficiently by using the result of $f(x)$, the intermediate results of $f(x)$, and auxiliary information about $f(x)$ that can be inexpensively maintained.

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

Keywords: caching, incremental computation, incremental programs, incrementalization, program analysis, program optimization, program transformation, programming environments, reuse

1 Introduction

Incremental programs. Given a program f and an operation $\oplus$, a program f' is called an *incremental version* of f under $\oplus$ if f' computes $f(x \oplus y)$ efficiently by making use of $f(x)$. Below are some examples:

- Suppose f is a program *sort*, x is a list of numbers, and $\oplus$ adds a number y to the old input x, i.e., $x \oplus y$ is $cons(y, x)$. Then f' can be an insertion program $sort'$ that inserts y at the appropriate place in the sorted list $sort(x)$. The incremental version $sort'$ satisfies that, if $r = sort(x)$, then $sort'(x, y, r) = sort(cons(y, x))$. While computing $sort(cons(y, x))$ from scratch must take $\Omega(n \log n)$ time, which is the lower bound for general sorting, doing insertion using $sort'(x, y, r)$ takes only $O(n)$ time.

*This work was supported in part by ONR Grant N00014-92-J-1973, NSF Grant CCR-9503319, and NSF Grant CCR-9711253. Author's address: Computer Science Department, 215 Lindley Hall, Indiana University, Bloomington, IN 47405. Phone: 812-855-4373; Fax -4829. Email: liu@cs.indiana.edu.

- Suppose f is a C compiler, x is a C program, and $\oplus$ performs changes to the C program. Then f' is an incremental C compiler that compiles a new program by updating the old compiled code rather than compiling from scratch.

- For general iterative programs, f is a loop body, x is the induction variable, and $\oplus$ is increment to the induction variable. Then f' is a general strength-reduced version that computes each iteration incrementally based on the result of the previous iteration.

Incrementalization for efficiency improvement. In essence, all nontrivial computation proceeds in certain repetitive fashion: iteration or recursion. For efficiency, each iteration needs to be computed incrementally using the stored results of the previous iteration. So, here comes in incrementalization: we regard the iteration body as a program f, and we regard the iteration increment as an operation $\oplus$; then, we can use an incremental version to replace the iteration body. As a result, we compute each iteration faster, and thus the overall computation faster.

It is easy to see that incrementalization has wide applications: interactive systems such as programming environments and document editing systems, reactive systems, real-time systems, distributed systems, dynamic management of large databases, as well as general program optimizations.

A general systematic transformational approach. This work aims at studying a general and systematic transformational approach to incrementalization. Given a program f and an operation $\oplus$, the approach aims to derive an incremental program that computes $f(x \oplus y)$ efficiently by using

P1: the value of $f(x)$,
P2: the intermediate results of $f(x)$, and
P3: auxiliary information of $f(x)$ that can be inexpensively maintained.

Using the value of $f(x)$ gives us incrementality over computing $f(x \oplus y)$ from scratch; using the intermediate results of $f(x)$ gives us greater incrementality than using only the value of $f(x)$; using auxiliary information gives us even greater incrementality than using only the return value and the intermediate results. We use P1, P2, and P3 to denote these three subproblems.

Related work. There has been a great deal of work on incremental computation [13]. Despite various classifications, we separate all the work into three categories.

The first category consists of *incremental algorithms*, which includes dynamic algorithms and on-line algorithms. They are particular algorithms manually derived to handle particular problems and particular input changes. Examples are incremental parsing, attribute evaluation, data-flow analysis, circuit evaluation, constraint solving, transitive closure, shortest path, minimum spanning tree, connectivity, scheduling, etc. Since these algorithms are manually derived to solve particular incremental problems, we say that they are ad hoc.

The second category is called *incremental execution frameworks.* The idea is to allow different application programs to run in such a framework without deriving explicit incremental algorithms. These are general methods for incremental problems. Examples are incremental attribute evaluation framework [14], function caching [12], lambda reduction [1], traditional partial evaluation [16], change detailing network [17], program abstraction [4], etc. Each such framework gives some language for describing application programs and, in particular, fixes the classes of input changes that the framework can handle, and uses a particular incremental algorithm to handle the input changes. Any input change to an application program is mapped to a change that the framework can handle. Thus, these frameworks are not always effective for particular applications. So we say that these frameworks have poor specializability.

The third category is called *incremental-program derivation approaches*. This class aims to be general, as does the second class, in that it handles any programs and any input changes; it also aims to be effective on each given problem by deriving an incremental program using special properties of the problem. Indeed, many methods for program efficiency improvement in optimizing compilers, transformational programming, and programming methodology do derive efficient incremental programs and use them in computing each iteration of an overall computation. Examples are strength reduction in optimizing compilers [2], finite differencing in transformational programming [11, 15], and maintaining loop invariance in programming methodology [3]. Our work falls into this category. While existing work either handles only limited primitive operators or gives only high-level strategies, our work is more general and systematic. It is general in that it explores the underlying principles of incremental computation, independent of particular language constructs, and it is systematic in that it comprises concrete program analyses and transformations, not just high-level strategies.

Outline. This rest of the paper is organized as follows. We first present an overview of the approach, in particular, solutions to P1: exploiting the previous result, P2: caching intermediate results, and P3: discovering auxiliary information. The presentation uses a small example. More examples are given afterwards. Then, we summarize the approach and describe the prototype system, CACHET.

2 Methods and Techniques

Language and example. We use an utterly small example to illustrate our approach. The example is written in a first-order, call-by-value functional language. Each program is a set of mutually recursive function definitions. The expression that defines a function is built from the most commonly used language constructs: variables, data constructions, primitive function applications, user-defined function applications, conditional expressions, and binding expressions. An example program *cmp* is given in Figure 1. It compare sum of odd and product of even positions of list x. We use the same language to describe the operation $\oplus$. For example, $x \oplus y = cons(y, x)$. Even though the language is simple, it can express all computable functions, and it is sufficiently powerful and convenient to write sophisticated programs f and operations $\oplus$.

We use an asymptotic time cost model. Our primary goal is to reduce the asymptotic running time of the incremental programs. Of course, caching intermediate results and

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

Figure 1: An example program.

auxiliary information takes extra space. Our secondary goal is to save space by maintaining only information useful for the incremental computation.

P1: Exploiting the previous result. Suppose that we have computed $f(x)$ and obtained its result r, as depicted on the left of Figure 2, and that we want to compute $f(x \oplus y)$ on the right. Clearly, all subcomputations of $f(x \oplus y)$ depend on either x or y. For those that depend on y, a new parameter, we do not attempt to reuse the old result r. For those that depend only on x, e.g., $f(x)$ if it is included, then we avoid recomputation by replacing them with retrievals from the old result r. Thus, the idea is to symbolically transform $f(x \oplus y)$ to separate subcomputations on x from those on y and replace those on x by retrievals from r. The resulting program f' may depend on x, y, and r, and it satisfies that if $f(x \oplus y) = r'$ then $f'(x, y, r) = r'$, as illustrated at the bottom of Figure 2. To summarize, we first

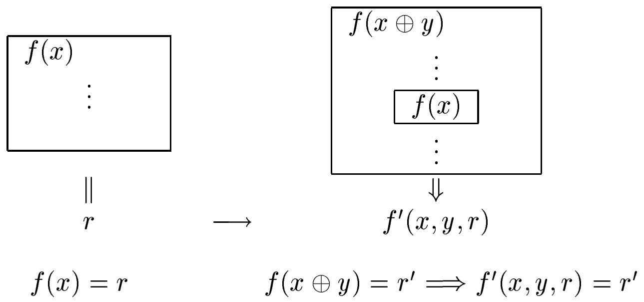


Figure 2: Exploiting the previous result.

introduce function f' with cached result r as an extra argument. Then, we do two things to obtain a definition of the incremental version: (1) unfold (i.e., expand using definition) and simplify, and (2) replace using the cached result (based on identity, for the case that $f(x)$ appears in the expanded $f(x \oplus y)$). Finally, we replace a function call to f with a call to the incremental version f'.

We can do much better than only directly using the value of $f(x)$. We exploit the semantics of each program construct. If the return value r of $f(x)$ is a tuple, and $g(x)$ is a component, then the value of $g(x)$ can be retrieved from r, as depicted on the left of Figure 3. Thus, a subcomputation $g(x)$ on the right can be replaced by a retrieval from r. Subcomputation $g(x)$ on the right may appear in certain context, e.g., in the true branch of a conditional expression. If $f(x)$ can be specialized to $g(x)$ under the same condition, then the value of $g(x)$ on the right can be retrieved from r in this branch, even if it may not be retrievable in the other branch. The transformation steps are as summarized above, except

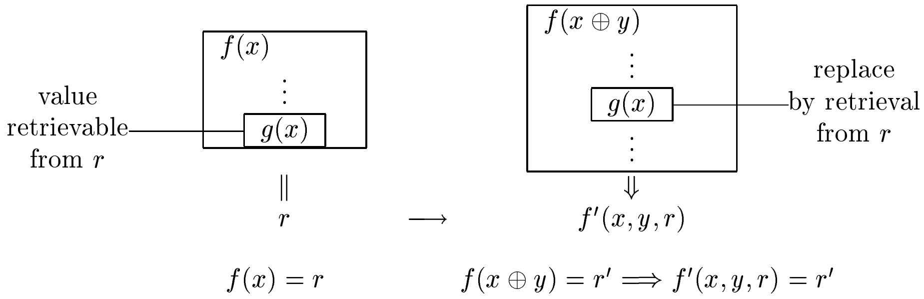


Figure 3: Exploiting the previous result (cont'd).

that replacements using the cached result are based also on equality reasoning and auxiliary specialization.

As for most program transformation techniques, it should be noted that the quality of the resulting program depends on that of the original program. This will be illustrated with our examples of different sorting programs: insertion sort, selection sort, and merge sort.

P1: Example. Consider the given function *sum* and the operation $\oplus$ below.

$$\boxed{\begin{array}{ll} sum(x) = & \textbf{if } null(x) \textbf{ then } 0 \\ & \textbf{else } car(x) + sum(cdr(x)) \\ x \oplus y = & cons(y, x) \end{array}}$$

We introduce $sum'(x, y, r)$, where $r = sum(x)$, to compute $sum(cons(y, x))$. Unfolding $sum(cons(y, x))$ yields

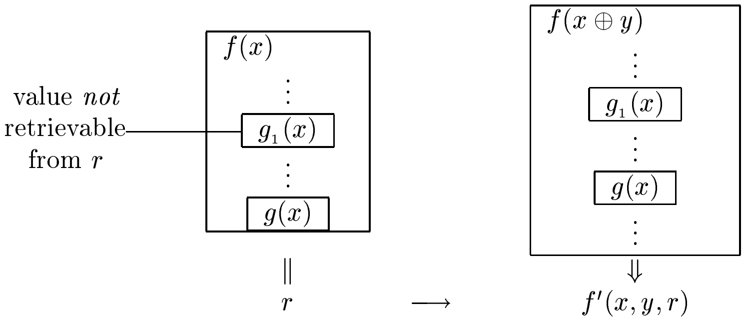
$$\begin{array}{l} \textbf{if } null(cons(y, x)) \textbf{ then } 0 \\ \textbf{else } car(cons(y, x)) + sum(cdr(cons(y, x))) \end{array}$$

where the condition is simplified to false, the first operand of $+$ is simplified to y, and the argument to *sum* is simplified to x. Then replace $sum(x)$ by r. We obtain

$$\boxed{sum'(y, r) = y + r}$$

where parameter x to sum' is dead and eliminated. We have, if $r = sum(x)$, then $sum'(y, r) = sum(cons(y, x))$. While $sum(cons(y, x))$ takes $O(n)$ time to compute, $sum'(y, r)$ takes only $O(1)$ time and needs one unit of space to hold the old result.

P2: Caching intermediate results. Often, intermediate results of $f(x)$ that are not retrievable from the return value are useful for efficient incremental computation. This is illustrated in Figure 4, which is the same as Figure 3 except for the two additional boxes for $g_1(x)$. A subcomputation $g_1(x)$ on the right may also be computed by $f(x)$ on the left, but its value is not retrievable from the result r. If we are given what intermediate results of $f(x)$ are useful for the incremental computation, then we can extend $f(x)$ to $\hat{f}(x)$ that returns also these results in $\hat{r}$, as illustrated at the bottom of Figure 4, and then an incremental version $\hat{f}'$ of $\hat{f}$ under $\oplus$ can use these results in $\hat{r}$ and compute the corresponding results for $\hat{f}(x \oplus y)$. The hard problem is that $f(x)$ may compute a huge number of intermediate



$$f(x) = r \qquad f(x \oplus y) = r' \Longrightarrow f'(x, y, r) = r'$$
$$\hat{f}(x) = \hat{r} \qquad \hat{f}(x \oplus y) = \hat{r}' \Longrightarrow \hat{f}'(x, y, \hat{r}) = \hat{r}'$$

Figure 4: Caching intermediate results.

results. How can we identify useful intermediate results? We propose a three-stage approach called *cache-and-prune*:

I : cache all intermediate results of f and obtain $\bar{f}$;
II : incrementalize $\bar{f}$ under $\oplus$ using P1 and obtain $\bar{f}'$;
III: prune $\bar{f}$, $\bar{f}'$ using dependency in $\bar{f}'$ and obtain $\hat{f}$, $\hat{f}'$.

The approach is modular. Stage I provides all intermediate results possibly used by Stage II. Stage II is reduced to P1. Stage III preserves only intermediate results used by Stage II. Thus, The overall method has a kind of optimality.

The approach can be used for general program optimization via caching, by incrementalizing the body of a loop or recursion. This will be illustrated with the classical Fibonacci function.

P2: Example. Consider the given function *cmp* and the operation $\oplus$ below.

$$\boxed{\begin{aligned} cmp(x) &= \ sum(odd(x)) \leq prod(even(x)) \\ x \oplus \langle y_1, y_2 \rangle &= \ cons(y_1, cons(y_2, x)) \end{aligned}}$$

Clearly, if we cache intermediate results $sum(odd(x))$ and $prod(even(x))$, then an incremental version only needs to add y_1 to the former and multiply y_2 by the latter.

Using cache-and-prune, we obtain functions $\widehat{cmp}$ and $\widehat{cmp}'$, such that if $\hat{r} = \widehat{cmp}(x)$, then

$\widehat{cmp}'(y_1, y_2, \hat{r}) = \widehat{cmp}(cons(y_1, cons(y_2, x)))$, and $cmp(x) = 1st(\widehat{cmp}(x))$.

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

where $<>$ denotes a tuple, and selectors $1st$, $2nd$, and so on select the corresponding components of a tuple. While $cmp(cons(y_1, cons(y_2, x)))$ takes $O(n)$ time to compute, $\widehat{cmp}'(y_1, y_2, \hat{r})$ takes only $O(1)$ time and needs two additional units of space for two intermediate results.

P3: Discovering auxiliary information. Sometimes, auxiliary information not computed by $f(x)$ at all is useful for efficient computation of $f(x \oplus y)$. However, it is difficult to discover such information. People have studied hard to discover various auxiliary information for various manually derived incremental algorithms. We propose a systematic approach that can discover a general class of auxiliary information. The idea is illustrated in Figure 5, which is the same as Figure 4 except for the additional box for $h(x)$ on the right.

First, we transform $f(x \oplus y)$ to separate subcomputations on x from those on y, as done for P1. If the value of a subcomputation, e.g., $g(x)$ or $g_1(x)$, can be retrieved from the return value of $f(x)$ or an intermediate result of $f(x)$, then it is left alone. However, if the value of a subcomputation depending only on x, e.g., $h(x)$, can not be retrieved from either, then it is collected as a candidate auxiliary information. Next, to determine whether such information can be used and maintained for efficient incremental computation, we extend $f(x)$ to compute and cache this information, as well as intermediate results, incrementalize the resulting program, and prune out useless values and computations, as done for P2, and we obtain the resulting programs $\tilde{f}$ and $\tilde{f}'$, as shown at the bottom of Figure 5. To

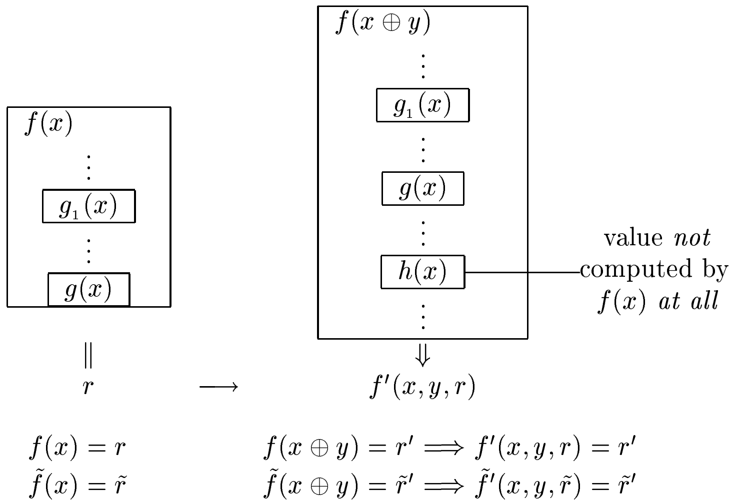


Figure 5: Discovering auxiliary information.

summarize, we use a two-phase approach.

A: discover candidate auxiliary information using P1;
B: use candidate auxiliary information using P2.

Thus, we have reduced a difficult problem to modular steps where solutions to previous problems can be used.

P3: Example. Consider the given function *cmp* and the operation $\oplus$ below.

$$\boxed{\begin{aligned} cmp(x) = &\ sum(odd(x)) \leq prod(even(x)) \\ x \oplus y = &\ cons(y, x) \end{aligned}}$$

After an input change, the sublists for the odd positions and even positions are swapped. Caching only intermediate results is useless for the incremental computation. We need to compute and save also the values of $sum(even(x))$ and $prod(odd(x))$. Then, an incremental version can use and maintain each of these values by a single addition, multiplication, or copy.

Using the two-phase approach, we obtain functions $\widetilde{cmp}$ and $\widetilde{cmp}'$, such that if $\tilde{r} = \widetilde{cmp}(x)$, then $\widetilde{cmp}'(y, \tilde{r}) = \widetilde{cmp}(cons(y, x))$, and $cmp(x) = 1st(\widetilde{cmp}(x))$.

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

While $cmp(cons(y, x))$ takes $O(n)$ time to compute, $\widetilde{cmp}'(y, \tilde{r})$ takes only $O(1)$ time and needs another two additional units of space for two pieces of auxiliary information.

3 Examples

More examples. The table summarizes the running times of both the batch versions and the incremental versions (with respect to the given $\oplus$ operations) for the examples *sum* and *cmp* seen above and three sorting programs to be seen below. For the example of Fibonacci function, an incremental version is used in the body of the recursion to improve the straightforward program to an optimized program.

Problem	Batch	Incremental
sum, *cmp*	$O(n)$	$O(1)$
insertion sort	$O(n^2)$	$O(n)$
selection sort	$O(n^2)$	$O(n)$
merge sort	$O(n \log n)$	$O(n)$

Problem	Straightforward	Optimized
Fibonacci function	$O(2^n)$	$O(n)$

Insertion sort. Insertion sort takes a list, recursively sorts the tail of the list, and then inserts the first element into the appropriate place in the sorted tail. Consider its definition below and a change operation that adds an element y to the input list.

$$
\boxed{\begin{array}{lll}
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else } insert(car(x), sort(cdr(x))) \\
insert(i,x) & = & \textbf{if } null(x) \textbf{ then } cons(i,nil) \\
 & & \textbf{else if } i \le car(x) \textbf{ then } cons(i,x) \\
 & & \textbf{else } cons(car(x), insert(i, cdr(x))) \\
x \oplus y = & cons(y,x) &
\end{array}}
$$

We introduce $sort'(x,y,r)$, where $r = sort(x)$, to compute $sort(cons(y,x))$. Unfolding $sort(cons(y,x))$ yields

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

where the condition is simplified to false, the first argument of $insert$ is simplified to y, and the argument to $sort$ is simplified to x. Then replace $sort(x)$ by r. We obtain

$$
\boxed{sort'(y,r) = insert(y,r)}
$$

where parameter x to $sort$ is dead and eliminated. We have, if $r = sort(x)$, then $sort'(y,r) = sort(cons(y,x))$. While $sort(cons(y,x))$ takes $O(n^2)$ time, $sort'(y,r)$ takes $O(n)$ time. This result is easy to obtain. Function $sort'$ simply calls the given function $insert$.

Selection sort. Selection sort takes a list, selects the least element in the list, puts it in the first place, and then recursively sort the rest of the list. Consider its definition below and a change operation that adds an element y to the input list. It is nontrivial how selection sort of the new list can use the previously sorted list. Our approach to P1 allows us to derive a definition of insertion not given in the original program.

$$
\boxed{\begin{array}{lll}
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else let } k = least(x) \textbf{ in} \\
 & & \quad\; cons(k, sort(rest(x,k))) \\
least(x) & = & \textbf{if } null(cdr(x)) \textbf{ then } car(x) \\
 & & \textbf{else let } s = least(cdr(x)) \textbf{ in} \\
 & & \quad\; \textbf{if } car(x) < s \textbf{ then } car(x) \textbf{ else } s \\
rest(x,k) & = & \textbf{if } k = car(x) \textbf{ then } cdr(x) \\
 & & \textbf{else } cons(car(x), rest(cdr(x),k)) \\
x \oplus y = & cons(y,x) &
\end{array}}
$$

We introduce $sort'(y,x,r)$ to compute $sort(cons(y,x))$, where $r = sort(x)$. First, unfold $sort(cons(y,x))$ and simplify:

$$
\begin{array}{rll}
sort(cons(y,x)) = & \textbf{if } null(cons(y,x)) \textbf{ then } nil & = \textbf{let } k = least(cons(y,x)) \textbf{ in} \\
 & \textbf{else let } k = least(cons(y,x)) \textbf{ in} & \quad cons(k, sort(rest(cons(y,x),k))) \\
 & \quad cons(k, sort(rest(cons(y,x),k))) &
\end{array} \tag{1}
$$

Continue. Unfold $least(cons(y,x))$ in (1) and simplify:

$$\begin{array}{rll} least(cons(y,x)) = & \textbf{if } null(cdr(cons(y,x))) \textbf{ then } car(cons(y,x)) & = \textbf{if } null(x) \textbf{ then } y \\ & \textbf{else let } s = least(cdr(cons(y,x))) \textbf{ in} & \quad \textbf{else let } s = least(x) \textbf{ in} \\ & \quad \textbf{if } car(cons(y,x)) < s \textbf{ then } car(cons(y,x)) \textbf{ else } s & \quad\quad \textbf{if } y < s \textbf{ then } y \textbf{ else } s \end{array} \qquad (2)$$

and unfold (2) into (1) and simplify:

$$\begin{array}{rll} (1) = & \textbf{let } k = \textbf{if } null(x) \textbf{ then } y & = \textbf{if } null(x) \textbf{ then } cons(y, sort(rest(cons(y,x),y))) \\ & \quad\quad \textbf{else let } s = least(x) \textbf{ in} & \quad \textbf{else let } s = least(x) \textbf{ in} \\ & \quad\quad\quad \textbf{if } y < s \textbf{ then } y \textbf{ else } s \textbf{ in} & \quad\quad \textbf{if } y < s \textbf{ then } cons(y, sort(rest(cons(y,x),y))) \\ & cons(k, sort(rest(cons(y,x),k))) & \quad\quad \textbf{else } cons(s, sort(rest(cons(y,x),s))) \end{array} \qquad (3)$$

Continue. Unfold $rest(cons(y,x),y)$ and $rest(cons(y,x),s)$ in (3) and simplify:

$$\begin{array}{rlll} rest(cons(y,x),y) = & \textbf{if } y = car(cons(y,x)) \textbf{ then } cdr(cons(y,x)) & = \textbf{if } y = y \textbf{ then } x & = x \\ & \textbf{else } cons(car(cons(y,x)), rest(cdr(cons(y,x)),y))) & \quad \textbf{else } cons(y, rest(x,y)) & \end{array} \qquad (4)$$

$$\begin{array}{rll} rest(cons(y,x),s) = & \textbf{if } s = car(cons(y,x)) \textbf{ then } cdr(cons(y,x)) & = \textbf{if } s = y \textbf{ then } x \\ & \textbf{else } cons(car(cons(y,x)), rest(cdr(cons(y,x)),s))) & \quad \textbf{else } cons(y, rest(x,s)) \end{array} \qquad (5)$$

and unfold (4) and (5) into (3) and simplify:

$$\begin{array}{rll} (3) = & \textbf{if } null(x) \textbf{ then } cons(y, sort(x)) & = \textbf{if } null(x) \textbf{ then } cons(y, sort(x)) \\ & \textbf{else let } s = least(x) \textbf{ in} & \quad \textbf{else let } s = least(x) \textbf{ in} \\ & \quad \textbf{if } y < s \textbf{ then } cons(y, sort(x)) & \quad\quad \textbf{if } y \le s \textbf{ then } cons(y, sort(x)) \\ & \quad \textbf{else } cons(s, sort(\textbf{if } s = y \textbf{ then } x & \quad\quad \textbf{else } cons(s, sort(cons(y, rest(x,s)))) \\ & \quad\quad\quad\quad \textbf{else } cons(y, rest(x,s)))) & \end{array} \qquad (6)$$

Next, we perform equality reasoning and auxiliary specialization and replace subcomputations in (6) with retrievals from the cached result r. First, $sort(x)$ in (6) is replaced by r. Also, $null(x) = null(r)$ since, using auxiliary specialization twice, we have $null(x)$ is true if and only if $null(sort(x))$ is true. Furthermore, when $null(x)$ is false, $sort(x)$ is specialized to **let** $k = least(x)$ **in** $cons(k, sort(rest(x,k)))$ by definition, which means $least(x) = car(r)$ and $sort(rest(x, least(x))) = cdr(r)$. Thus $least(x)$ in (6) is replaced by $car(r)$. Finally, recursive call $sort(cons(y, rest(x,s)))$ is replaced by $sort'(y, rest(x,s), sort(rest(x,s)))$, where $sort(rest(x,s))$ in the latter is replaced by $cdr(r)$. We obtain

$$\begin{array}{rl} sort'(y,x,r) = & \textbf{if } null(r) \textbf{ then } cons(y,r) \\ & \textbf{else let } s = car(r) \textbf{ in} \\ & \quad \textbf{if } y \le s \textbf{ then } cons(y,r) \\ & \quad \textbf{else } cons(s, sort'(y, rest(x,s), cdr(r))) \end{array}$$

Eliminate dead parameter x and dead code in the corresponding argument, we obtain

$$\boxed{\begin{array}{rl} sort'(y,r) = & \textbf{if } null(r) \textbf{ then } cons(y,r) \\ & \textbf{else let } s = car(r) \textbf{ in} \\ & \quad \textbf{if } y \le s \textbf{ then } cons(y,r) \\ & \quad \textbf{else } cons(s, sort'(y, cdr(r))) \end{array}}$$

which is exactly an insertion program.

Merge sort. Merge sort takes a list, separates it into two sublists of roughly equal lengths, recursively sorts both, and then merges the two sorted sublists. Consider its definition below and, again, a change operation that adds an element y to the input list.

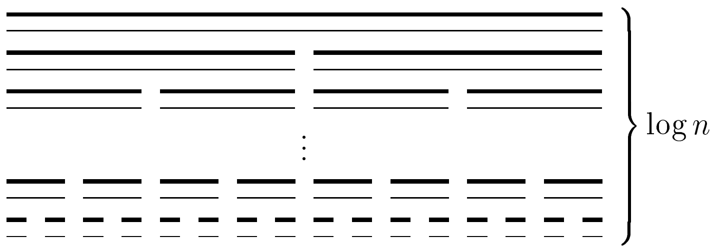
$$
\boxed{
\begin{array}{lcl}
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else if } null(cdr(x)) \textbf{ then } x \\
 & & \textbf{else } merge(sort(odd(x)), sort(even(x))) \\
odd(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else } cons(car(x), even(cdr(x))) \\
even(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else } odd(cdr(x)) \\
merge(x, y) & = & \textbf{if } null(x) \textbf{ then } y \\
 & & \textbf{else if } null(y) \textbf{ then } x \\
 & & \textbf{else if } car(x) \leq car(y) \textbf{ then} \\
 & & \quad cons(car(x), merge(cdr(x), y)) \\
 & & \textbf{else } cons(car(y), merge(x, cdr(y))) \\
x \oplus y = & & cons(y, x)
\end{array}
}
$$

If we are given that $sort(cons(y, x))$ equals merging a single-element list of y with the previously sorted list of x, then we can straightforwardly obtain an incremental version, which is essentially an insertion with a constant-factor overhead.

$$\boxed{sort'(y, r) = merge(cons(y, nil), r)}$$

However, this equality is nontrivial. Even proving it needs a nontrivial induction. If no additional equality is given, can we compute merge sort of the new list more efficiently than computing from scratch? The answer is yes, simply using cache-and-prune.

The derivation is straightforward. We illustrate the idea with the following picture, rather than code as for selection sort. The top wider line denotes list x; the thinner line below denotes $sort(x)$; the two wider lines below denote $odd(x)$ and $even(x)$, respectively;, and the two thinner lines below denote $sort(odd(x))$ and $sort(even(x))$, respectively. This goes down until each list has a single element.



First, cache all the intermediate results of sorted sublists, as depicted above. Then, incrementalize after a new element y is added to the top wider line. Clearly, y belongs to one of the two wider lines immediately below, which means the intermediate result for the other winder line can be reused. This goes down until a single element is left, and comes back up to perform merge at each level. The depth is $\log n$, and the amount of work for each level, from bottom to top, is 1 unit, 2 units, 4 units, ... $n/2$ units, with a total of $O(n)$ time. Finally, prune out intermediate results such as $odd(x)$ and $even(x)$. The resulting incremental merge sort is as follows:

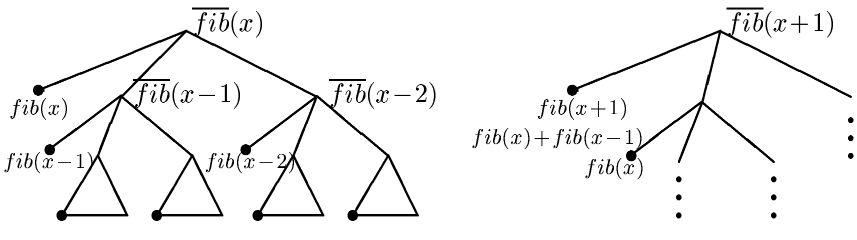
$$
\begin{array}{ll}
\widehat{sort}'(y,\hat{r}) = & \textbf{if } null(1st(\hat{r})) \textbf{ then } < cons(y,nil) > \\
& \textbf{else if } null(cdr(1st(\hat{r}))) \textbf{ then} \\
& \quad <merge(cons(y,nil), 1st(\hat{r})), <cons(y,nil)>, <1st(\hat{r})>> \\
& \textbf{else let } u_1 = \widehat{sort}'(y,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

where function *merge* is as in the given program. Note that this incremental merge sort takes $O(n \log n)$ space rather than $O(n)$ space. However, no additional equality is needed for this derivation. The resulting program takes $O(n)$ time instead of $O(n \log n)$ time. To our knowledge, this algorithm was not known previously, probably due to its $\log n$ factor of additional space, but it reduces the running time by a $\log n$ factor without additional knowledge about the problem.

Fibonacci function. Fibonacci function is a classical example where powerful optimizations are needed for efficient computation.

$$
\begin{array}{ll}
fib(x) = & \textbf{if } x \leq 1 \textbf{ then } 1 \\
& \textbf{else } fib(x-1) + fib(x-2)
\end{array}
$$

First, cache all intermediate results of fib. We obtain function $\overline{fib}$ that, if run, returns a tree of exponential size, as depicted on the left of the picture below, where $fib(x)$ is the first node on the left, and $\overline{fib}(x-1)$ and $\overline{fib}(x-2)$ are recursive subtrees. Then, incrementalize to compute $\overline{fib}(x+1)$, whose computation tree is depicted on the right of the picture, where the node $fib(x+1)$ equals $fib(x) + fib(x-1)$ by definition, and both $fib(x)$ and $fib(x-1)$ can be retrieved from the cached results on the left. Overall, only the two leftmost nodes need to be used and maintained for the incremental computation. Finally, all other results are pruned.



Using this derived incremental version to form a new Fibonacci program, we obtain

$$
\begin{array}{ll}
fib_1(x) = & \textbf{if } x \leq 1 \textbf{ then } <1,1> \\
& \textbf{else let } r_1 = fib_1(x-1) \textbf{ in} \\
& \quad <1st(r_1) + 2nd(r_1), 1st(r_1)>
\end{array}
$$

While the straightforward program takes $O(2^n)$ time, the optimized program takes only $O(n)$ time.

Further examples. Below are some further examples taken from VLSI design, string processing, graph algorithm, image processing, respectively. Each one is a nontrivial problem. We only summarize the time complexities of the straightforward versions and the optimized versions for them. For the square-root example, all expensive operations in hardware (multiplications and exponentiations) are replaced by inexpensive operations (additions, subtractions, and shifts).

Problem	Straightforward	Optimized
non-restoring binary integer square root	$k^2, 2^i$	$+, -, *2, /2$
string editing problem	$O(3^{n+m})$	$O(n * m)$
dag path sequence problem	$O(2^n)$	$O(n^2)$
local neighborhood problem	$O(n^2 m^2)$	$O(n^2)$

4 Discussion

Summary of the approach. We have given an approach for deriving incremental programs that use the old result, intermediate results, and auxiliary information. Our work is the first in which these three aspects are explicitly identified and put into a general framework that unifies existing methods and techniques. The approach is modular, systematic, and powerful. Details of the analysis and transformations for P1, P2, and P3 are described in separate papers [9, 8, 7]. These papers also contain detailed as well as additional examples.

Even though we have presented the approach and examples using programs written in a simple functional language, the underlying principle of incrementalization is general and applies to other languages as well. For example, we have applied the approach to incrementalize loops and aggregate array computations [6].

The incrementalization approach has a spectrum of applications, from compiler optimizations to programming methodologies:

- fully automatic for optimizing compilers;
- semi-automatic in transformational programming;
- off-line as methodology for program efficiency improvement and deriving incremental algorithms.

For example, our semi-automatic implementation, CACHET [5], can be used for transformational programming; an automatic version is being implemented for use by a software development group at Motorola. Also, the optimization of aggregate array computations [6] is automatic and is being implemented for use in optimizing compilers.

CACHET: A prototype implementation. CACHET is a semi-automatic supporting tool for deriving incremental programs [5]. It is implemented using the Synthesizer Generator [14], a system for generating language-based editing environments.

Compared with other program transformation systems, such as KIDS [15] and APTS [10], CACHET benefits from many existing techniques and tools built into the Synthesizer Generator, in particular, for generating attribute-grammar-based programming environments. We can easily use many program manipulation facilities—lexical analysis, parsing, semantic analysis, pretty printing, interactive editing, etc. Furthermore, program analyses can be specified using attribute equations, and thus, with incremental attribute evaluator automatically generated from these equations [14], we achieve incremental program analysis as programs are repeatedly transformed. Major implementation effort is summarized as follows:

- *program transformation* by *direct tree manipulation*

  built-in transformations: $\begin{cases}\text{unfolding, simplification, specialization}\\ \text{transformations enabled by equality analyses}\\ \text{invoking buffers for recursive transformations}\end{cases}$

- *program analysis* by *attribute evaluation*

  attributes: $\begin{cases}\text{propagating global information, collecting context information}\\ \text{analyzing dependencies, reasoning about equalities}\end{cases}$

- *external input* as *annotation*

  annotations: $\begin{cases}\text{information not solely depending on the tree}\\ \text{information too inconvenient/costly to compute from the tree}\end{cases}$

where annotations are external user inputs and are conceptually neither parts of the program tree nor conventional attributes.

CACHET has been used to derive numerous incremental programs. In particular, we have used it on many examples when we studied the transformations for caching intermediate results and discovering auxiliary information, and we found it to be a very helpful tool. Figure 6 is a snapshot of system CACHET, in the middle of incrementalizing selection sort.

Conclusion. Incremental computation is important for efficient computation. We've given a general systematic approach for incrementalization. Our ultimate goal is to build powerful tools for program efficiency improvement. This involves implementing powerful and efficient program analyses and transformations.

A number of improvements to the current work are needed. First, improve the efficiency and effectiveness of the program analyses and transformations used, and develop analyses and transformations for other language features. Second, study theoretical foundations of incremental computation, including complexity theory and classification of problems by their degree of incrementality. Third, develop specialized methods and techniques for special applications. Finally, continue the development of CACHET as well as related program analysis and transformation systems.

References

[1] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, May 1996.

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

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

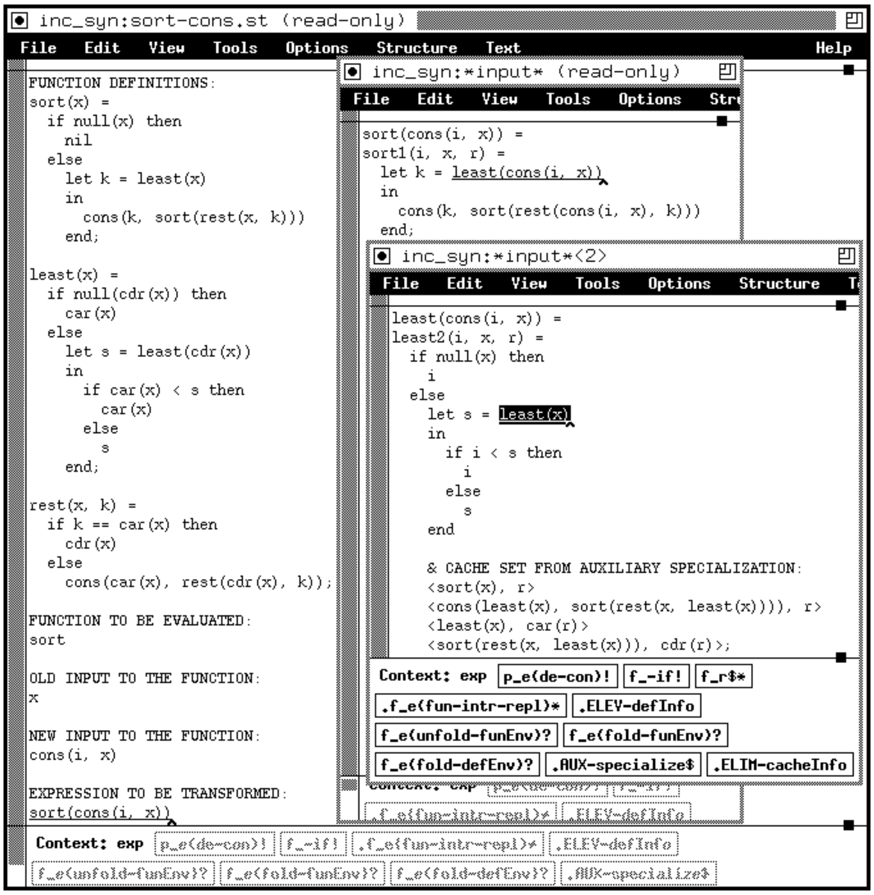


Figure 6: Snapshot of CACHET

[4] 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.

[5] 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.

[6] 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.

[7] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 157–170. ACM, New York, Jan. 1996.

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

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

[10] R. Paige. Viewing a program transformation system at work. In M. Hermenegildo and J. Penjam, editors, *Proceedings of Joint 6th International Conference on Programming Languages: Implementations, Logics and Programs and 4th International Conference on Algebraic and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer-Verlag, Berlin, Sept. 1994.

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

[12] 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.

[13] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 502–510. ACM, New York, Jan. 1993.

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

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

[16] 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.

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

Y. Annie Liu is an assistant professor in the Computer Science Department at Indiana University. Her primary research interests are programming languages, compilers, and software systems. She is particularly interested in general and systematic approaches to improving the efficiency of computations.

Liu received a BS from Peking University, an ME from Tsinghua University, and an MS and a PhD from Cornell University, all in computer science. She was a post-doctoral associate at Cornell University from 1995 to 1996. Liu is a member of IFIP WG2.1. She has served on the program committees of several international conferences, and she co-chairs the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems.