Efficient Computation via Incremental Computation

Yanhong A. Liu*

December 1999

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 of $f(x)$ that can be inexpensively maintained.

Since every non-trivial computation proceeds by iteration or recursion, the approach can be used for achieving efficient computation by computing each iteration incrementally using an appropriate incremental program. This approach has applications in interactive systems, optimizing compilers, transformational programming, and many other areas, where problems were previously solved mostly in ad hoc and often error-prone ways. The design and implementation of a prototype system, CACHET, for incrementalization 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 $f(x)$. Here are some examples:

- Suppose f is a program $sort$, x is a list of numbers, and $\oplus$ prepends 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 $sort'(y, r) = sort(cons(y, x))$. While computing $sort(cons(y, x))$ from scratch must take $\Omega(n \log n)$ time, which is the lower bound for general sorting, computing $sort'(y, r)$ by inserting y into r takes only $O(n)$ time.

*This work was supported in part by ONR Grant N00014-92-J-1973, NSF Grant CCR-9503319, NSF Grant CCR-9711253, ONR Grant N00014-99-1-0132, and a grant from Motorola. 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 $\oplus$ increments 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 a repetitive fashion, by iteration or recursion. For efficiency, each iteration needs to be computed incrementally using the stored results of the previous iteration. Here enters *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.

Incremental computation has wide applications throughout computer science. They can be divided into two groups: those that explicitly handle changes and require fast response time, and those that repeatedly perform expensive computations and require overall optimization. The former includes interactive systems such as programming environments and document editing systems, reactive systems such as control systems, systems with soft or hard real-time requirements, distributed systems where failures occur dynamically, and dynamic management of large databases. The latter includes optimizing compilers, transformational program development, algorithm design, and programming methodology.

In all these applications, it is usually much easier to write straightforward programs without worrying about efficient incremental updates, e.g., writing a batch attribute evaluator without worrying about changes to the syntax tree [3, 72], writing a straightforward recursion for a combinatorial optimization problem without worrying about memoization or tabulation [19, 48], designing a VLSI chip by writing an algorithm using standard arithmetic operations, not shifts of bits and so on [56, 44], or writing an image processing program without worrying about eliminating redundant computations on overlapping regions of the image [88, 47]. More importantly, it is much easier to understand such straightforward programs, to prove their correctness, and to update and reuse them as needed. Then, to achieve the required computation efficiency in all these applications, it is highly desirable to be able to derive an efficient incremental version from a straightforward batch version following a systematic method and furthermore with mechanical assistance.

A general systematic transformational approach. This work studies a general and systematic transformational approach to incrementalization for improving the efficiency of computation. 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 $f(x)$, i.e., the return value of $f(x)$, P2: the intermediate results of $f(x)$, i.e., values computed in computing $f(x)$, not necessarily the return value, and

P3: auxiliary information of $f(x)$, i.e., values not in the return value or intermediate results of $f(x)$ and that can be inexpensively maintained for efficiently computing $f(x \oplus y)$.

Using the value of $f(x)$ gives incrementality over computing $f(x \oplus y)$ from scratch; using the intermediate results of $f(x)$ gives greater incrementality than using only the value of $f(x)$; using auxiliary information gives 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 [69]. We separate all the work into three categories.

The first category consists of *incremental algorithms*, which includes dynamic algorithms and on-line algorithms. These are particular algorithms manually derived to handle particular problems and particular input changes. Examples are incremental parsing [40], attribute evaluation [73, 85], data-flow analysis [76], circuit evaluation [6], constraint solving [30], transitive closure [86], shortest path [70], minimum spanning tree [29, 23], connectivity [23, 71], and scheduling. 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 goal is to study general methods for incremental problems. The idea is to allow different application programs to run in such a framework without deriving explicit incremental algorithms. Examples are incremental attribute evaluation framework [72, 73], function caching [67], incremental lambda reduction [25, 1], traditional partial evaluation [80], change detailing network [87], and program abstraction [38]. Each such framework provides some language for describing application programs, defines 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 program and any input change; it also aims to be effective on each given problem by deriving an incremental program using special properties of the problem, as does the first class. 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 [17, 5], finite differencing in transformational programming [61, 79], and maintaining loop invariance in programming methodology [21, 35]. It is easy to see that methods in this class have the potential to be most general and powerful.

Our work falls into the third category. The principles of our approach are essentially the same as those underlying the work by Allen, Cocke, Kennedy, and others on strength reduction [4, 17, 34, 16, 5, 59, 42], by Earley on high-level iterators [22], by Fong and Ullman on inductive variables [28, 26, 27], by Paige, Schwartz, and Koenig on finite differencing [57, 61, 58], by Dijkstra, Gries, and Reynolds [21, 35, 74, 36] on maintaining and strengthening loop invariants, by Boyle, Moore, Manna, and Waldinger on induction, generalization, and deductive synthesis [13, 53, 54], by Dershowitz on extension techniques [20], by Bird on promotion and accumulation [9, 10], by Broy, Bauer, Partsch, and others on transforming recursive functional programs in CIP [14, 7, 63], by Smith on finite differencing of functional programs in KIDS [79], as well as the work pioneered by Michie on memoization [55, 18, 39, 82] and by others on related techniques [15, 8, 64] . The most basic idea can be traced back to the Difference Engine of Charles Babbage in the 19th century [31].

Our approach exploits program semantics, using analysis of data structures and control structures, to discover incrementality not directly embedded in particular primitives. It comprises step-wise program analysis and transformation modules that can, for the most part, be mechanized. Therefore, compared to work by Allen, Cocke, Kennedy, Earley, Fong, Paige, etc., where a set of rules is developed to transform expensive primitive operations into efficient incremental operations, our approach is more semantics-based and more general; compared to work by Dijkstra, Gries, Boyle, Manna, Dershowitz, Bird, Broy, Smith, etc., where only general strategies are suggested, our approach is more systematic and more automatable.

Outline. This rest of the paper is organized as follows. Section 2 describes the approach, in particular, solutions to P1: exploiting the previous result, P2: caching intermediate results, and P3: discovering auxiliary information, and uses a small example. Section 3 gives additional examples. Section 4 discusses correctness, power vs. limitation, applications and usage, and a prototype system, CACHET.

2 Methods and techniques

This section describes our methods and techniques for addressing subproblems P1 to P3; together they form an overall approach for incrementalization.

Language and example. We use a first-order, call-by-value functional language with all the most basic program constructs. 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 compares the sum of odd positions and the 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 *cmp* as a small example to illustrate our approach.

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

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

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.

2.1 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., $f(x)$, as shown in the box for $f(x \oplus y)$, 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 with 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 shown on the right at the bottom of Figure 2. To summarize, we first introduce function f' with

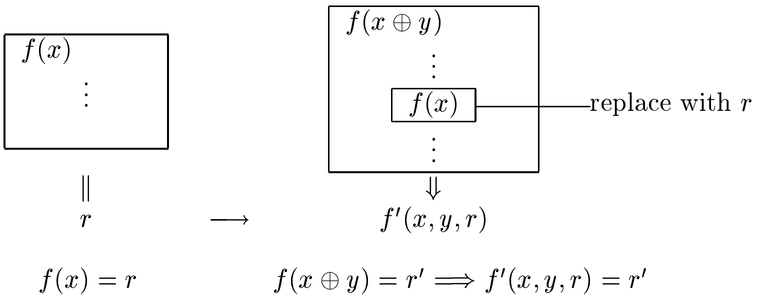


Figure 2: Exploiting the previous result.

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'. Since replacement of a subcomputation may cause other subcomputations on which the replaced subcomputation depends to become dead, dead code elimination is performed at the end.

We can do much better than only directly using the value of $f(x)$. We exploit the semantics of each program construct, i.e., data structures and control structures. For example, 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 by $g(x)$ sitting on the bottom of the box for $f(x)$ on the left of Figure 3. Thus, a subcomputation $g(x)$ in $f(x \oplus y)$ on the right can be replaced with a retrieval from r. Also, subcomputation $g(x)$ in $f(x \oplus y)$ on the right may appear in certain context, e.g., in the true branch of a conditional expression. If $f(x)$ on the left 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.

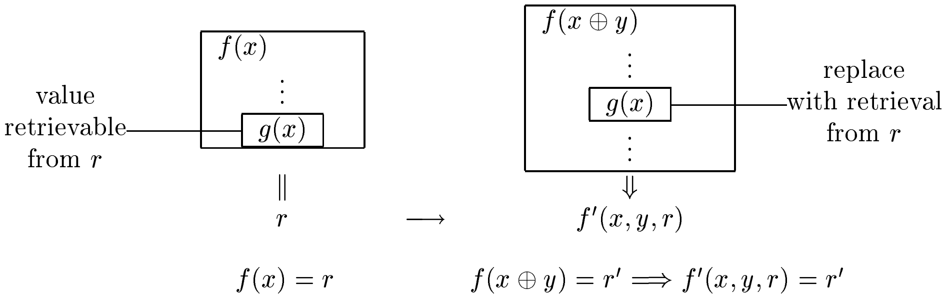


Figure 3: Exploiting the previous result (cont'd).

The transformation steps are as previously summarized, except that replacements using the cached result are based also on equality reasoning and auxiliary specialization, as illustrated above.

A number of important analyses and transformations are used here. Unfolding and algebraic simplification are among the most basic transformations [15], while equality reasoning and auxiliary specialization, which can be regarded as a kind of powerful equality reasoning, could be arbitrarily powerful. However, limiting the power of these transformations, by using simple rewrite rules on data structures and control structures, such as $car(cons(a, b)) = b$ and **if** *true* **then** a **else** $b = a$, and using fully automatable analyses for arithmetic and booleans [66, 41], we have been able to derive all the examples discussed in this paper and others that we have seen and tried. Two efficient static analyses are also used: a forward dependence analysis to identify subcomputations depending only on x, and a backward dependence analysis to identify dead code.

As for most program transformation techniques, it should be noted that the quality of the resulting program depends on that of the original program. In the worst case, if no replacement with retrievals can be done, then $f(x \oplus y)$ is computed as from scratch. We will show in Section 3 how our method derives different incremental programs for three different sorting programs.

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 of sum is simplified to x. Then replace $sum(x)$ with r. We obtain

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

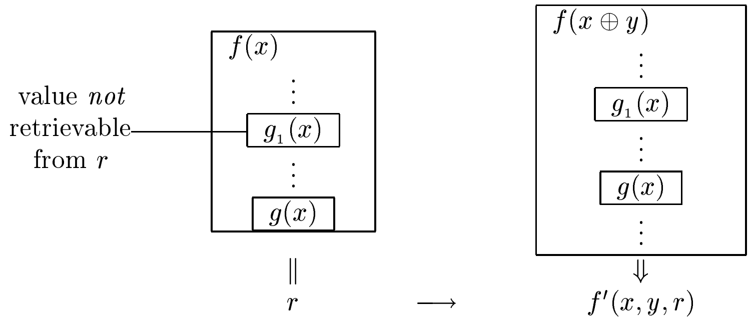
where parameter x to sum' is dead and eliminated. We have, if $r = sum(x)$, then $sum'(y, r) = sum(cons(y, x))$. While $sum(cons(y, x))$ takes $O(n)$ time to compute, $sum'(y, r)$ takes only $O(1)$ time and needs one unit of space to hold the old result.

2.2 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 $g_1(x)$ and the additional bottom line of formulas. Basically, a subcomputation $g_1(x)$ in $f(x \oplus y)$ on the right may also be computed in computing $f(x)$ on the left, but its value is not retrievable from the result r of $f(x)$. If we know 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 shown on the left of the bottom line in 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 $\hat{f}(x \oplus y)$, as shown on the right of the bottom line in Figure 4. The hard problem is that $f(x)$ may compute a huge number of intermediate results. How can we identify useful intermediate results?

We propose a three-stage method called *cache-and-prune*. Stage I constructs a function $\bar{f}$ that extends f to return all intermediate results computed by f. The return value of $\bar{f}$ is a tree structure, mirroring the control structure incurred by recursive calls, where the original value of f is, for convenience, the leftmost child of the root. Now that all intermediate results are cached in the return value of $\bar{f}$, Stage II incrementalizes $\bar{f}$ under $\oplus$ to obtain an incremental version $\bar{f}'$ that can use all of them, as done for P1. Function $\bar{f}'$ computes a new tree of intermediate results, but only the result corresponding to the new value of f is originally desired. Stage III analyzes $\bar{f}'$ to determine all cached values needed for computing the desired value and prunes out the rest in both $\bar{f}$ and $\bar{f}'$, yielding $\hat{f}$ and $\hat{f}'$, respectively.

The method for P2 consists of three relatively independent stages and thus is modular. Stage I gives us maximality by providing all intermediate results possibly used by Stage II. Stage II uses these intermediate results for the exclusive purpose of incrementalization and is reduced to P1. Stage III gives us a kind of minimality by preserving only 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 used by Stage II. Thus, the overall method for P2 has a kind of optimality with respect to the method used in Stage II. The analyses and transformations used for Stages I and III are simple and efficient.

The method for P2 can be used for general program optimization via caching, by incrementalizing the body of a loop or recursion. Since intermediate results are computed by the original program anyway, caching and using them will not increase the asymptotic running time of the transformed program. It can drastically decrease the running time if the original program performs repeated subcomputations. We illustrate this with the classical Fibonacci function in Section 3.

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

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

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.

Use cache-and-prune. First, caching all intermediate results yields function $\overline{cmp}$ below on the left, where functions $\overline{odd}$, $\overline{sum}$, $\overline{even}$, and $\overline{prod}$ return their intermediate results in a similar fashion and are omitted. We use <> to denote a tuple, and selectors $1st$, $2nd$, and so on to select the corresponding components of a tuple. Then, incrementalizing $\overline{cmp}$ under the given $\oplus$ yields function $\overline{cmp}'$ below on the right, such that if $\bar{r} = \overline{cmp}(x)$, then $\overline{cmp}'(y_1, y_2, \bar{r}) = \overline{cmp}(cons(y_1, cons(y_2, x)))$.

$$\begin{array}{ll} \overline{cmp}(x) = & \textbf{let } u_1 = \overline{odd}(x) \textbf{ in} \\ & \textbf{let } u_2 = \overline{sum}(1st(u_1)) \textbf{ in} \\ & \textbf{let } u_3 = \overline{even}(x) \textbf{ in} \\ & \textbf{let } u_4 = \overline{prod}(1st(u_3)) \textbf{ in} \\ & < 1st(u_2) \leq 1st(u_4),\ u_1,\ u_2,\ u_3,\ u_4 > \end{array}$$

$$\begin{array}{ll} \overline{cmp}'(y_1, y_2, \bar{r}) = & \textbf{let } u_1 = < cons(y_1, 1st(2nd(\bar{r}))),\ 2nd(\bar{r}) > \textbf{ in} \\ & \textbf{let } u_2 = < y_1 + 1st(3rd(\bar{r})),\ 3rd(\bar{r}) > \textbf{ in} \\ & \textbf{let } u_3 = < cons(y_2, 1st(4th(\bar{r}))),\ 4th(\bar{r}) > \textbf{ in} \\ & \textbf{let } u_4 = < y_2 * 1st(5th(\bar{r})),\ 5th(\bar{r}) > \textbf{ in} \\ & < 1st(u_2) \leq 1st(u_4),\ u_1,\ u_2,\ u_3,\ u_4 > \end{array}$$

Finally, pruning all returned components not needed for computing $1st(u_2) \leq 1st(u_4)$ in $\overline{cmp}'(y_1, y_2, \bar{r})$, we obtain functions $\widehat{cmp}$ and $\widehat{cmp}'$ below, 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 $cmp(x) = 1st(\widehat{cmp}(x))$.

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

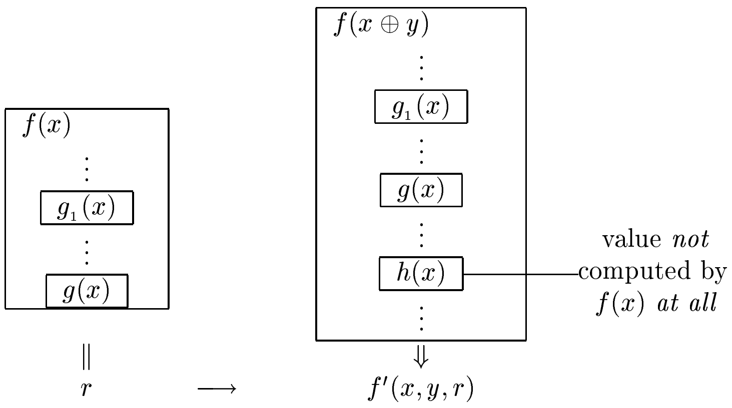
While $cmp(cons(y_1, cons(y_2, x)))$ takes $O(n)$ time, $\widehat{cmp}'(y_1, y_2, \hat{r})$ takes only $O(1)$ time and needs two additional units of space for two intermediate results.

2.3 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. Even for manually derived incremental algorithms, only a small number of special auxiliary data structures have been studied. We propose a systematic method that can discover a general class of auxiliary information. The idea is illustrated in Figure 5 and is explained below. Note that Figure 5 is the same as Figure 4 except for the additional box for $h(x)$ on the right and the different bottom line of formulas.

The method has two phases. In Phase A, we transform $f(x \oplus y)$, shown on the right of Figure 5, to separate subcomputations on x from those on y, as done for P1. If the value of a subcomputation, e.g., $g(x)$ or $g_1(x)$ in the box for $f(x \oplus y)$, can be retrieved from the return value of $f(x)$ or an intermediate result of $f(x)$, e.g., as $g(x)$ or $g_1(x)$ respectively in the box for $f(x)$ on the left, then it is left alone. However, if the value of a subcomputation depending only on x, e.g., $h(x)$ in the box for $f(x \oplus y)$, can not be retrieved from either, then it is collected as a candidate auxiliary information. In Phase B, we determine whether such information can be used and maintained for efficient incremental computation, as done for P2. We extend $f(x)$ to compute and cache the candidate auxiliary information, as well as intermediate results, incrementalize the resulting program, and prune out useless values and computations. This yields the resulting programs $\tilde{f}$ and $\tilde{f}'$, as shown in the bottom line of Figure 5, where $\tilde{f}$ returns useful intermediate results as well as auxiliary information, and $\tilde{f}'$ incrementally uses and maintains them.

The overall method for P3 is composed of analyses and transformations basically as those used for P1 and P2. Phase A uses transformations as those for P1, followed by a forward dependence analysis to identify subcomputations depending on x but not replaced with retrievals from the result of $\bar{f}(x)$, i.e., the return value and intermediate results. Phase B uses the transformations for P2, except that $\bar{f}(x)$ is further extended to compute the candidate auxiliary information. Thus, we have reduced a difficult problem to modular steps where solutions to previous problems can be used. Even though the overall method is complex, each module is relatively simple.



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

Figure 5: Discovering auxiliary information.

Since auxiliary information is not computed by the original program, we use such information only if we can conservatively determine that it can be efficiently computed and maintained. Actually, to obtain incremental programs that are asymptotically at least as fast, we only need to require that auxiliary information be computed initially as fast as the original program. Usually, the cost of this initial computation is amortized over repeated computation using the incremental program, and efficient use and maintenance of the auxiliary information allow the overall computation to be much faster.

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

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

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 method, we obtain functions $\widetilde{cmp}$ and $\widetilde{cmp}'$ below, such that if $\tilde{r} = \widetilde{cmp}(x)$, then $\widetilde{cmp}'(y, \tilde{r}) = \widetilde{cmp}(cons(y, x))$, and $cmp(x) = 1st(\widetilde{cmp}(x))$.

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

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

3 Additional examples

This section gives additional examples. We use three different sorting programs to illustrate the power of our method for deriving incremental programs, and we use the classical Fibonacci function to illustrate how our method is used for optimizing recursive programs. More examples are summarized at the end of this section.

The table below 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, where n is the size of the input list. 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, where n is the input number; if we use n for the size of the input number (e.g., in bits), as in standard complexity theory, then the improvement is from $O(2^{2^n})$ to $O(2^n)$.

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	$O(2^n)$	$O(n)$

3.1 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 $sort(x)$ with 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 $insert$.

3.2 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 applied to the new list can be transformed to use the previously sorted list. Our approach to P1 allows us to derive a definition of insertion not given in the original program.

$$\boxed{\begin{array}{lcl}
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
& & \textbf{else let } k = least(x) \textbf{ in} \\
& & \quad\quad cons(k, sort(rest(x,k))) \\
least(x) & = & \textbf{if } null(cdr(x)) \textbf{ then } car(x) \\
& & \textbf{else let } s = least(cdr(x)) \textbf{ in} \\
& & \quad\quad \textbf{if } car(x) < s \textbf{ then } car(x) \textbf{ else } s \\
rest(x,k) & = & \textbf{if } k = car(x) \textbf{ then } cdr(x) \\
& & \textbf{else } cons(car(x), rest(cdr(x),k)) \\
\multicolumn{3}{l}{x \oplus y = \ cons(y,x)}
\end{array}}$$

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

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

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

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

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

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

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

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

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

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

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

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 with 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 with $car(r)$. Finally, recursive call $sort(cons(y, rest(x, s)))$ is replaced with $sort'(y, rest(x, s), sort(rest(x, s)))$, where $sort(rest(x, s))$ in the latter is replaced with $cdr(r)$. We obtain

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

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

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

which is exactly an insertion program.

3.3 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.

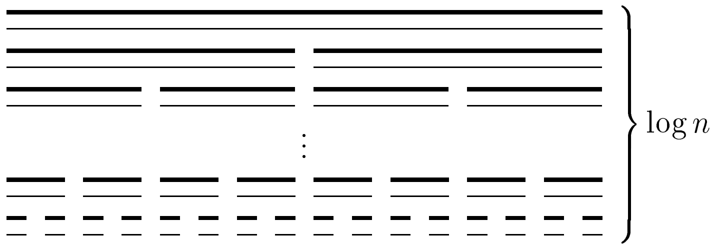
$$
\boxed{
\begin{array}{lcl}
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
& & \textbf{else if } null(cdr(x)) \textbf{ then } x \\
& & \textbf{else } merge(sort(odd(x)), sort(even(x))) \\
odd(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
& & \textbf{else } cons(car(x), even(cdr(x)))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

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:

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

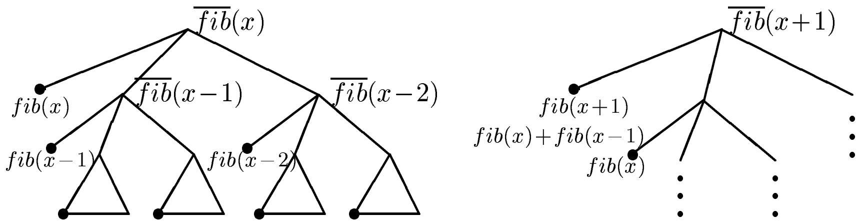
where function $merge$ is as in the given program. Note that this incremental merge sort takes $O(n \log n)$ space rather than $O(n)$ space. However, no additional equality is needed for this derivation. The resulting program $\widehat{sort}'(y,\hat{r})$ takes $O(n)$ time while sorting from scratch using $sort(cons(y,x))$ takes $O(n \log n)$ 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.

3.4 Fibonacci function

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

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

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

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

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

3.5 More examples

Below are more examples taken from graph algorithms [9, 50], VLSI design [56, 44], image processing [88, 47], and string processing [68, 48], respectively. Each one is a nontrivial problem.

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

The table summarizes the time complexities of the straightforward versions and the optimized versions, where n and m are the sizes of the input parameters. For the square root example, all expensive operations in hardware (multiplications and exponentiations) are replaced with inexpensive operations (additions, subtractions, and shifts).

The dag path sequence problem was used by Bird to illustrate an important program transformation method called promotion and accumulation [9], where an $O(n^2)$ time recursive program is derived (incorrectly in [9] but corrected in [10]). It is one of our first examples that shows the use of auxiliary information [50], where an efficient program is obtained easily following our systematic method.

The square root example is from formal hardware design, where an efficient design was derived manually and proved correct using a theorem prover [56]. We showed how an efficient design can be obtained systematically using incrementalization [44]. As a result, we also discovered an unnecessary shift operation done in [56] and optimizations that were unnecessarily delayed to the lower-level hardware in [56].

The local neighborhood problem is typical in image processing [84, 37, 88], where aggregate computations are performed for the neighbors of pixels and these neighbors overlap. We identify such aggregate computations as function f and appropriate update operations as $\oplus$, and incrementalize the aggregate computations; to our knowledge, our method is the first that can automatically derived such optimizations [47].

The string editing problem represents a class of combinatorial optimization problems, where dynamic programming is used to obtain efficient polynomial-time algorithms [68, 19]. Our incrementalization method allows us to systematically derive such efficient algorithms [48] for all examples found in [2, 68, 19]. Furthermore, for all but two of the problems, no array is needed, i.e., linked data structures are sufficient; also, pruning enables us to use only the amount of space need, obviating the additional space optimization described in, e.g., [19, pages 318–319].

4 Discussion

We have given an approach for deriving incremental programs that use the old result, intermediate results, and auxiliary information. The approach is modular. On the one hand, we can regard each of the methods for P1, P2, and P3 that uses additional values as an extension to the previous method. On the other hand, we can view the previous methods as aids to the next method. Indeed, a user may decide to use only the method for P1 so as not to use any additional space, or may decide to use the method for P3 so as to use all three kinds of values. The approach is general and systematic, as discussed below. Details of the analyses and transformations for P1, P2, and P3 are described in separate papers [52, 51, 50], respectively. These papers also contain more extensive discussions, related work, and detailed as well as additional examples.

Correctness. Each of the methods for P1, P2, and P3 preserves the semantics in the sense that if the original program terminates with a value, then the incremental program terminates with the same value, exactly as summarized in the bottom line in each of the Figures 2 to 5. This is because all transformations in P1, P2, and P3 are based on function definitions, algebraic laws, replacements of equals with equals, or dependence analyses. These transformations preserve the exact semantics with the exception that unfolding by definition and eliminating useless computations may make a non-terminating computation terminate. Note that our transformations do not increase non-termination as we do not use folding (which could be dangerous [77]); calls to incremental versions are only used to replace existing function calls.

Our method also guarantees that the incremental program computes asymptotically at least as fast as computing from scratch using the original program, provided that the auxiliary information used, if any, can be initiated at least as fast. Note that we can not guarantee that an incremental program is always faster; indeed, it is undecidable whether such a faster program exists at all. However, if we can conservatively determine that certain expensive subcomputations are replaced with retrievals from the cached results, then we know that the incremental program runs faster; also, if expensive subcomputations are repeated in a given iterative or recursive program, then incrementalizing the body of the loop or recursion yields a much faster program. By expensive computations, we mean those that contribute to the asymptotic running time of a program. Work in program running time analysis, e.g., [83, 75], can help determine this; we have been studying such analyses in our own recent work as well [46, 32].

Power and limitation. Our work is the first in which reusing the previous result, caching intermediate results, and discovering auxiliary information are explicitly identified and put into a general framework. It unifies existing methods and allows domain-specific properties to be effectively used. The power of the overall method depends on the power of the equality reasoning and time analysis used and on the ability to discover special auxiliary data structures. In principle, if these were arbitrarily strong, then the method can derive all incremental algorithms or efficient algorithms that use incremental bodies. However, even limiting equality reasoning to use simple rules involving data structures and control structures, limiting time analysis to use simple heuristics (e.g., function calls are more expensive than other primitives), and exploiting existing auxiliary data structures by building them in a library and using them from there, the method is able to derive all the examples we have seen and tried in the program transformation literature.

The overall method combines many existing analyses and transformations, also improving some of them [45, 49], and puts them in an overall step-wise procedure. Thus it is systematic and, in particular, automatable modulo the equality reasoning, time analysis, and discovery of special auxiliary data structures. Because of this, we are able to derive many examples easily and quickly, often obtaining clearer, shorter, and more efficient programs, with high confidence in their correctness.

The method as it currently stands has a number of limitations and needs many improvements. First, it lacks an exact characterization of how the power of equality reasoning and time analysis affects the classes of problems that can be incrementalized. Our characterizations so far have been informal. Second, the method needs to be extended to handle other language features. Even though the method is general and the underlying principles apply to other features, precise analyses and transformations need to be studied. For example, we have applied the approach to incrementalize loops [44] and aggregate array computations [47]. The idea is to capture expensive computations in the presence of these other features as functions f, capture updates to the parameters of these computations as operations $\oplus$, and apply incrementalization by exploiting equality reasoning and other analyses based on the semantics of the language features involved. For example, for optimizing aggregate array computations [47], we reduce various analyses to solving constraints on loop variables and array subscripts and use an automated tool, Omega [66], to solve them.

The power of the method in the presence of these other features needs to be characterized as well. The higher level a language is, the easier it is to do equality reasoning and other analyses for it. For example, using techniques similar to incrementalization on a set-theoretic language [61, 58], Paige and others have derived a number of new and improved algorithms for various applications, e.g., [62, 11, 33]. It is well-known that features like goto, pointers, and higher-order functions are generally difficult to analyze, but seldom are straightforward programs or algorithms first written using them. Precise characterization of these needs to be studied specially. Finally, it is not yet known how to systematically discover special auxiliary data structures, e.g., disjoint sets [81], dynamic trees [78], and topology trees [29], even though only a small number of them are used so far in incremental or dynamic algorithms, e.g., [29, 24]. Currently, we can just put them in a library and use them from there. All these problems are open for future study.

Applications and usage. We have described wide applications of incremental computation in Section 1. The method described in this paper can be used for all of them. Based on the degree of automation, the method has a spectrum of usages, from compiler optimizations to programming methodologies. At one end, limiting equality reasoning and time analysis to use fully automatic techniques, the method can be implemented in an advanced optimizing compiler to incrementalize loops or recursions. In the middle of the spectrum, limiting those components to use semi-automatic techniques, the method can be used for transformational program development. At the other end, the method can be followed on paper for deriving incremental algorithms or optimizing straightforward algorithms, e.g., deriving efficient dynamic programming algorithms from straightforward exponential-time recursions.

Implementation. Since the method is systematic, and doing program transformation by hand is tedious and error-prone, it is important to provide automatic assistance to the user by implementing the method. Our prototype system, CACHET, is an initial attempt to implement this powerful method.

CACHET was initially built as a semi-automatic supporting tool for deriving incremental programs [43]. Most of the analyses and transformations are built in, and an interactive environment allows the user to select and invoke them. The original implementation of equality reasoning and time analysis was ad hoc. We have been gradually automating CACHET (for use by a software development group at Motorola) and making equality reasoning more systematic [89]. In particular, equality reasoning on arithmetic operations and boolean operations are now performed using dedicated tools, Omega [66] and MONA [41], respectively. We have also been building a system, ALPA, for automated time analysis [46, 32] and plan to use it in CACHET. Our optimizations for aggregate array computations [47] have also been automated. Our goal is to integrate all analyses and transformations in an open environment, and provide the user with options for different degrees of automation.

CACHET has been used to derive numerous incremental programs, including all the examples discussed in this paper. In fact, currently, CACHET can perform incrementalization fully automatically for all examples in this paper except the dag path sequence problem. Figure 6 is a snapshot of CACHET, in the middle of incrementalizing selection sort.

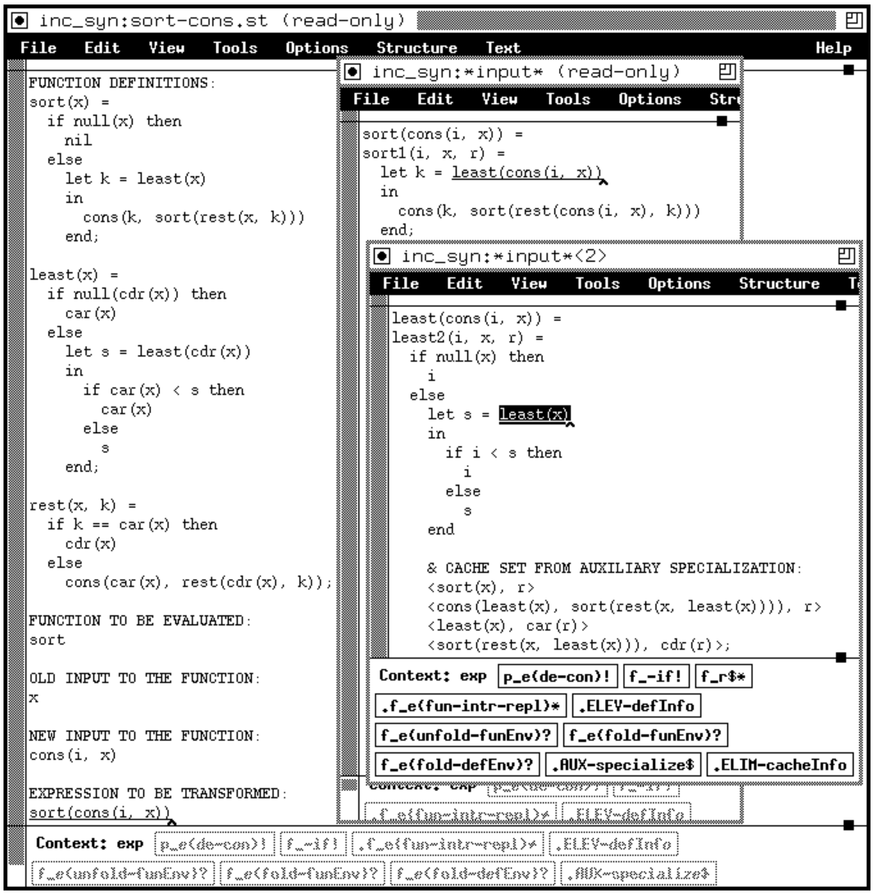


Figure 6: Snapshot of CACHET

The ultimate goal of this work is the construction of a complete system for incrementalization. This by itself poses many challenging research problems, well-known in systems that perform or support program analyses and transformations [72, 12, 60, 43].

CACHET is currently implemented using the Synthesizer Generator [72], a system for generating language-based editing environments. The Synthesizer Generator allows parsing rules, pretty printing rules, and semantic rules to be specified in a declarative fashion, and transformations to be specified as direct rewrite of the program tree. In particular, semantic rules are specified as attribute equations, and attributes are evaluated incrementally as the program is transformed. Thus, the system response is fast, and this is an obvious advantage over other program transformation systems, e.g., CIP [7], KIDS [79], and APTS [60]. However, incrementalization needs circular dependence analyses, external analysis tools, and complex transformations, while efficient circular attribute evaluation, interleaved with complex transformations that may be enabled by external analysis results, is not supported by the Synthesizer Generator or other existing tools and is a subject open for study. Currently, we code these complex analyses and transformations directly as functions or procedures, rather than declaratively, making it difficult to modify, extend, or scale up the system. These are important issues for future research.

Conclusion. Incremental computation is important for efficient computation. We have given a general systematic approach for incrementalization that exploits the old result, intermediate results, and auxiliary information. Our ultimate goal is to build a powerful tool that supports incrementalization and related analyses and transformations.

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] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

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

[4] F. E. Allen. Program optimization. In *Annual Review of Automatic Programming*, volume 5, pages 239–307. Pergamon Press, New York, 1969.

[5] 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.

[6] 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.

[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. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, Dec. 1980.

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

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

[11] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Sci. Comput. Program.*, 24(3):189–220, 1995.

[12] 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.

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

[14] 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.

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

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

[17] J. Cocke and J. T. Schwartz. Programming languages and their compilers; preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[18] N. H. Cohen. Eliminating redundant recursive calls. *ACM Trans. Program. Lang. Syst.*, 5(3):265–299, July 1983.

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

[23] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. In *Proceedings of the 33rd Annual IEEE Symposium on FOCS*, Pittsburgh, Pennsylvania, Oct. 1992. IEEE CS Press, Los Alamitos, Calif.

[24] D. Eppstein, C. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. In *Proceedings of SODA '90*, pages 1–11, 1990.

[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] A. C. Fong. Generalized common subexpressions in very high level languages. In POPL 1977 [65], pages 48–57.

[27] A. C. Fong. Inductively computable constructs in very high level languages. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 21–28. ACM, New York, Jan. 1979.

[28] A. C. Fong and J. D. Ullman. Inductive variables in very high level languages. In *Conference Record of the 3rd Annual ACM Symposium on Principles of Programming Languages*, pages 104–112. ACM, New York, Jan. 1976.

[29] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, Nov. 1985.

[30] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, Jan. 1990.

[31] H. H. Goldstine. Charles Babbage and his analytical engine. In *The Computer from Pascal to von Neumann*, chapter 2, pages 10–26. Princeton University Press, Princeton, New Jersey, 1972.

[32] 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.

[33] D. Goyal. *A Language Theoretic Approach to Algorithms*. PhD thesis, Department of Computer Science, New York University, 2000. To come.

[34] D. Gries. *Compiler Construction for Digital Computers*. John Wiley & Sons, New York, 1971.

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

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

[37] P. T. Highnam. *Systems and Programming Issues in the Design and Use of a SIMD Linear Array for Image Processing*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Apr. 1991.

[38] 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.

[39] J. Hughes. Lazy memo-functions. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 129–146. Springer-Verlag, Berlin, Sept. 1985.

[40] 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.

[41] N. Klarlund. *MONA Version 1.3 User Manual*, Oct. 1998.

[42] J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, 1993.

[43] 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.

[44] Y. A. Liu. Principled strength reduction. In *Proceedings of the IFIP TC2 Working Conference on Algorithmic Languages and Calculi*. Chapman & Hall, London, U.K., Feb. 1997.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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, Venice, Italy, Sept. 1999. Springer-Verlag, Berlin.

[50] 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.

[51] 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.

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

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

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

[55] D. Michie. "memo" functions and machine learning. *Nature*, 218:19–22, Apr. 1968.

[56] 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.

[57] B. Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In POPL 1977 [65], pages 58–71.

[58] R. Paige. Programming with invariants. *IEEE Software*, pages 56–69, Jan. 1986.

[59] R. Paige. Symbolic finite differencing—Part I. In *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 36–56. Springer-Verlag, Berlin, May 1990.

[60] 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.

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

[62] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, Dec. 1987.

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

[64] 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.

[65] *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, Jan. 1977.

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

[67] 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.

[68] P. W. Purdom and C. A. Brown. *The Analysis of Algorithms.* Holt, Rinehart and Winston, 1985.

[69] 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.

[70] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305, 1996.

[71] M. Rauch. Improved data structures for fully dynamic biconnectivity. In *Conference Proceedings of the 26th Annual ACM STOC.* ACM, New York, 1994.

[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 *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, Sept. 1989.

[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. Sands. Total correctness by local improvement in program transformation. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, Jan. 1995.

[78] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

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

[80] 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.

[81] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[82] A. Webber. Optimization of functional programs by grammar thinning. *ACM Trans. Program. Lang. Syst.*, 17(2):293–330, Mar. 1995.

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

[84] 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.

[85] D. Yeh and U. Kastens. Improvements on an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Notices*, 23(12):45–50, 1988.

[86] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, July 1993.

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

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