Static Caching for Incremental Computation

YANHONG A. LIU and SCOTT D. STOLLER
Indiana University
and
TIM TEITELBAUM
Cornell University

A systematic approach is given for deriving incremental programs that exploit caching. The *cache-and-prune* method presented in the article consists of three stages: (I) the original program is extended to cache the results of all its intermediate subcomputations as well as the final result, (II) the extended program is incrementalized so that computation on a new input can use all intermediate results on an old input, and (III) unused results cached by the extended program and maintained by the incremental program are pruned away, leaving a pruned extended program that caches only useful intermediate results and a pruned incremental program that uses and maintains only the useful results. All three stages utilize static analyses and semantics-preserving transformations. Stages I and III are simple, clean, and fully automatable. The overall method has a kind of optimality with respect to the techniques used in Stage II. The method can be applied straightforwardly to provide a systematic approach to program improvement via caching.

Categories and Subject Descriptors: D.1 [**Programming Techniques**]: Automatic Programming—*automatic analysis of algorithms; program transformation*; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*optimization*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Caching, dependence analysis, incremental computation, incremental programs, intermediate results, memoization, optimization, program efficiency improvement, program transformation, static analysis

1. INTRODUCTION

Incremental computation takes advantage of repeated computations on inputs that differ slightly from one another, making use of previously computed results in computing a new output rather than computing from scratch. Methods of incremental computation have widespread application, e.g., optimizing compilers [Aho et al.

This work was supported by ONR Grant N00014-92-J-1973 and NSF Grant CCR-9711253. This article is a revised and extended version of a paper that appeared in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, La Jolla, California, June 1995.

Authors' addresses: Y. A. Liu and S. D. Stoller, Computer Science Department, Indiana University, Bloomington, IN 47405; email: {liu; stoller}@cs.indiana.edu; T. Teitelbaum, Department of Computer Science, Cornell University, Ithaca, NY 14853; email: tt@cs.cornell.edu.

1986; Cocke and Kennedy 1977; Earley 1976], transformational programming [Paige 1983; Partsch 1990; Smith 1990], and interactive editing systems [Ballance et al. 1992; Reps and Teitelbaum 1988].

Given a program f and an input change operation $\oplus$, a program f' that computes the result of $f(x \oplus y)$ efficiently by making use of the value of $f(x)$ is called an *incremental version* of f under $\oplus$. For example, suppose, in a large employee database, a program *salarySum* sums salaries of all employees, and the input change operation is to insert or delete a few employees. Then an incremental program can compute a new sum from the old sum by adding or subtracting the salaries of these few employees. For another example, suppose a programming environment needs all the attributes on a program syntax tree, and the input change operation is to replace any small subtree. Then an incremental attribute evaluation program can compute the new attributes by propagating changes and updating old attributes whose values need to be changed [Reps et al. 1983; Yeh and Kastens 1988].

Often, for efficiently computing a new output after an input change, certain *intermediate results*—results computed in the middle of the old computation, not just its return value—need to be used and maintained. Suppose, in the above employee database example, a program *salaryAve* computes the average salary of employees. Then a new average cannot be computed from the old average and the salaries of the few inserted or deleted employees. However, the number of employees and the sum of their salaries are intermediate results of *salaryAve*. A new average can be computed using them and the salaries of the changed employees, and these intermediate results can be efficiently maintained. For another example, attribute evaluators are often used to return only a synthesized attribute at the tree root [Katayama 1984], so other attributes need not be maintained for batch computation. However, to produce the desired output quickly after a subtree replacement, other intermediate attributes may need to be maintained.

Since programs often compute large numbers of intermediate results, it is often difficult, yet essential, to determine which intermediate results are needed for incremental computation and how to cache them, use them in the incremental computation, and maintain their values for incremental computation after a further input change.

This article describes a systematic method, called *cache-and-prune*, for addressing these problems in incremental computation. The method is based on static program analyses and semantics-preserving program transformations. In other words, it achieves caching statically by transforming programs. The method has three stages. Stage I extends program f to a program $\bar{f}$ that returns all intermediate results. Stage II incrementalizes $\bar{f}$ under $\oplus$ to obtain an incremental version $\bar{f}'$ that can use all intermediate results. Stage III uses the dependencies in $\bar{f}'$, prunes $\bar{f}$ to a program $\hat{f}$ that returns only the useful intermediate results, and prunes $\bar{f}'$ to a program $\hat{f}'$ that uses and incrementally maintains only the useful intermediate results. Since every nontrivial program computes by iteration or recursion, the cache-and-prune method provides a systematic approach for general program optimization via caching by computing each iteration or recursion using an incremental program that exploits intermediate results.

The cache-and-prune method is modular. Each stage has a designated goal.

Stage I caches every intermediate result, "cache as cache can," so to speak. Stage II uses cached results to reduce the running time. Stage III prunes unused results to reduce the space consumption and further reduce the running time. Stages I and III are based on efficient automatic static analyses. Stage II performs algebraic simplifications, replacements using cached results, and dead-code elimination; a systematic method has been studied in Liu and Teitelbaum [1995] and is summarized in this article.

It is the goal of this article to propose the cache-and-prune method, to describe the analyses and transformations for caching and pruning, and to study the role of incrementalization in this method. We also illustrate how to use the method for general program optimization via caching. The generality of the cache-and-prune method contrasts with previous work on caching, which either relies on a fixed set of rules [Allen et al. 1981; Paige and Koenig 1982], applies only to programs with certain properties or schemas [Bird 1980; Cohen 1983; Pettorossi 1984; 1987], requires program annotations [Hoover 1992; Keller and Sleep 1986; Sundaresh and Hudak 1991], *etc.* The dependence analysis for pruning in Stage III uses domain projections to specify specific components of compound values, rather than just heads or tails of list values, and thus complements existing methods for such analyses [Hughes 1990; Jones and Le Métayer 1989; Reps and Turnidge 1996].

This article is organized as follows. Section 2 defines the problem of caching intermediate results. Section 3 outlines the cache-and-prune method and its correctness. Sections 4, 5, and 6 describe caching, incrementalization, and pruning, respectively. Section 7 discusses the power and limitation of the method, the program analysis and transformation techniques used, and the trade-offs between time and space. Section 8 describes examples. Section 9 presents a comprehensive comparison with related work in caching and concludes.

2. DEFINING THE PROBLEM

We use a simple first-order, call-by-value functional programming language. The expressions of the language are given by the following grammar:

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

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

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

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

An input change operation $\oplus$ to a program f_0 combines an old input $x = \langle x_1, ..., x_n \rangle$ and a change $y = \langle y_1, ..., y_m \rangle$ to form a new input $x' = \langle x'_1, ..., x'_n \rangle = x \oplus y$, where each x'_i is some function of x_j's and y_k's. For example, an input change operation $\oplus_1$ to the function foo or fib in Figure 1 may be defined by

$foo(x)$: sum three preceding "foo" numbers of x

$$foo(x) = \textbf{if } x \leq 2 \textbf{ then } 1 \textbf{ else } boo(x) + foo(x-3)$$
$$boo(x) = foo(x-1) + foo(x-2)$$

$fib(x)$: compute Fibonacci number x

$$fib(x) = \textbf{if } x \leq 1 \textbf{ then } 1 \textbf{ else } fib(x-1) + fib(x-2)$$

$sort(x)$: sort a list x using merge sort

$$sort(x) = \textbf{if } null(x) \textbf{ then } nil \textbf{ else if } null(cdr(x)) \textbf{ then } x \textbf{ else } merge(sort(odd(x)), sort(even(x)))$$
$$odd(x) = \textbf{if } null(x) \textbf{ then } nil \textbf{ else } cons(car(x), even(cdr(x)))$$
$$even(x) = \textbf{if } null(x) \textbf{ then } nil \textbf{ else } odd(cdr(x))$$
$$merge(x,y) = \textbf{if } null(x) \textbf{ then } y \textbf{ else if } null(y) \textbf{ then } x \textbf{ else if } car(x) \leq car(y) \textbf{ then } cons(car(x), merge(cdr(x), y)) \textbf{ else } cons(car(y), merge(x, cdr(y)))$$

Fig. 1. Example function definitions for *foo*, *fib*, and *sort*.

$x' = x \oplus_1 y = x + 1$, and an input change operation $\oplus_2$ to the function *sort* in Figure 1 may be defined by $x' = x \oplus_2 y = cons(y, x)$.

We use an asymptotic cost model for measuring time complexity and write $t(f(v_1, ..., v_n))$ to denote the asymptotic time of computing $f(v_1, ..., v_n)$. Thus, assuming all primitive functions take constant time, it suffices to consider only the values of function applications as candidate intermediate results to be cached. Of course, caching intermediate results takes extra space, which reflects the well-known trade-off between space and speed. Our primary goal is to improve the asymptotic running time of the incremental computation. Our secondary goal is to save space by maintaining only intermediate results useful for achieving the primary goal.

Given a program f_0 and an input change operation $\oplus$, we can use the approach in Liu and Teitelbaum [1995] to derive a program f_0', an incremental version of f_0 under $\oplus$, such that, if $f_0(x) = r$, then whenever $f_0(x \oplus y)$ returns a value, $f_0'(x, y, r)$ returns the same value and is asymptotically at least as fast.[1] For example, for the function *foo* in Figure 1 and input change operation $x \oplus_1 y = x + 1$, the function *foo'* in Figure 2 is derived. Unfortunately, computing $foo'(x, r)$ is not much faster than computing $foo(x + 1)$ from scratch.

Often, $f_0(x \oplus y)$ can be computed faster by caching and using, in addition to the return value of $f_0(x)$, the intermediate results computed in $f_0(x)$. For instance, in the *foo* example above, the value of $foo(x-1) + foo(x-2)$, which could be used in computing $foo'(x, r)$ faster, is also computed by $foo(x)$ but cannot be retrieved from r. Therefore, the problem is to identify, among the possibly large number of intermediate results computed by $f_0(x)$, those that are useful for computing $f_0(x \oplus y)$, cache these useful intermediate results, use them in computing $f_0(x \oplus y)$, and maintain their corresponding values for computation after a further input change. Using the method in this article, we can obtain the functions $\widehat{foo}$ and $\widehat{foo}'$ in Figure 2 that cache the values of $foo(x-1) + foo(x-2)$ and $foo(x-1)$, use them in computing $foo(x+1)$, and maintain their corresponding values. Computing

[1]While $f_0(x)$ abbreviates $f_0(x_1, ..., x_n)$, and $f_0(x \oplus y)$ abbreviates $f_0(\langle x_1, ..., x_n \rangle \oplus \langle y_1, ..., y_m \rangle)$, $f_0'(x, y, r)$ abbreviates $f_0'(x_1, ..., x_n, y_1, ..., y_m, r)$. Note that some of the parameters of f_0' may be dead and eliminated [Liu and Teitelbaum 1995].

$foo'(x,r)$ = **if** $x \leq 1$ **then** 1 **else if** $x = 2$ **then** 3 **else** $r + foo(x-1) + foo(x-2)$	If $foo(x) = r$, then $foo'(x,r) = foo(x{+}1)$. For x of length n, $foo'(x,r)$ takes time $O(3^n)$; $foo(x{+}1)$ takes time $O(3^n)$.
$\widehat{foo}(x)$ = **if** $x \leq 2$ **then** $< 1 >$ **else let** $v_1 = \widehat{boo}(x)$ **in** $< 1st(v_1){+}foo(x-3),\ v_1 >$	$foo(x) = 1st(\widehat{foo}(x))$. For x of length n, $\widehat{foo}(x)$ takes time $O(3^n)$; $foo(x)$ takes time $O(3^n)$.
$\widehat{boo}(x)$ = **let** $v_1 = foo(x-1)$ **in** $< v_1 {+} foo(x-2),\ < v_1 >>$	
$\widehat{foo}'(x,\hat{r})$ = **if** $x \leq 1$ **then** $< 1 >$ **else if** $x = 2$ **then** $< 3, < 2, < 1 >>>$ **else** $< 1st(\hat{r}){+}1st(2nd(\hat{r}))$, $< 1st(\hat{r}){+}1st(2nd(2nd(\hat{r})))$, $< 1st(\hat{r}) >>>$	If $\widehat{foo}(x) = \hat{r}$, then $\widehat{foo}'(x,\hat{r}) = \widehat{foo}(x{+}1)$. For x of length n, $\widehat{foo}'(x,\hat{r})$ takes time $O(1)$; $\widehat{foo}(x{+}1)$ takes time $O(3^n)$.

Fig. 2. Resulting function definitions of foo', $\widehat{foo}$, and $\widehat{foo}'$.

Table I. Notation

Function	Return Value	Denoted as	Incremental Function
f_0	original value	r	f_0'
$\bar{f}_0$	all intermediate results	$\bar{r}$	$\bar{f}_0'$
$\hat{f}_0$	useful intermediate results	$\hat{r}$	$\hat{f}_0'$

$\widehat{foo}'(x,\hat{r})$ takes only $O(1)$ time. We use this example as a running example.

Notation. We use $<>$ to construct tuples that bundle intermediate results with the original return value of a function. We use selectors $1st$, $2nd$, $3rd$, ... to select the first, second, third, ... elements of such a tuple. We use x to denote the previous input to f_0; r, the cached result of $f_0(x)$; y, the input change parameter; x', the new input $x \oplus y$; and f_0', an incremental version of f_0 under $\oplus$. We use $\bar{f}_0$ to denote an extended function that returns all intermediate results of f_0; $\bar{r}$, the cached result of $\bar{f}_0(x)$; and $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$. Similarly, we use $\hat{f}_0$ to denote a pruned function that returns only the useful intermediate results; $\hat{r}$, the cached result of $\hat{f}_0(x)$; and $\hat{f}_0'$, a function that incrementally maintains only the useful intermediate results (including the original return value). Table I summarizes the notation.

3. OVERVIEW OF THE APPROACH

Suppose we know that certain intermediate results in $f_0(x)$ are useful in computing $f_0(x \oplus y)$ faster. Then we can extend f_0 to a program $\hat{f}_0$ that returns these intermediate results. Then, we can derive an incremental version $\hat{f}_0'$ of $\hat{f}_0$ under $\oplus$ such that, if $\hat{f}_0(x) = \hat{r}$, then whenever $\hat{f}_0(x \oplus y)$ returns a value, $\hat{f}_0'(x, y, \hat{r})$ returns the same value [Liu and Teitelbaum 1995]. This says that $\hat{f}_0'$ can use these intermediate results of $f_0(x)$ and incrementally maintain the corresponding intermediate results of $f_0(x \oplus y)$. Thus, the hard issue left is how to identify, among the possibly large number of intermediate results of $f_0(x)$, those that are useful for computing $f_0(x \oplus y)$. This section proposes two methods, *selective caching* and *cache-and-prune*, but we favor the cache-and-prune method and thus explore it in

the sections after.

Selective Caching. To identify useful intermediate results, a relatively straightforward method is to mimic the derivation approach in Liu and Teitelbaum [1995] to identify subcomputations of $f_0(x \oplus y)$ that are also in $f_0(x)$ but whose values cannot be retrieved from the cached result r of $f_0(x)$. Then, we can transform $f_0(x)$ to return these values and cache them. Such a selective caching method involves function unfolding, algebraic simplification, and so on, as does the derivation approach in Liu and Teitelbaum [1995]. Transformations involving algebraic simplifications are expensive compared with transformations based on static analysis.

Moreover, suppose computing $f_0(x \oplus y)$ needs intermediate result $g(x)$ of $f_0(x)$; then we also need to maintain the value of $g(x \oplus y)$ to support incremental computation after a further input change, i.e., letting $\hat{f_0}^{(1)} = \langle f_0, g \rangle$, we need to compute $\hat{f_0}^{(1)}(x \oplus y)$ incrementally using the cached result $\hat{r}^{(1)}$ of $\hat{f_0}^{(1)}(x)$. However, computing $g(x \oplus y)$ incrementally may introduce the need to cache other intermediate results of $f_0(x)$, i.e., there may be other intermediate results of $f_0(x)$, and thus of $\hat{f_0}^{(1)}(x)$, that can be used to compute $g(x \oplus y)$, and thus $\hat{f_0}^{(1)}(x \oplus y)$, faster but that cannot be retrieved even from $\hat{r}^{(1)}$. To find these other intermediate results, the selective caching method needs to be applied again to the extended program $\hat{f_0}^{(1)}$ and operation $\oplus$.

This process may repeat until we obtain a program $\hat{f_0}^{(i)}$ such that all intermediate results of $\hat{f_0}^{(i)}(x)$ that can be used in computing $\hat{f_0}^{(i)}(x \oplus y)$ can be retrieved from the cached result $\hat{r}^{(i)}$ of $\hat{f_0}^{(i)}(x)$. Intuitively, there exists an upper bound of such $\hat{f_0}^{(i)}$'s, namely, a program that returns all intermediate results of f_0. Thus, we can arrange this iteration to always terminate, by stopping the iteration when necessary and using this upper bound. However, the number of iterations depends on f_0, $\oplus$, and the techniques used to ensure termination. Also, each iteration is expensive. So, we propose instead a simple three-stage method called cache-and-prune.

Cache-and-Prune. The cache-and-prune method consists of three stages. While a static dependency analysis may be iterated a number of times in Stage III, the expensive derivation in Liu and Teitelbaum [1995] is performed only once in Stage II.

Stage I constructs a program $\bar{f_0}$, an extended version of f_0, such that $\bar{f_0}(x)$ returns the values of all function calls made in computing $f_0(x)$. Basically, $\bar{f_0}(x)$ returns a nested tuple containing both the intermediate results and the value of $f_0(x)$, such that

$$1st(\bar{f_0}(x)) = f_0(x) \quad \text{and} \quad t(\bar{f_0}(x)) \leq t(f_0(x)). \tag{2}$$

Stage II derives a program $\bar{f_0}'$, an incremental version of $\bar{f_0}$ under $\oplus$, using the approach in Liu and Teitelbaum [1995], such that if $\bar{f_0}(x) = \bar{r}$ and $\bar{f_0}(x \oplus y) = \bar{r}'$, then

$$\bar{f_0}'(x, y, \bar{r}) = \bar{r}' \quad \text{and} \quad t(\bar{f_0}'(x, y, \bar{r})) \leq t(\bar{f_0}(x \oplus y)), \tag{3}$$

and thus, together with (2), we have

$$1st(\bar{f_0}'(x, y, \bar{r})) = 1st(\bar{f_0}(x \oplus y)) = f_0(x \oplus y). \tag{4}$$

Stage III produces a program $\hat{f_0}$, a pruned version of $\bar{f_0}$, such that $\hat{f_0}(x)$ returns

$\Pi(\bar{r})$, where $\bar{r}$ is the return value of $\bar{f}_0(x)$, and Π is a projection that selects the first and other components of $\bar{r}$ on which $1st(\bar{f}_0'(x,y,\bar{r}))$ *transitively* depends. The dependency is transitive in the sense that if $1st(\bar{f}_0'(x,y,\bar{r}))$ depends on $\Pi_1(\bar{r})$, and $\Pi_1(\bar{f}_0'(x,y,\bar{r}))$ depends on $\Pi_2(\bar{r})$, then $1st(\bar{f}_0'(x,y,\bar{r}))$ depends also on $\Pi_2(\bar{r})$. This transitivity is caused by the need to *maintain* intermediate results corresponding to those that are *used* for computing $1st(\bar{f}_0'(x,y,\bar{r}))$. In other words, this stage eliminates those intermediate results cached in $\bar{r}$ that are not transitively needed in incrementally computing $1st(\bar{f}_0'(x,y,\bar{r}))$, the value of $f_0(x \oplus y)$. In particular, if $f_0(x) = r$, then

$$1st(\hat{f}_0(x)) = r \quad \text{and} \quad t(\hat{f}_0(x)) \leq t(f_0(x)). \tag{5}$$

Additionally, we obtain a program $\hat{f}_0'$, a pruned version of $\bar{f}_0'$, such that if $\bar{f}_0'(x,y,\bar{r})$ returns $\bar{r}'$, then $\hat{f}_0'(x,y,\hat{r})$, where $\hat{r}$ is $\Pi(\bar{r})$, returns $\Pi(\bar{r}')$. This pruning is possible because $\Pi(\bar{r}')$ depends only on $\Pi(\bar{r})$, as can be easily shown using the transitivity above. With the relationship between $\hat{f}_0$ and $\bar{f}_0$, together with (2) and (3), we can prove that if $\hat{f}_0(x) = \hat{r}$ and $f_0(x \oplus y) = r'$, then

$$\hat{f}_0'(x,y,\hat{r}) = \hat{f}_0(x \oplus y) \quad \text{and} \quad t(\hat{f}_0'(x,y,\hat{r})) \leq t(f_0(x \oplus y)), \tag{6}$$

and thus, together with (5), we have

$$1st(\hat{f}_0'(x,y,\hat{r})) = 1st(\hat{f}_0(x \oplus y)) = r'. \tag{7}$$

Thus, $\hat{f}_0'(x,y,\hat{r})$ incrementally computes the desired output and the corresponding intermediate results and is asymptotically at least as fast as computing the desired output from scratch.

Putting (5), (6), and (7) together, if $f_0(x) = r$, then

$$1st(\hat{f}_0(x)) = r \quad \text{and} \quad t(\hat{f}_0(x)) \leq t(f_0(x)), \tag{8}$$

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

$$1st(\hat{f}_0'(x,y,\hat{r})) = r', \quad \hat{f}_0'(x,y,\hat{r}) = \hat{f}_0(x \oplus y), \quad \text{and} \quad t(\hat{f}_0'(x,y,\hat{r})) \leq t(f_0(x \oplus y)), \tag{9}$$

i.e., the programs $\hat{f}_0$ and $\hat{f}_0'$ preserve the semantics and compute asymptotically at least as fast as f_0. Note, however, that $\hat{f}_0(x)$ may terminate more often than $f_0(x)$, and $\hat{f}_0'(x,y,\hat{r})$ may terminate more often than $f_0(x \oplus y)$.

4. STAGE I: CACHING ALL INTERMEDIATE RESULTS

Stage I transforms program f_0 into a program $\bar{f}_0$ that embeds all intermediate results of f_0 in the return value of $\bar{f}_0$. It performs a straightforward extension transformation followed by administrative simplifications.

4.1 Extension

We first perform a local, structure-preserving transformation called *extension*. For each function definition $f(v_1, ..., v_n) = e$, we construct a function definition

$$\bar{f}(v_1, ..., v_n) = \mathcal{E}xt[\![e]\!] \tag{10}$$

where $\mathcal{E}xt[\![e]\!]$ extends an expression e to return a nested tuple that contains the values of all function calls made in computing e. The transformation considers

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

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

subexpressions of e in applicative and left-to-right order, introduces bindings that name the results of function calls, builds up tuples of these values together with the values of the original subexpressions, and passes these values from subcomputations to enclosing computations.

The definition of $\mathcal{E}xt$ is given in Figure 3. We assume that each introduced binding uses a fresh variable name. For a constructed tuple $<>$, $1st$ returns the first element, which is the original return value, and rst returns a tuple of the remaining elements, which are the corresponding intermediate results. We use an infix operation @ to concatenate two tuples. For transforming a conditional expression, the transformation $\mathcal{P}ad[\![e]\!]$ generates a tuple of $\_$'s of length equal to the number of function applications in e, where $\_$ is a dummy constant that just occupies a spot. The length of the tuple generated by $\mathcal{P}ad[\![e]\!]$ can easily be determined by static inspection of e. The use of $\mathcal{P}ad$ ensures that each possible intermediate results appears in a fixed position independent of the value of the Boolean expression.

Essentially, $\bar{f}(v_1,...,v_n)$ and $f(v_1,...,v_n)$ perform the same computation, and thus they take the same asymptotic time. In particular, they have the same termination behavior, and if they terminate, then

$$1st(\bar{f}(v_1,...,v_n)) = f(v_1,...,v_n). \tag{11}$$

For functions *foo* and *boo* of Figure 1, after the extension transformation, we

obtain the functions $\overline{foo}_1$ and $\overline{boo}_1$:

$$
\begin{array}{l}
\overline{foo}_1(x) = \textbf{let } v_1 = \textbf{let } v_{11} = <x> \textbf{ in let } v_{12} = <2> \textbf{ in} \\
\qquad\qquad <1st(v_{11}) \leq 1st(v_{12})> @\, rst(v_{11}) @\, rst(v_{12}) \quad \textbf{in} \\
\qquad \textbf{if } 1st(v_1) \textbf{ then let } v_2 = <1> \textbf{ in} \\
\qquad\qquad\qquad <1st(v_2)> @\, rst(v_1) @\, rst(v_2) @ <\_, \_> \\
\qquad\qquad \textbf{else let } v_3 = \textbf{let } v_{31} = \textbf{let } v_{311} = <x> \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } u_1 = \overline{boo}_1(1st(v_{311})) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad <1st(u_1), u_1> @\, rst(v_{311}) \quad \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{let } v_{32} = \textbf{let } v_{321} = \textbf{let } v_{3211} = <x> \textbf{ in let } v_{3212} = <3> \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad <1st(v_{3211}) - 1st(v_{3212})> @ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad rst(v_{3211}) @\, rst(v_{3212}) \quad \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } u_2 = \overline{foo}_1(1st(v_{321})) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad <1st(u_2), u_2> @\, rst(v_{321}) \quad \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad <1st(v_{31}) + 1st(v_{32})> @\, rst(v_{31}) @\, rst(v_{32}) \quad \textbf{in} \\
\qquad\qquad\qquad <1st(v_3)> @\, rst(v_1) @ <> @\, rst(v_3) \\
\overline{boo}_1(x) = \textbf{let } v_1 = \textbf{let } v_{11} = \textbf{let } v_{111} = <x> \textbf{ in let } v_{112} = <1> \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad <1st(v_{111}) - 1st(v_{112})> @\, rst(v_{111}) @\, rst(v_{112}) \quad \textbf{in} \\
\qquad\qquad\qquad \textbf{let } u_1 = \overline{foo}_1(1st(v_{11})) \textbf{ in} \\
\qquad\qquad\qquad <1st(u_1), u_1> @\, rst(v_{11}) \quad \textbf{in} \\
\qquad\qquad \textbf{let } v_2 = \textbf{let } v_{21} = \textbf{let } v_{211} = <x> \textbf{ in let } v_{212} = <2> \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad <1st(v_{211}) - 1st(v_{212})> @\, rst(v_{211}) @\, rst(v_{212}) \quad \textbf{in} \\
\qquad\qquad\qquad \textbf{let } u_2 = \overline{foo}_1(1st(v_{21})) \textbf{ in} \\
\qquad\qquad\qquad <1st(u_2), u_2> @\, rst(v_{21}) \quad \textbf{in} \\
\qquad\qquad <1st(v_1) + 1st(v_2)> @\, rst(v_1) @\, rst(v_2)
\end{array}
\tag{12}
$$

The transformation $\mathcal{E}xt$ is local and structure-preserving. However, it may introduce unnecessary bindings for values of expressions other than function applications, leave many tuple operations for passing intermediate results unsimplified, and place bindings at undesirable positions, such as within binding definitions. The result is complicated code and reduced readability.

4.2 Administrative Simplification

Administrative simplifications are performed using a *cleaning* transformation to clean up the programs obtained by the extension transformation. For each function definition $f(v_1, ..., v_n) = e$ obtained from the extension transformation, we obtain a function definition

$$f(v_1, ..., v_n) = \mathcal{C}lean[\![e]\!]\, \emptyset \tag{13}$$

where $\mathcal{C}lean[\![e]\!]\, I$ cleans up an expression e using the information set I at e, i.e., it examines subexpressions in applicative and left-to-right order, collects information sets at subexpressions, simplifies tuple operations for passing intermediate results, unwinds binding expressions that become unnecessary as a result of simplifying their subexpressions, and lifts bindings out of enclosing expressions, when possible, to enhance readability.

Here, an information set at e, denoted $I_{[e]}$, is a set of equations collected from the bindings introduced in the context of e. For example, if some $f(v_1, ..., v_n)$ is defined to be e, and e is **let** $v_1 = e_1$ **in let** $v_2 = e_2$ **in** e_3, where v_1 and v_2 are introduced by the extension transformation, then $I_{[e]} = \emptyset$ and $I_{[e_3]} = \{v_1 \leftrightarrow e_1, v_2 \leftrightarrow e_2\}$.

$\mathcal{C}lean$ uses a function $\mathcal{S}imp_{\mathcal{C}lean}$ to perform basic simplifications on tuple operations and introduced bindings, as summarized in Figure 4. Given an expression e

$\mathcal{S}imp_{\mathcal{C}lean}[\![e_1 @ e_2]\!]\ I$	$= <e_{11}, ..., e_{1n_1}, e_{21}, ..., e_{2n_2}>$	if $e_1 \leftrightarrow <e_{11}, ..., e_{1n_1}> \in I$ and $e_2 \leftrightarrow <e_{21}, ..., e_{2n_2}> \in I$
	$= e_1 @ e_2$	otherwise
$\mathcal{S}imp_{\mathcal{C}lean}[\![1st(e)]\!]\ I$	$= e_1$	if $e \leftrightarrow <e_1, e_2, ..., e_n> \in I$
	$= 1st(e)$	otherwise
$\mathcal{S}imp_{\mathcal{C}lean}[\![rst(e)]\!]\ I$	$= <e_2, ..., e_n>$	if $e \leftrightarrow <e_1, e_2, ..., e_n> \in I$
	$= rst(e)$	otherwise
$\mathcal{S}imp_{\mathcal{C}lean}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ I$	$= e_2[e_1/v]$	if v is introduced by $\mathcal{E}xt$ and occurs at most once in e_2
	$=$ **let** $v = e_1$ **in** e_2	otherwise

Fig. 4. Definition of $\mathcal{S}imp_{\mathcal{C}lean}$.

and an information set I, $\mathcal{S}imp_{\mathcal{C}lean}$ simplifies e based on the equations in I.

$\mathcal{C}lean$ uses a function $\mathcal{S}ubl_{\mathcal{C}lean}$ to apply basic simplifications recursively to subexpressions and lift bindings out of enclosing expressions, as defined in Figure 5. The presentation of $\mathcal{S}ubl_{\mathcal{C}lean}$ is simplified by omitting detailed control structures that sequence it through the subexpressions. A subexpression is *reduced* if and only if it is the result of having already applied $\mathcal{C}lean$ to the subexpression at that position. For an expression **let** $v = e_1$ **in** e_2 where e_1 is not itself a binding expression, if e_1 is a conditional, then, for further simplifying its two branches, $\mathcal{S}ubl_{\mathcal{C}lean}$ lifts the condition out; otherwise, if v is introduced by the extension transformation, $\mathcal{S}ubl_{\mathcal{C}lean}$ cleans e_2 with the assumption that v equals e_1 added to the information set.

Finally, we define the function $\mathcal{C}lean$ as in (14). If an expression e has subexpressions, then $\mathcal{C}lean$ calls $\mathcal{S}ubl_{\mathcal{C}lean}$ to recursively clean them. Then $\mathcal{C}lean$ calls $\mathcal{S}imp_{\mathcal{C}lean}$ to simplify the top-level expression.

$$\begin{aligned} \mathcal{C}lean[\![e]\!]I &= e'' \\ &\text{where } e'' = \mathcal{S}imp_{\mathcal{C}lean}[\![e']\!]I \\ &\qquad e' = \begin{cases} \mathcal{S}ubl_{\mathcal{C}lean}[\![e]\!]I & \text{if } e \text{ is not } v \\ e & \text{otherwise} \end{cases} \end{aligned} \tag{14}$$

Each cleaned function $\bar{f}$ still satisfies the properties stated at (11).

For the functions $\overline{foo}_1$ and $\overline{boo}_1$ in (12), after the cleaning transformation, we obtain the functions $\overline{foo}$ and $\overline{boo}$ below:

$$\begin{aligned} \overline{foo}(x) = \ & \textbf{if } x \leq 2 \textbf{ then } <1, \_, \_> \\ & \textbf{else let } u_1 = \overline{boo}(x) \textbf{ in} \\ & \quad \textbf{let } u_2 = \overline{foo}(x-3) \textbf{ in} \\ & \quad <1st(u_1)+1st(u_2),\ u_1,\ u_2> \end{aligned} \qquad \begin{aligned} \overline{boo}(x) = \ & \textbf{let } u_1 = \overline{foo}(x-1) \textbf{ in} \\ & \textbf{let } u_2 = \overline{foo}(x-2) \textbf{ in} \\ & <1st(u_1)+1st(u_2),\ u_1,\ u_2> \end{aligned} \tag{15}$$

They are much simpler and are easier for the subsequent stages to process.

4.3 Complexity and Optimization

$\mathcal{E}xt$ and $\mathcal{C}lean$ transform each function definition separately. If we assume that programs are composed of function definitions of bounded sizes, which is especially true for large programs, then both $\mathcal{E}xt$ and $\mathcal{C}lean$ are linear in the size of the entire program.

$\mathcal{S}ubl_{\mathcal{C}lean}[\![g(e_1,...,e_n)]\!]\ I$ where g is c, p, or f

$= \mathcal{S}ubl_{\mathcal{C}lean}[\![g(e_1,...,e_{i-1},e'_i,e_{i+1},...,e_n)]\!]\ I$ if $e_1,...,e_{i-1}$ are reduced, not **let**, but e_i is not reduced
where $e'_i = \mathcal{C}lean[\![e_i]\!]\ I$

$= \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v{=}e_{i1} \textbf{ in } g(e_1,...,e_{i-1},e_{i2},e_{i+1},...,e_n)]\!]\ I$ if $e_1,...,e_{i-1}$ are reduced, not **let**, e_i is reduced, but e_i is **let** $v{=}e_{i1}$ **in** e_{i2}

$= g(e_1,...,e_n)$ otherwise

$\mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I$

$= \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e'_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I$ if e_1 is not reduced
where $e'_1 = \mathcal{C}lean[\![e_1]\!]\ I$

$= \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v{=}e_{11} \textbf{ in if } e_{12} \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I$ if e_1 is reduced, but e_1 is **let** $v{=}e_{11}$ **in** e_{12}

$= \textbf{if } e_1 \textbf{ then } \mathcal{C}lean[\![e_2]\!]\ I \textbf{ else } \mathcal{C}lean[\![e_3]\!]\ I$ otherwise

$\mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v{=}e_1 \textbf{ in } e_2]\!]\ I$

$= \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v{=}e'_1 \textbf{ in } e_2]\!]\ I$ if e_1 is not reduced
where $e'_1 = \mathcal{C}lean[\![e_1]\!]\ I$

$= \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v_1{=}e_{11} \textbf{ in let } v{=}e_{12} \textbf{ in } e_2]\!]\ I$ if e_1 is reduced, but e_1 is **let** $v_1{=}e_{11}$ **in** e_{12}

$= \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_{11} \textbf{ then let } v{=}e_{12} \textbf{ in } e_2 \textbf{ else let } v{=}e_{13} \textbf{ in } e_2]\!]\ I$ if e_1 is reduced, but e_1 is **if** e_{11} **then** e_{12} **else** e_{13}

$= \textbf{let } v{=}e_1 \textbf{ in } \mathcal{C}lean[\![e_2]\!]\ I'$ otherwise
where $I' = \begin{cases} I \cup \{v \mapsto e_1\} & \text{if } v \text{ is introduced} \\ I & \text{otherwise} \end{cases}$

Fig. 5. Definition of $\mathcal{S}ubl_{\mathcal{C}lean}$.

For a function definition of size s, the typical behaviors of $\mathcal{E}xt$ and $\mathcal{C}lean$ are also linear, and this is observed in our prototype implementation. The worst-case time of $\mathcal{E}xt$ is quadratic, and it occurs only if there are $\Theta(s)$ conditional branches that contain function applications. This is caused by the padding that is done when extending conditionals, e.g., **if** $null(x)$ **then** $f_1(x)$ **else if** $null(cdr(x))$ **then** $f_2(x)$ **else** $f_3(x)$ is extended to **if** $null(x)$ **then** $<f_1(x),\_,\_>$ **else if** $null(cdr(x))$ **then** $<\_,f_2(x),\_>$ **else** $<\_,\_,f_3(x)>$. The worst-case time of $\mathcal{C}lean$ is exponential, but it occurs only if there are $\Theta(s)$ conditionals in the binding of a binding expression. This is caused by the lifting of the conditions out of the binding, resulting in duplication of the body, e.g., **let** $v = ($**if** $null(x)$ **then** e_1 **else if** $null(cdr(x))$ **then** e_2 **else** $e_3)$ **in** e_4 is cleaned to **if** $null(x)$ **then let** $v{=}e_1$ **in** e_4 **else if** $null(cdr(x))$ **then let** $v{=}e_2$ **in** e_4 **else let** $v{=}e_3$ **in** e_4. In any case, the time complexities of $\mathcal{E}xt$ and $\mathcal{C}lean$ are linear in the size of their outputs.

An obvious optimization can be incorporated into the extension transformation: it can avoid introducing bindings for subexpressions that do not contain function applications. The optimized extension transformation introduces fewer tuple operations for passing intermediate results and fewer bindings to be unwound or lifted, leaving less work for the administrative simplifications. Although this optimization does not reduce the worst-case asymptotic time or space complexities of the exten-

sion transformation, it does provide significant benefits in practice. Most programs contain many more subexpressions than function calls, so this optimization can significantly reduce the number of bindings and tuple operations introduced by the extension transformation, and thus speed up the cleaning transformation as well.

5. STAGE II: INCREMENTALIZATION

Stage II derives a program $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$. To derive an incremental version f_0' of f_0 under $\oplus$, for any f_0, the basic idea is to identify subcomputations in the expanded $f_0(x \oplus y)$ whose values can be retrieved from the cached result r of $f_0(x)$, replace them by corresponding retrievals, and capture the resulting way of computing $f_0(x \oplus y)$ in an incremental version $f_0'(x, y, r)$. Such a derivation method is given in Liu and Teitelbaum [1995]. This section summarizes the method and discusses interactions with the transformations for caching and pruning.

5.1 Basic Incrementalization Method

The basic observation is that all subcomputations in $f_0(x \oplus y)$ depend on x, y, or both, and we can try to separate out those depending only on x and replace them with retrievals from the cached result r of $f_0(x)$.

To separate out subcomputations depending only on x, we unfold $f_0(x \oplus y)$ and simplify subcomputations. For example, $x + 1 - 1$ simplifies to x, $cdr(cons(y, x))$ simplifies to x, and (**let** $v = e_1$ **in** e_2) simplifies to $e_2[v := e_1]$ if v occurs at most once in e_2 or if e_1 takes constant time. Simplifications can exploit equalities in the context of a subcomputation. For example, in **let** $v = e_1$ **in if** $v = u$ **then** e_2 **else** v, expression e_2 has the context $v = e_1$ and $v = u$. If e_2 is **if** $u = e_1$ **then** e_3 **else** e_4, then it can be simplified to e_3.

To replace subcomputations with retrievals from the cache result r of $f_0(x)$, we can replace occurrences of $f_0(x)$ by r. We can also exploit r further. In particular, we can use components of r. For example, if $f_0(x) = cons(g(x), h(x))$, then we can replace $g(x)$ by $car(r)$ and replace $h(x)$ by $cdr(r)$. Also, we can use r conditionally. For example, if $f_0(x) =$ **if** e_1 **then** e_2 **else** e_3, then $e_2 = r$ when e_1 is true, and $e_3 = r$ when e_1 is false; thus we can replace e_2 by r in **if** e_1 **then** $cons(y, e_2)$ **else** y, and we can replace e_3 by **if** e_1 **then** e_3 **else** r if $t(e_1) \leq t(e_3)$. The latter yields separate cases. If an expression e is a Boolean-valued function application, we can use replacements that are valid for both values of e. For example, for Boolean-valued function application $g(e_1)$, if $g(f_0(x)) = true$ when $g(e_1)$ is true, and $g(f_0(x)) = false$ when $g(e_1)$ is false, then we can replace $g(e_1)$ by $g(r)$.

The overall derivation examines subcomputations recursively, in applicative and left-to-right order, and applies simplifications and replacements described above. If the resulting subcomputation is a function application $f(e_1, ..., e_n)$ that depends on x, then replace it with an incremental version: if $f(e_1, ..., e_n)$, together with its context and available cached results, forms an instance of an already introduced incremental function f', then replace $f(e_1, ..., e_n)$ using f' with appropriate instantiation; otherwise, introduce an appropriate incremental version to compute $f(e_1, ..., e_n)$, with an additional argument that is the cached result of some $f(e_1', ..., e_n')$ that best

matches $f(e_1, ..., e_n)$,[2] and continue to apply unfolding, simplification, and replacement on the new function. After we finish transforming a new function, we perform dead-code elimination. This is needed since if we retrieve the value of a subcomputation from a cached result, then subcomputations used only in computing this value become dead. The overall derivation starts with $\bar{f}_0(x \oplus y)$.

This method is systematic and parameterized by modules for equality reasoning, time analysis, and strategies used in introducing incremental functions. It can be made automatic or semiautomatic depending on the power one expects from each module. For example, equality reasoning can exploit algebraic properties of only constructors or exploit arithmetic properties as well; time analysis can conservatively compare times of primitives or handle all recursive functions; and strategies for introducing incremental functions can allow one argument for a cached result and one incremental version for each given function or allow multiple arguments for cached results and multiple incremental versions. If the conservative option is chosen for each module, then the overall derivation is fully automatic and always terminates; alternatively, if the more ambitious option is chose, the derivation is semiautomatic. The method in Liu and Teitelbaum [1995] limits each incremental function to use one argument for a cached result, but it leaves other choices as parameters of the method.

Even though this method uses transformations like unfoldings and simplifications, it is tailored to achieve the special goal of using a previously computed result. It uses a specific sequence of special transformations and is therefore systematic. In particular, it introduces incremental functions that correspond to nonincremental functions and uses cached results as extra arguments, and it performs unfoldings, simplifications, and replacements using cached results. The effect cannot be achieved by general optimization strategies, such as the general fold/unfold transformation strategies [Burstall and Darlington 1977], in such an automatable way. In particular, the method does not introduce eureka definitions, which can be new functions with arbitrary arguments, nor does it perform arbitrary folding, which may cause a resulting program not to terminate even when the original program terminates.

The incrementalization method is also powerful. Even if restricted to use automatic methods for equality reasoning, time analysis, and introduction of incremental functions, it can derive efficient incremental programs that could not be automatically derived before. Nevertheless, as with all program optimization techniques, how well the resulting programs perform depends on how the original programs are written.

Consider the function $\overline{foo}$ in (15) that caches all intermediate results of *foo*. We derive an incremental version of $\overline{foo}$ under $\oplus_1$, following the approach in Liu and Teitelbaum [1995], as follows. We transform $\overline{foo}(x+1)$, with $\overline{foo}(x) = \bar{r}$:

[2]The particular algorithm used in Liu and Teitelbaum [1995] is omitted here.

1. unfold $\overline{foo}(x+1)$ and then $\overline{boo}(x+1)$ and simplify primitives

$$
\begin{array}{l}
= \ \textbf{if } x \le 1 \textbf{ then } <1, \_, \_> \\
\quad \textbf{else let } u_{11} = \overline{foo}(x) \textbf{ in} \\
\qquad \textbf{let } u_{12} = \overline{foo}(x-1) \textbf{ in} \\
\qquad \textbf{let } u_1 = <1st(u_{11})+1st(u_{12}), \\
\qquad\qquad u_{11},\ u_{12}> \textbf{ in} \\
\qquad \textbf{let } u_2 = \overline{foo}(x-2) \textbf{ in} \\
\qquad <1st(u_1)+1st(u_2),\ u_1,\ u_2>
\end{array}
$$

2. separate cases and replace calls to $\overline{foo}$ by retrievals

$$
\begin{array}{l}
= \ \textbf{if } x \le 1 \textbf{ then } <1, \_, \_> \\
\quad \textbf{else if } x = 2 \textbf{ then} \\
\qquad <3, <2, <1, \_, \_>, <1, \_, \_>>, <1, \_, \_>> \\
\quad \textbf{else let } u_{11} = \bar{r} \textbf{ in} \\
\qquad \textbf{let } u_{12} = 2nd(2nd(\bar{r})) \textbf{ in} \\
\qquad \textbf{let } u_1 = <1st(u_{11})+1st(u_{12}),\ u_{11},\ u_{12}> \textbf{ in} \\
\qquad \textbf{let } u_2 = 3rd(2nd(\bar{r})) \textbf{ in} \\
\qquad <1st(u_1)+1st(u_2),\ u_1,\ u_2>
\end{array}
$$

Since each binding takes constant time to compute, the bindings can be unfolded by simplification, yielding the following:

$$
\begin{array}{l}
\overline{foo}'_1(x, \bar{r}) = \ \textbf{if } x \le 1 \textbf{ then } <1, \_, \_> \\
\qquad\qquad \textbf{else if } x = 2 \textbf{ then } <3,\ <2, <1, \_, \_>, <1, \_, \_>>,\ <1, \_, \_>> \\
\qquad\qquad \textbf{else } <1st(\bar{r})+1st(2nd(2nd(\bar{r}))) + 1st(3nd(2nd(\bar{r}))), \\
\qquad\qquad\qquad <1st(\bar{r})+1st(2nd(2nd(\bar{r}))),\ \bar{r},\ 2nd(2nd(\bar{r}))>,\ 3rd(2nd(\bar{r}))>
\end{array}
\tag{16}
$$

If the equality reasoning used is powerful enough, it can infer that the first component in the third branch equals $1st(\bar{r}) + \big(1st(2nd(2nd(\bar{r}))) + 1st(3nd(2nd(\bar{r})))\big)$, which equals $1st(\bar{r}) + 1st(2nd(\bar{r}))$, where the first equality is by associativity of $+$, and the second is by definition of $\overline{boo}$. We obtain the incremental function $\overline{foo}'$ below such that, if $\overline{foo}(x) = \bar{r}$, then $\overline{foo}'(x, \bar{r}) = \overline{foo}(x+1)$:

$$
\begin{array}{l}
\overline{foo}'(x, \bar{r}) = \ \textbf{if } x \le 1 \textbf{ then } <1, \_, \_> \\
\qquad\qquad \textbf{else if } x = 2 \textbf{ then } <3,\ <2, <1, \_, \_>, <1, \_, \_>>,\ <1, \_, \_>> \\
\qquad\qquad \textbf{else } <1st(\bar{r})+1st(2nd(\bar{r})), \\
\qquad\qquad\qquad <1st(\bar{r})+1st(2nd(2nd(\bar{r}))),\ \bar{r},\ 2nd(2nd(\bar{r}))>,\ 3rd(2nd(\bar{r}))>
\end{array}
\tag{17}
$$

Clearly, both $\overline{foo}'_1(x, \bar{r})$ and $\overline{foo}'(x, \bar{r})$ compute $\overline{foo}(x+1)$ in only $O(1)$ time. However, they both take $O(3^n)$ space, since $\bar{r}$ does.

5.2 Interaction with Cache and Prune

Incrementalizing a program f_0 under $\oplus$ yields an incremental program f'_0 that computes $f_0(x \oplus y)$ efficiently using the value of $f_0(x)$. With a Stage I that caches all intermediate results of f_0 to obtain $\bar{f}_0$, and a Stage III that prunes unused intermediate results in $\bar{f}'_0$ to obtain $\hat{f}'_0$, our method yields a program that computes $f_0(x \oplus y)$ efficiently using all useful intermediate results of $f_0(x)$.

As shown in Section 3, if an incrementalization method ensures that an incremental program always computes at least as fast as its nonincremental counterpart, then the cache-and-prune method guarantees that $\hat{f}'_0(x, y, \hat{r})$ always computes at least as fast as $f_0(x \oplus y)$. However, it is important to also guarantee that $\hat{f}'_0(x, y, \hat{r})$ always computes at least as fast as $f'_0(x, y, r)$, i.e., the cost saved by using intermediate results in $\hat{r}$ is greater than or equal to the cost of maintaining them in $\hat{f}'_0(x, y, \hat{r})$. This ensures that the cache-and-prune method is at least as good as the incrementalization method alone for arbitrary programs.

First, we show how caching and maintaining an intermediate result may cause $\bar{f}'_0(x, y, \bar{r})$ to be slower than $f'_0(x, y, r)$. Let $g(x)$ be an intermediate result of $f_0(x)$, and let $\bar{f}_0$ extend f_0 to return this intermediate result also. Suppose that $f'_0(x, y, r)$ computes $f_0(x \oplus y)$ given $r = f_0(x)$ and that $\bar{f}'_0(x, y, \bar{r})$ computes $\bar{f}_0(x \oplus y)$ given

$\bar{r} = \bar{f}_0(x)$. It is possible that $f_0'(x, y, r)$ does not compute $g(x \oplus y)$ even though $f_0(x \oplus y)$ does, e.g., if $g(x \oplus y)$ is found to be dead. However, $\bar{f}_0'(x, y, \bar{r})$ must compute $g(x \oplus y)$, since $\bar{f}_0'(x, y, \bar{r})$ computes $\bar{f}_0(x \oplus y)$, whose return value contains $g(x \oplus y)$. So, dead-code elimination performed during the incrementalization cannot eliminate the computation of $g(x \oplus y)$ from $\bar{f}_0'(x, y, \bar{r})$. If $g(x \oplus y)$ cannot be computed incrementally as efficiently as $f_0(x \oplus y)$ can, then $\bar{f}_0'(x, y, \bar{r})$ can be slower than $f_0'(x, y, r)$.

Next, we show how pruning can help compensate for this. Suppose $\bar{f}_0(x)$ returns some intermediate result $g_1(x)$, but $g_1(x \oplus y)$ cannot be computed incrementally as efficiently as $f_0(x \oplus y)$. If computing $1st(\bar{f}_0'(x, y, \bar{r}))$, i.e., $f_0(x \oplus y)$, does not use the value $g_1(x)$ in $\bar{r}$, then pruning eliminates $g_1(x)$ in $\bar{r}$ and $g_1(x \oplus y)$ in $\bar{f}_0'(x, y, \bar{r})$, and we may obtain an incremental program $\hat{f}_0'(x, y, \hat{r})$ that is as fast as $f_0'(x, y, r)$. However, this does not completely solve the problem. The whole idea of caching intermediate results is for the incrementalization to use them, by replacing subcomputations with retrievals from these cached results. So, the problem is to determine when it is worth using an intermediate result.

Based on the above interactions with the transformations for caching and pruning, the rest of this section studies three requirements on the incrementalization method to guarantee that $\hat{f}_0'(x, y, \hat{r})$ is at least as fast as $f_0'(x, y, r)$. These requirements are based on how an intermediate result $g(x)$ might be used in the incremental computation, causing $g(x \oplus y)$ to be maintained. The interactions shed light on the fundamental issues that arise with the cache-and-prune method.

The first requirement is that if a subcomputation $h(x)$ in the transformed $\bar{f}_0(x \oplus y)$ can be replaced by a retrieval using either $1st(\bar{r})$ or an intermediate result $g(x)$ other than $1st(\bar{r})$, then $1st(\bar{r})$ is used instead of $g(x)$.[3] The goal of this is to leave $g(x)$ unused, so it can be pruned out in Stage III, saving both time and space. In fact, Stage I can be enhanced to avoid caching intermediate results that are embedded in the original return value, using an embedding analysis [Liu et al. 1996]; to keep Stage I simple, we did not present this here. A related requirement, although not needed for ensuring that $\hat{f}_0'(x, y, \hat{r})$ is as fast as $f_0'(x, y, r)$, helps achieve our secondary goal of minimizing space complexity: if there are multiple intermediate results from which $h(x)$ can be retrieved, then a retrieval from an intermediate result that is already used is preferable.

The second requirement is that a subcomputation $h(x)$ in the transformed $\bar{f}_0(x \oplus y)$ is replaced by a retrieval using an intermediate result $g(x)$ other than $1st(\bar{r})$ only if $h(x)$ takes at least as much time as $g(x)$.[4] Together with the first requirement, we know that $h(x)$ cannot be replaced by using $1st(r)$ but can be replaced by using $g(x)$. This means that $f_0'(x, y, r)$ incurs the cost of computing $h(x)$ if and only if $\hat{f}_0'(x, y, \hat{r})$ does not incur the cost of computing $h(x)$ but incurs the cost of maintaining $g(x \oplus y)$. We show below how to ensure that the time spent in maintaining $g(x \oplus y)$ does not surpass the time saved by not computing $h(x)$.

The third requirement is that a subcomputation $h(x)$ in the transformed $\bar{f}_0(x \oplus y)$

[3] In other words, we require that if $h(x)$ is replaced by a retrieval from r in deriving f_0', then it is replaced by a retrieval from $1st(\bar{r})$ in deriving $\bar{f}_0'$.

[4] To compare the times of $h(x)$ and $g(x)$, an automated analysis could be used. A simpler method is to check whether $h(x)$ can be simplified to yield $g(x)$, since simplifications never increase cost.

is replaced in some branch by a retrieval using an intermediate result $g(x)$ other than $1st(\bar{r})$ only if, in every other branch where $g(x \oplus y)$ is maintained, $g(x \oplus y)$ takes constant time.[5] This is because intermediate result $g(x)$ may be used conditionally in some branch, but then $g(x \oplus y)$ needs to be maintained, possibly in other branches. We want to guarantee that in these other branches, the time of computing $g(x \oplus y)$ can be ignored asymptotically.

With the above requirements on the incrementalization method, if (a) the size of y is bounded, (b) when the size of y is bounded, the time of computing $x \oplus y$ is bounded, and (c) g is at most *linear-power exponential* time, i.e., g is either polynomial time or exponential time but with linear exponent, then we have

$$t(\hat{f}_0'(x, y, \hat{r})) \leq t(f_0'(x, y, r)). \tag{18}$$

It is easy to see that the three conditions are true for all practical and feasible incremental applications, and therefore we assume they are satisfied. To prove (18), we notice that

$$\begin{array}{rll}
t(\hat{f}_0'(x, y, \hat{r})) & \leq t(f_0'(x, y, r)) + t(g(x \oplus y)) & \text{by definition of } \hat{f}_0' \text{ and derivation} \\
& \leq t(f_0'(x, y, r)) + t(g(x)) & \text{by conditions on } y, \oplus, \text{ and } g \\
& \leq t(f_0'(x, y, r)) + t(h(x)) & \text{by requirements above} \\
& \leq t(f_0'(x, y, r)) + t(f_0'(x, y, r)) & \text{by } h \text{ being a subcomputation of } f_0' \\
& \leq t(f_0'(x, y, r)) & \text{by definition of } t.
\end{array}$$

$\hat{f}_0'$ might be a constant factor slower than f_0', if there is an intermediate result $g(x \oplus y)$ of $f_0(x \oplus y)$ that became dead in $f_0'(x, y, r)$ due to retrieval from r but is live in $\hat{f}_0'(x, y, \hat{r})$ due to retrieval of $g(x)$ from $\hat{r}$. This does not cause $\hat{f}_0'(x, y, \hat{r})$ to be slower than $f_0(x \oplus y)$ because $g(x \oplus y)$ is an intermediate result of $f_0(x \oplus y)$. Since incrementalization can ensure that $f_0'(x, y, r)$ is absolutely no slower than $f_0(x \oplus y)$, it can ensure that $\bar{f}_0'(x, y, \bar{r})$ is absolutely no slower than $\bar{f}_0(x \oplus y)$. The only reason that $\bar{f}_0'(x, y, \bar{r})$ might be slower than $f_0(x \oplus y)$ is the cost of tuple operations introduced in $\bar{f}_0(x \oplus y)$ and thus in $\bar{f}_0'(x, y, \bar{r})$. This cost is usually small and in the worst case increases the running time by a factor equal to the size of the largest function definition in the original program f_0. Furthermore, pruning $\bar{f}_0'(x, y, \bar{r})$ to produce $\hat{f}_0'(x, y, \hat{r})$ largely eliminates this cost because unused intermediate results and the corresponding tuple operations are eliminated.

Note that the three conditions on y, $\oplus$, and g are not needed if intermediate results are maintained lazily during sequences of calls to $\hat{f}_0'$. This laziness ensures that an intermediate result in an iteration is maintained only if it is actually needed in the next iteration. So, if $\hat{f}_0'(x, y, \hat{r})$ uses an intermediate result $g(x)$ to replace a computation $h(x)$ that takes at least as much time as $g(x)$, then $f_0'(x, y, r)$ has the cost of computing $h(x)$, and thus $\hat{f}_0'(x, y, \hat{r})$ is at least as fast as $f_0'(x, y, r)$.

6. STAGE III: PRUNING

Stage III prunes programs $\bar{f}_0$ and $\bar{f}_0'$ to programs $\hat{f}_0$ and $\hat{f}_0'$, respectively, so that $\hat{f}_0$ returns only the useful intermediate results and so that $\hat{f}_0'$ maintains only these useful intermediate results. To achieve this, we analyze program $\bar{f}_0'$ to determine

[5]Implementing a conservative and reasonably accurate test for this is straightforward. Note that this typically holds in branches corresponding to the base cases of a recursive function definition.

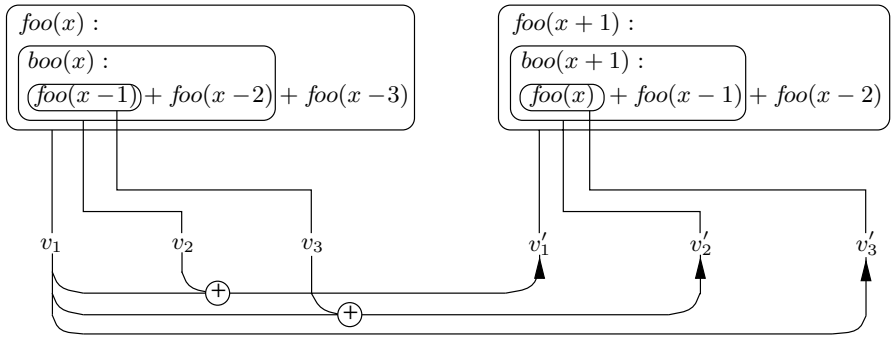

Fig. 6. Transitive dependencies.

the components of $\bar{r}$, i.e., the value of $\bar{f}_0(x)$, on which $1st(\bar{f}_0'(x, y, \bar{r}))$, i.e., the value of $f_0(x \oplus y)$, transitively depends, and we prune $\bar{f}_0$ and $\bar{f}_0'$.

6.1 Transitive Dependency for Maintaining Intermediate Results

Since the goal is to compute $f_0(x \oplus y)$, we start with the first component of $\bar{f}_0'(x, y, \bar{r})$ and determine the components of $\bar{r}$ on which this value depends; these components may include components other than the first one of $\bar{r}$. To support incremental computation after a further input change, we need to maintain these other components of $\bar{f}_0'(x, y, \bar{r})$ as well as the first component, but they may depend on still other components of $\bar{r}$, forming a kind of *transitive dependency*.

Figure 6 illustrates the transitive dependencies for the program $\overline{foo}'$. By definitions of *foo* and *boo* and associativity of +, as used by the derivation of $\overline{foo}'$ in (17), we have

$$\begin{aligned} foo(x+1) = boo(x+1) + foo(x-2) &= \big(foo(x)+foo(x-1)\big) + foo(x-2) \\ &= foo(x) + \big(foo(x-1)+foo(x-2)\big) = foo(x) + boo(x). \end{aligned}$$

Thus, to compute the value v_1' of $foo(x+1)$, $\overline{foo}'$ sums the value v_1 of $foo(x)$ and the (intermediate) value v_2 of $boo(x)$. To maintain the corresponding (intermediate) value v_2' of $boo(x+1)$, $\overline{foo}'$ sums the value v_1 of $foo(x)$ and the (intermediate) value v_3 of $foo(x-1)$. To maintain the corresponding (intermediate) value v_3' of $foo(x)$, $\overline{foo}'$ just uses the value v_1 of $foo(x)$. Therefore, to summarize, the value v_1' of $foo(x+1)$ transitively depends on the components corresponding to intermediate results v_1, v_2, and v_3, which are maintained as $v_1' = v_1 + v_2$, $v_2' = v_1 + v_3$, and $v_3' = v_1$, respectively. Other intermediate results do not need to be computed or maintained; they can be pruned out. Similarly, we could draw a dependency graph for the program $\overline{foo}_1'$.

To summarize, we need to compute the closure of the transitive dependencies for maintaining all useful intermediate results.

6.2 Dependency Analysis Using Projections

We first describe our use of domain projections [Gunter 1992; Scott 1982] to represent components of the tuple values constructed in Stage I and manipulated by Stage II. Then, we give a backward dependency analysis that determines which com-

ponents of $\bar{r}$ are needed for computing certain designated components of $\bar{f}_0'(x, y, \bar{r})$. Finally, we present an algorithm that computes the closure of the transitive dependencies for maintaining intermediate results.

Projections. Our domain D of interest contains $\perp$, indicating a computation diverges, values d returned by functions in the original program f_0, and constructed tuples $< d_1, ..., d_n >$, where each d_i is (recursively) an element of D other than $\perp$. The length of a constructed tuple is statically bounded, but the depth of tuple nesting may not be statically bounded. Intuitively, any components of a constructed tuple value can be replaced by the dummy constant $\_$, introduced in Stage I, if we do not care about the values of those components. Even when a subcomputation involves $\_$, the result of the parent computation need not be $\_$. For any value d in domain D, we define $\perp \sqsubseteq d$. For two values d_1 and d_2 other than $\perp$ in D, we define $d_1 \sqsubseteq d_2$ if and only if

$$d_1 = \_, \quad d_1 = d_2, \quad \text{or}$$
$$d_1 = \ < d_{11}, ..., d_{1n} >, \ d_2 = \ < d_{21}, ..., d_{2n} >, \text{ and } d_{1i} \sqsubseteq d_{2i} \text{ for } i = 1..n.$$

A projection over the domain D is a function $\Pi : D \to D$ such that $\Pi(\Pi(d)) = \Pi(d)$ and $\Pi(d) \sqsubseteq d$ for all $d \in D$. Three important projections are *ID*, *ABS*, and *BOT*. *ID* is the identity function: $ID(d) = d$. *ABS* is the absence function: $ABS(d) = \_$ for all $d \neq \perp$. *BOT* is the bottom function: $BOT(d) = \perp$.

A nonbottom projection Π of interest here can be represented as a set of selection functions, each of which is a sequence of elements of $\{1^{st}, 2^{nd}, ...\}$. The null sequence is denoted ϵ. Intuitively, if Π contains a sequence $i_k{}^{th} i_{k-1}{}^{th} ... i_1{}^{th}$, then the i_kth element of the i_{k-1}th element of the $\cdots$ of the i_1th element of Π's argument is selected, and if Π contains ϵ, then all components of Π's argument are selected. A projection Π replaces those components of its argument that are not selected with the constant $\_$. For example $\{1^{st}\}$, $\{1^{st}, 1^{st}2^{nd}\}$, and $\{1^{st}1^{st}2^{nd}, \epsilon\}$ are projections, and

$$\{1^{st}, 1^{st}2^{nd}\}(< d_1, << d_{211}, d_{212} >, d_{22} >>) \ = \ < d_1, << d_{211}, d_{212} >, \_ >> .$$

For convenience of presentation, we use Π^{-i} to denote the set $\{\pi \mid \pi\, i^{th} \in \Pi\}$, i.e., Π^{-i} is the part of Π that considers the ith component. With the set representation, a projection $\Pi = ID$ if and only if $\epsilon \in \Pi$ or $\Pi^{-i} = ID$ for $i = 1..n$ for arguments of Π of length n; a projection $\Pi = ABS$ if and only if $\Pi = \emptyset$. For $\Pi \notin \{ID, ABS\}$, $\Pi(< d_1, ..., d_n >) = < \Pi^{-1}(d_1), ..., \Pi^{-n}(d_n) >$. For any two projections Π_1 and Π_2 other than *BOT*, $\Pi_1 \sqsubseteq \Pi_2$ if and only if

$$\Pi_1 = ABS, \quad \Pi_2 = ID, \quad \text{or}$$
$$\Pi_1{}^{-i} \sqsubseteq \Pi_2{}^{-i} \text{ for } i = 1..n \text{ for arguments of } \Pi_1 \text{ and } \Pi_2 \text{ of length } n.$$

Dependency Analysis. We use a backward dependence analysis to compute which components of $\bar{r}$ are needed for computing certain components of $\bar{f}_0'(x, y, \bar{r})$.

Following the style of Wadler and Hughes [1987], for each function f of n parameters, and each i from 1 to n, we define f^i to be a *dependency transformer* that takes a projection that is applied to the result of f and returns a projection that is *sufficient* to be applied to the ith parameter. The *sufficiency condition* that f^i

$v^v(\Pi)$	$= \Pi$	
$u^v(\Pi)$	$= ABS$	if $u \neq v$
$(<e_1,...,e_n>)^v(\Pi)$	$= e_1^v(\Pi^{-1}) \cup ... \cup e_n^v(\Pi^{-n})$	
$(ith(e))^v(\Pi)$	$= e^v(\{\pi\, i^{th} \mid \pi \in \Pi\})$	
$(g(e_1,...,e_n))^v(\Pi)$	$= e_1^v(ID) \cup ... \cup e_n^v(ID)$	if g is c or p but not $<>$ or ith
$(f(e_1,...,e_n))^v(\Pi)$	$= e_1^v(f^1(\Pi)) \cup ... \cup e_n^v(f^n(\Pi))$	
$(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^v(\Pi)$	$= e_1^v(ID) \cup e_2^v(\Pi) \cup e_3^v(\Pi)$	
$(\textbf{let } u{=}e_1 \textbf{ in } e_2)^v(\Pi)$	$= e_1^v(e_2^u(\Pi)) \cup e_2^v(\Pi)$	assume $u \neq v$ after renaming

Fig. 7. Definition of $e^v(\Pi)$ for $\Pi \neq BOT, ABS$.

must satisfy is if $\Pi_i = f^i(\Pi)$ then

$$\Pi(f(v_1,...,v_i,...,v_n)) \sqsubseteq f(v_1,...,\Pi_i(v_i),...,v_n). \tag{19}$$

Similarly, we define e^v to be a dependency transformer that takes a projection that is applied to e and returns a projection that is sufficient to be applied to every instance of v in e. A similar sufficiency condition must be satisfied: if $\Pi' = e^v(\Pi)$ then

$$\Pi(e) \sqsubseteq e[\Pi'(v)/v]. \tag{20}$$

For a function f whose definition is $f(v_1,...,v_n) = e$, we define $f^i(\Pi) = e^{v_i}(\Pi)$. The definition of e^v may in turn refer to f^i; thus the definitions may be mutually recursive. We define

$$e^v(BOT) = BOT \qquad \text{and} \qquad e^v(ABS) = ABS. \tag{21}$$

For $\Pi \notin \{BOT, ABS\}$, we define $e^v(\Pi)$ in Figure 7. It is easy to show that each rule guarantees sufficient information. Thus, the sufficiency conditions are satisfied by induction.

Let $i_{\bar{r}}$ be the index of $\bar{r}$ in the parameters of $\bar{f_0}'$. With the above definitions, we know that $\bar{f_0}'^{i_{\bar{r}}}(\Pi)$ computes how much of $\bar{r}$ is needed when $\Pi(\bar{f_0}'(x,y,\bar{r}))$ is needed.

To compute $f^i(\Pi)$ for some f^i and $\Pi \notin \{BOT, ABS\}$ (otherwise, we can use (21)), if the definition of f^i does not involve recursion, then we can compute directly using the definition. If the definition of f^i involves recursion, then the argument projections and resulting projections of some dependency transformers may contain selection functions of unbounded depth. To approximate the result, we restrict the selection functions of the projections to be of bounded depth d, namely, if a projection contains a selection function ${i_k}^{th}\,{i_{k-1}}^{th}...{i_1}^{th}$ but $k > d$, then we truncate it to ${i_d}^{th}\,{i_{d-1}}^{th}...{i_1}^{th}$. A simple choice for the depth bound would be 1. A more prudent choice could be the length of the longest simple cycle that contains f in the call graph. This limits the domain of projections to be finite. Now, to solve the recursive definitions of these dependency transformers, we just compute the limits of the ascending chains by starting at $f^i(\Pi) = ABS$ for all f^i and Π and iterating using the definitions. This iteration with the approximated domain of projections always terminates, since when the depth of nesting being examined is bounded, the

ascending chains are finite. Note that the resulting projection of an $f^i(\Pi)$ is valid for all calls to f, including recursive calls.

Computing the Closure of Transitive Dependency. To compute the components Π of $\bar{r}$ on which $1st(\bar{f}_0'(x, y, \bar{r}))$ transitively depends, we start with Π being $\{1^{st}\}$ and compute the smallest projection Π of $\bar{r}$ on which $\Pi(\bar{f}_0'(x, y, \bar{r}))$ depends, i.e., the smallest projection Π such that

$$\{1^{st}\} \sqsubseteq \Pi \quad \text{and} \quad \Pi(\bar{f}_0'(x, y, \bar{r})) \sqsubseteq \bar{f}_0'(x, y, \Pi(\bar{r})). \tag{22}$$

Of course, the projection $\Pi = ID$ is always a solution. But our goal is to make Π as small as possible and thus to avoid as much unnecessary caching as possible.

Since $\bar{f}_0'^{i_{\bar{r}}}(\Pi)$ computes the components of $\bar{r}$ on which $\Pi(\bar{f}_0'(x, y, \bar{r}))$ depends, we define

$$\begin{array}{lcl} \Pi^{(0)} & = & \{1^{st}\} \\ \Pi^{(i+1)} & = & \Pi^{(i)} \ \cup \ \bar{f}_0'^{i_{\bar{r}}}(\Pi^{(i)}) \end{array} \tag{23}$$

and iterate until the sequence converges. Thus, Π is the least projection that satisfies $\{1^{st}\} \sqsubseteq \Pi$ and $\bar{f}_0'^{i_{\bar{r}}}(\Pi) \sqsubseteq \Pi$. We call this projection the *closure projection.* Note that the above computation always terminates, since $\bar{f}_0'^{i_{\bar{r}}}(\Pi^{(i)})$ terminates and returns only sets of selection functions of bounded depth.

The time complexity of the closure computation depends on the required size of the projection domain and the complexity of the dependency analysis. Suppose d is the maximum depth of selection functions we consider, and l is the maximum length of the constructed tuples, which is bounded by the largest number of function applications in a function definition in the program f_0. Then the maximum number c of disjoint components in these projections is at most l^d, which characterizes the maximum size of the projection domain.

We estimate the complexity of the dependency analysis in the simplest manner. Consider a program $\bar{f}_0'$. Let n be the number of function definitions, and let a be the maximum number of parameters in any of these definitions. Then there are at most na dependency transformers. Since an argument projection may contain any subset of the c components, there are at most 2^c argument projections to each transformer. Thus, the number of projections $f^i(\Pi)$ to be computed is at most $na2^c$. Let s_f be the number of function applications in a function definition; then the maximum number of transformers used in a transformer definition is $s_f a$. Being careful, we can recompute each $f^i(\Pi)$ only when some computed projection used by $f^i(\Pi)$ changes, where each can change at most c times. Thus, the total number of computations of $f^i(\Pi)$ using its immediate definition is at most $na^2 2^c c s_f$. Each such computation takes at most $O(sc)$ time, where s is the maximum size of a function definition, i.e., the number of subexpressions in the defining expression, and c reflects the time needed to compute operations, such as union, on two projections. Therefore, the total time is at most $O(na^2 2^c c^2 s s_f)$. This bound includes the computations of all $f^i(\Pi)$. Computing the dependency closure takes at most c additional projection unions, each taking at most $O(c)$ time. Thus, the total time of closure computation is still at most $O(na^2 2^c c^2 s s_f)$.

Although this worst-case time bound is high (exponential in l and doubly exponential in d), the actual complexity is typically much smaller, as seen in the running example and the examples in Section 8. The reason for this is that typical program

structure induces locality in the dependencies, so many of the projections are not needed in computing the closure projection. To exploit this, the projections $f^i(\Pi)$ can be computed on demand during the computation of the closure projection. If we limit the depth of selection functions to be independent of the number of function definitions, then a, c, s, and s_f are all constant factors determined by the size of a function definition. Thus the total time is linear in the number of function definitions, although the constant factors could be very large.

Example. Applying dependency analysis to the function $\overline{foo}'$ in (17), we obtain

$$\begin{aligned}
&\overline{foo}'^2(\Pi)\\
&= (x \le 1)^{\bar r}(ID) \cup (<1,_,_>)^{\bar r}(\Pi) \cup\\
&\quad (x = 2)^{\bar r}(ID) \cup (<3,<2,<1,_,_>,<1,_,_>>,<1,_,_>>)^{\bar r}(\Pi) \cup\\
&\quad (<1st(\bar r)+1st(2nd(\bar r)),\ <1st(\bar r)+1st(2nd(2nd(\bar r))),\ \bar r,\ 2nd(2nd(\bar r))>,\ 3rd(2nd(\bar r))>)^{\bar r}(\Pi)\\
&= (1st(\bar r)+1st(2nd(\bar r)))^{\bar r}(\Pi^{-1}) \cup\\
&\quad (1st(\bar r)+1st(2nd(2nd(\bar r))))^{\bar r}((\Pi^{-2})^{-1}) \cup (\bar r)^{\bar r}((\Pi^{-2})^{-2}) \cup (2nd(2nd(\bar r)))^{\bar r}((\Pi^{-2})^{-3}) \cup\\
&\quad (3rd(2nd(\bar r)))^{\bar r}(\Pi^{-3}).
\end{aligned}$$

For this example, since the definition of $\overline{foo}'^2$ is not recursive, we can compute $\overline{foo}'^2(\Pi)$ for a given Π directly without iteration and approximation. For example,

$$\begin{aligned}
\overline{foo}'^2(\{1^{st}\}) &= \{1^{st}, 1^{st}2^{nd}\}\\
\overline{foo}'^2(\{1^{st}2^{nd}\}) &= \{1^{st}, 1^{st}2^{nd}2^{nd}\}\\
\overline{foo}'^2(\{1^{st}2^{nd}2^{nd}\}) &= \{1^{st}\}
\end{aligned}$$

illustrate the dependencies depicted in Figure 6. An example where the dependency transformer is defined recursively is shown in the merge sort example in Section 8. Computing the projection for the closure of the transitive dependencies, we have

$$\begin{aligned}
\Pi^{(1)} &= \Pi^{(0)} \cup \overline{foo}'^2(\Pi^{(0)}) = \{1^{st}, 1^{st}2^{nd}\}\\
\Pi^{(2)} &= \Pi^{(1)} \cup \overline{foo}'^2(\Pi^{(1)}) = \{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}\\
\Pi^{(3)} &= \Pi^{(2)} \cup \overline{foo}'^2(\Pi^{(2)}) = \{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}.
\end{aligned}$$

Thus, we obtain the projection $\{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}$.

Similarly, applying dependency analysis to the function $\overline{foo}_1'$ in (16), we obtain

$$\begin{aligned}
&\overline{foo}_1'^2(\Pi)\\
&= (1st(\bar r)+1st(2nd(2nd(\bar r))) + 1st(3nd(2nd(\bar r))))^{\bar r}(\Pi^{-1}) \cup\\
&\quad (1st(\bar r)+1st(2nd(2nd(\bar r))))^{\bar r}((\Pi^{-2})^{-1}) \cup (\bar r)^{\bar r}((\Pi^{-2})^{-2}) \cup (2nd(2nd(\bar r)))^{\bar r}((\Pi^{-2})^{-3}) \cup\\
&\quad (3rd(2nd(\bar r)))^{\bar r}(\Pi^{-3}).
\end{aligned}$$

Computing the projection for the closure of the transitive dependencies, we have

$$\begin{aligned}
\Pi^{(1)} &= \Pi^{(0)} \cup \overline{foo}'^2(\Pi^{(0)}) = \{1^{st}, 1^{st}2^{nd}2^{nd}, 1^{st}3^{rd}2^{nd}\}\\
\Pi^{(2)} &= \Pi^{(1)} \cup \overline{foo}'^2(\Pi^{(1)}) = \{1^{st}, 1^{st}2^{nd}2^{nd}, 1^{st}3^{rd}2^{nd}\}.
\end{aligned}$$

We obtain the closure projection $\{1^{st}, 1^{st}2^{nd}2^{nd}, 1^{st}3^{rd}2^{nd}\}$.

6.3 Pruning under the Closure Projection

With the closure projection Π obtained above, we want to prune the extended program $\bar f_0$ to get a program $\hat f_0$ such that $\Pi(\bar f_0(x)) \sqsubseteq \hat f_0(x)$, and prune the incremental program $\bar f_0'$ to get a program $\hat f_0'$ such that $\Pi(\bar f_0'(x, y, \bar r)) \sqsubseteq \hat f_0'(x, y, \Pi(\bar r))$. Of course, setting $\hat f_0$ to be $\bar f_0$ and $\hat f_0'$ to be $\bar f_0'$ would always work, but we want

to make $\hat{f}_0(x)$ as close to $\Pi(\bar{f}_0(x))$, and $\hat{f}_0'(x, y, \Pi(\bar{r}))$ as close to $\Pi(\bar{f}_0'(x, y, \bar{r}))$, as possible, to avoid caching and maintaining unnecessary intermediate results as much as possible.

To do this, for each expression e that defines a function $f(v_1, ...v_n)$, we associate a projection with each subexpression of e, indicating how much of the subexpression is needed assuming Π of $\bar{f}_0$ (or $\bar{f}_0'$) is needed. The definition and computation of the associated projections can be done in a fashion similar to the dependency analysis. For the program $\bar{f}_0'$ and the closure projection Π, the final projection associated with each variable will be the same as computed for the variable using dependency analysis.

After the computation, subexpressions associated with *ID* are left unchanged, and subexpressions associated with *ABS* are replaced by $\_$. If a variable whose value is a constructed tuple is associated with a projection Π other than *ID* or *ABS*, then we construct a tuple with the components selected by Π filled with the corresponding selections and the rest filled with $\_$. For example, if a variable v is associated with a projection $\{1^{st}, 1^{st}2^{nd}\}$, and v represents a tuple of length three whose second component is a tuple of length two, then v is replaced by $<1st(v),\ <1st(2nd(v)), \_>,\ \_>$.

As the result of such replacements, we have $\Pi(\bar{f}_0(x)) \sqsubseteq \hat{f}_0(x)$, but not necessarily $\hat{f}_0(x) = \Pi(\bar{f}_0(x))$ as anticipated in Section 3. Nevertheless, the resulting $\hat{f}_0$ is still good enough to guarantee (8): we can just project $\Pi(\bar{r})$ out of the return value of $\hat{f}_0(x)$. We do have $\hat{f}_0'(x, y, \Pi(\bar{r})) = \Pi(\bar{f}_0'(x, y, \bar{r}))$. Thus, assuming $\hat{r} = \Pi(\bar{r})$, we have (9). We intend to use $\hat{f}_0$ only once to get the initial value, and then use $\hat{f}_0'$ repeatedly to compute all successive values.

For the functions $\overline{foo}$ in (15) and $\overline{foo}'$ in (17), only $\{1^{st}, 1^{st}2^{nd}, 1^{st}\ 2^{nd}\ 2^{nd}\}$ of their return values are needed. After the replacements as above, we obtain the functions $\widehat{foo}_1$ and $\widehat{foo}_1'$ below such that, if $\widehat{foo}_1(x) = \hat{r}_1$, then $\widehat{foo}_1'(x, \hat{r}_1) = \widehat{foo}_1(x+1)$.

$$
\begin{array}{l}
\widehat{foo}_1(x) = \ \textbf{if } x \le 2 \textbf{ then } <1, \_, \_> \\
\qquad\qquad \textbf{else let } u_1 = \widehat{boo}_1(x) \textbf{ in} \\
\qquad\qquad\quad \textbf{let } u_2 = \widehat{foo}_1(x-3) \textbf{ in} \\
\qquad\qquad\quad <1st(u_1)+1st(u_2), \\
\qquad\qquad\qquad <1st(u_1), <1st(2nd(u_1)), \_, \_>, \_>, \\
\qquad\qquad\qquad \_>
\end{array}
\qquad\qquad
\begin{array}{l}
\widehat{boo}_1(x) = \ \textbf{let } u_1 = \widehat{foo}_1(x-1) \textbf{ in} \\
\qquad\qquad\quad \textbf{let } u_2 = \widehat{foo}_1(x-2) \textbf{ in} \\
\qquad\qquad\quad <1st(u_1)+1st(u_2), \\
\qquad\qquad\qquad <1st(u_1), \_, \_>, \\
\qquad\qquad\qquad \_>
\end{array}
\qquad (24)
$$

$$
\begin{array}{l}
\widehat{foo}_1'(x, \hat{r}_1) = \ \textbf{if } x \le 1 \textbf{ then } <1, \_, \_> \\
\qquad\qquad\qquad\quad \textbf{else if } x = 2 \textbf{ then } <3, <2, <1, \_, \_>, <\_, \_, \_>>, <\_, \_, \_>> \\
\qquad\qquad\qquad\quad \textbf{else } <1st(\hat{r}_1)+1st(2nd(\hat{r}_1)), \\
\qquad\qquad\qquad\qquad\quad <1st(\hat{r}_1)+1st(2nd(2nd(\hat{r}_1))), <1st(\hat{r}_1), \_, \_>, \_>, \_>
\end{array}
\qquad (25)
$$

Similarly, for the functions $\overline{foo}$ in (15) and $\overline{foo}_1'$ in (16), only $\{1^{st}, 1^{st}2^{nd}2^{nd}, 1^{st}3^{rd}2^{nd}\}$ of their return values are needed. After replacements, we obtain the functions $\widehat{foo}_{11}$

and $\widehat{foo}_{11}'$:

$$\begin{array}{l}\widehat{foo}_{11}(x) = \textbf{if } x \le 2 \textbf{ then } <1,\_,\_> \\ \qquad\qquad \textbf{else let } u_1 = \widehat{boo}_{11}(x) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } u_2 = \widehat{foo}_{11}(x-3) \textbf{ in} \\ \qquad\qquad\quad <1st(u_1)+1st(u_2), \\ \qquad\qquad\qquad <\_, <1st(2nd(u_1)),\_,\_>, \\ \qquad\qquad\qquad\quad <1st(3rd(u_1)),\_,\_>>, \\ \qquad\qquad\qquad \_>\end{array} \qquad \begin{array}{l}\widehat{boo}_{11}(x) = \textbf{let } u_1 = \widehat{foo}_{11}(x-1) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } u_2 = \widehat{foo}_{11}(x-2) \textbf{ in} \\ \qquad\qquad\quad <1st(u_1)+1st(u_2), \\ \qquad\qquad\qquad <1st(u_1),\_,\_>, \\ \qquad\qquad\qquad <1st(u_2),\_,\_>>\end{array} \tag{26}$$

$$\begin{array}{l}\widehat{foo}_{11}'(x,\hat{r}_{11}) = \textbf{if } x \le 1 \textbf{ then } <1,\_,\_> \\ \qquad\qquad\qquad\quad \textbf{else if } x = 2 \textbf{ then } <3, <\_,<1,\_,\_>,<1,\_,\_>>, <\_,\_,\_>> \\ \qquad\qquad\qquad\quad \textbf{else } <1st(\hat{r}_{11})+1st(2nd(2nd(\hat{r}_{11})))+1st(3rd(2nd(\hat{r}_{11}))), \\ \qquad\qquad\qquad\qquad\quad <\_,\ <1st(\hat{r}_{11}),\_,\_>, <1st(2nd(2nd(\hat{r}_{11}))),\_,\_>>,\ \_>\end{array} \tag{27}$$

A number of simplifications can be applied to the resulting functions. First, unfold a binding expression if a binding variable occurs at most once in the body. This is enabled by replacements using $\_$. Second, combine split components in a constructed tuple whose selected components are not preceded by any $\_$. Such splits are caused by replacements for variables whose values are constructed tuples. Third, lift common selections that use $1st$. This avoids unnecessarily computing a compound value and using only part of it. Fourth, replace occurrences of $1st(\hat{f}(e_1,...,e_n))$ by occurrences of $f(e_1,...,e_n)$. The last three simplifications are needed only on the nonincremental program $\hat{f}_0$.

Furthermore, we can eliminate $\_$ components. We must be careful if, in a tuple, such a component precedes a component that is not $\_$, since the selectors need to be adjusted. In particular, if k of the components preceding a component i are eliminated from a tuple, then we must replace all uses of the selector ith for the tuple with $(i-k)$th. This elimination needs to be done consistently for $\hat{f}_0$ and $\hat{f}_0'$. The change of selectors is needed only in the incremental program $\hat{f}_0'$.

These simplifications and eliminations can be fully automated. For example, the simplification for unfolding binding expressions can be performed straightforwardly based on an occurrence counting analysis [Jones et al. 1993]. These simplifications and eliminations help reduce running time and space, as well as code size, for both $\hat{f}_0$ and $\hat{f}_0'$, and they can also greatly reduce the asymptotic space complexity of $\hat{f}_0$, but they do not affect the asymptotic time and space complexities of the resulting incremental program $\hat{f}_0'$.

For the function $\widehat{foo}_1$ in (24), unfold the binding for u_2, replace $1st(\widehat{foo}_1(x-3))$ by $foo(x-3)$, and combine split components of u_1. For the function $\widehat{boo}_1$ in (24), unfold the binding for u_2, replace $1st(\widehat{foo}_1(x-2))$ by $foo(x-2)$, and lift $1st(u_1)$. We obtain the following:

$$\begin{array}{l}\widehat{foo}_2(x) = \textbf{if } x \le 2 \textbf{ then } <1,\_,\_> \\ \qquad\qquad \textbf{else let } u_1 = \widehat{boo}_2(x) \textbf{ in} \\ \qquad\qquad\quad <1st(u_1)+foo(x-3), u_1, \_>\end{array} \qquad \begin{array}{l}\widehat{boo}_2(x) = \textbf{let } v_1 = foo(x-1) \textbf{ in} \\ \qquad\qquad\quad <v_1+foo(x-2), <v_1,\_,\_>, \_>\end{array} \tag{28}$$

Function $\widehat{foo}_1'$ remains the same. Finally, we eliminate unnecessary $\_$ components in the functions $\widehat{foo}_2$ and $\widehat{boo}_2$ in (28) and $\widehat{foo}_1'$ in (25) and obtain the functions $\widehat{foo}$, $\widehat{boo}$, and $\widehat{foo}'$ as given in Figure 2.

Similarly, we apply simplifications to the functions $\widehat{foo}_{11}$ in (26) and $\widehat{foo}'_{11}$ in (27), and we replace selectors $2nd(2nd(\hat{r}_{11})$ and $3rd(2nd(\hat{r}_{11})$ by $1st(2nd(\hat{r}_{11})$ and $2nd(2nd(\hat{r}_{11})$, respectively. We obtain the following:

$$\begin{array}{l} \widehat{foo}_{12}(x) = \textbf{if } x \le 2 \textbf{ then } <1> \\ \qquad\qquad \textbf{else let } u_1 = \widehat{boo}_{12}(x) \textbf{ in} \\ \qquad\qquad\quad <1st(u_1)+foo(x-3), \\ \qquad\qquad\qquad <<1st(2nd(u_1))>, \\ \qquad\qquad\qquad\quad <1st(3rd(u_1))>>> \end{array} \qquad \begin{array}{l} \widehat{boo}_{12}(x) = \textbf{let } u'_1 = foo(x-1) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } u'_2 = foo(x-2) \textbf{ in} \\ \qquad\qquad\quad <u'_1+u'_2, <u'_1>, <u'_2>> \end{array} \tag{29}$$

$$\begin{array}{l} \widehat{foo}'_{12}(x, \hat{r}_{12}) = \textbf{if } x \le 1 \textbf{ then } <1> \\ \qquad\qquad\qquad\quad \textbf{else if } x = 2 \textbf{ then } <3, <<1>, <1>>> \\ \qquad\qquad\qquad\quad \textbf{else } <1st(\hat{r}_{12})+1st(1st(2nd(\hat{r}_{12})))+1st(2nd(2nd(\hat{r}_{12}))), \\ \qquad\qquad\qquad\qquad\quad <<1st(\hat{r}_{12})>, <1st(1st(2nd(\hat{r}_{12})))>>> \end{array} \tag{30}$$

In both cases, pruning leaves us with resulting programs that use space for only two additional values. Thus, in both cases, incremental computation takes not only constant time but also constant space.

7. DISCUSSION

We have obtained an extended program $\hat{f}_0$, which caches appropriate intermediate results, and a corresponding program $\hat{f}'_0$ that incrementally maintains these intermediate results. The programs $\hat{f}_0$ and $\hat{f}'_0$ preserve the semantics of computations and compute asymptotically at least as fast as computing from scratch or computing using only the old return value but not intermediate results.

7.1 Incrementalization and Cache-and-Prune: Power and Limitation

The cache-and-prune method consists of three relatively independent stages and thus is modular. Each stage fulfills its goal with a desired property. Stage I gives us maximality by providing all the intermediate results possibly used by Stage II. Stage II uses these intermediate results for the exclusive purpose of incrementalization. Stage III gives us a kind of minimality by preserving only the intermediate results actually used by Stage II. So, the overall approach has a kind of optimality with respect to the incrementalization method of Stage II. Overall, the cache-and-prune method is a powerful framework, in the following three respects.

First, even using simple automatic methods for the incrementalization in Stage II, the cache-and-prune method can identify intermediate results that can be used and maintained efficiently in an incremental program $\hat{f}'$, as demonstrated by many examples, some given in Section 8. This coincides with the intuition that, in many repeated computations, incremental computation can use previously computed results in simple and therefore automatable ways. The principle of cache-and-prune is general and applies also to other language features, e.g., imperative programs that use arrays [Liu and Stoller 1998].

Second, the idea of cache-and-prune is not limited to using intermediate results. If some other information might be useful for the incremental computation, we can cache them as well as intermediate results, incrementalize this further extended program, and prune the resulting programs. A class of such *auxiliary information* can actually be found by mimicking the incrementalization method [Liu et al. 1996]. For comparison, Table II lists the basic ideas for incrementalization using only

Table II. Comparing Incrementalization, Selective Caching, and Discovering Auxiliary Information

Method	Identify Subcomputations in Transformed $f_0(x \oplus y)$
inc	whose values can be retrieved from the cached results r of $f_0(x)$
cache	that are also subcomputations in $f_0(x)$ but whose values cannot be retrieved from the cached results r of $f_0(x)$
aux	whose values cannot be retrieved from the cached results $\bar{r}$ of $\bar{f_0}(x)$

the return value (inc), for selectively caching intermediate results (cache), and for discovering a class of auxiliary information (aux):

Third, the cache-and-prune method can be used for general program optimization via caching. We can incrementalize a loop body under the loop increment to obtain an incrementalized loop body, and we can incrementalize a recursive function under an appropriate input change operation to form an efficient new recursion. Besides using an incremental program for these repeated computations, we can also use a slight variant of the incrementalization method to replace subcomputations with retrievals from results computed earlier in the same iteration.

Three caveats also come with the cache-and-prune method, showing its limitations. First, although the method allows all intermediate results to be cached and used, the exploitation of such results has limitations. For example, there may be values that are computed only in certain branches of $f_0(x)$ but are useful under different conditions in $f_0(x \oplus y)$; this is the case for the path sequence problem, the knapsack problem, etc. In such cases, the cache-and-prune method as described cannot effectively incrementalize $f_0(x \oplus y)$, unless the values that are needed under those different conditions are treated as a kind of auxiliary information [Liu et al. 1996].

Second, to use the cache-and-prune method for general program optimization, an important issue is to determine appropriate f_0's and $\oplus$'s. For many programs, including all **while** and **for** loops, this is relatively straightforward [Liu 1997], but a systematic method is lacking for general recursions. As a rule of thumb, when f_0 is used repeatedly in a computation, and $\oplus$ specifies small changes in the program state, incrementalization can result in asymptotic speedup; otherwise, it can give at most constant factor speedup.

Third, as with the incrementalization method described in Section 5, how well the cache-and-prune method works depends on how the original programs are written. Also, certain efficient computations can only be coded using language features not treated in this article. For example, some image-processing algorithms require arrays. The principle of cache-and-prune is general, but the method must be extended to analyze and transform programs written using those other features [Liu 1997; Liu and Stoller 1998].

7.2 Transformation and Analysis Techniques

The idea of caching all intermediate results followed by incrementalization can be regarded as a realization of the reduction from Kleene's course-of-values recursion to primitive recursion [Kleene 1952] (private communication, A. Nerode, 1995). We summarize techniques that are relevant to the program analyses and transformations used for caching and pruning.

The transformation $\mathcal{E}xt$ in Stage I is similar to the construction of call-by-value

complete recursive programs by Cartwright [1984]. However, a call-by-value computation sequence returned by such a program is a flat list of all intermediate results, while our extended program returns a nested tuple, a tree structure that mirrors the hierarchy of function calls. The transformations in Stage I also mimic the CPS transformations in some respects [Lawall and Danvy 1993; Plotkin 1975]: sequencing subexpressions, naming intermediate results, passing the collected information, and performing administrative reductions on the resulting program. However, our transformations are simpler than the CPS transformations, since the collected intermediate results are passed directly to return values, rather than to continuation functions.

The backward dependency analysis and pruning transformations in Stage III use domain projections to specify sufficient information. Wadler and Hughes [1987] use projections for strictness analysis. Their analysis is also backward but seeks necessary rather than sufficient information. Launchbury [1989] uses projections for binding-time analysis. It is a forward analysis and is proved equivalent to strictness analysis [Launchbury 1991]. Mogensen [1989] also uses projections for binding-time analysis, based on a restricted class of regular tree grammars.

Several analyses are in the same spirit as ours. The necessity interpretation by Jones and Le Métayer [1989] uses necessity patterns that correspond to projections. Necessity patterns specify only heads and tails of list values. The absence analysis by Hughes [1990] uses the name context in place of projection. It handles only a finite domain of list contexts where every head context and every tail context is the same. The analysis for backward slicing by Reps and Turnidge [1996] uses projections based on regular tree grammars. Their grammars specify only atoms or nested pairs. We use projections that specify specific components of tuple values and thus can provide more accurate information, but our methods for making the domains finite are crude.

Recently, Liu [1998] uses projections based on general regular tree grammars to specify specific components of tuples and gives more natural methods for making the domains finite. Since the dependency analysis and pruning transformations simply eliminate dead components and related computations on compound values, they are useful for general program optimizations. For example, in many functional programs, compound values are created only to be taken apart somewhere else, and perhaps some of the components are used. It is desirable to avoid constructing and passing the unnecessary components [Traub 1986]. A summary of applications is given in Liu [1998].

There are analyses and transformations not yet mentioned that we believe could be incorporated in our framework. For example, type analysis is useful for many program manipulations, including incrementalization. The transformations in Stages I and III apply to both typed and untyped languages and could easily be augmented with corresponding manipulations needed for types. Also, in Stage III, further manipulation with projections could enable more simplifications, such as *component lifting*. For example, if we lift the single element in the second of the second component of $\widehat{foo}$ and $\widehat{foo}'$ in Figure 2, and change $1st(2nd(2nd(\widehat{r})))$ to $2nd(2nd(\widehat{r}))$, we obtain $\widehat{foo}$ in Figure 2, and $\widehat{boo}$ and $\widehat{foo}'$ below:

$$\widehat{boo}(x) = \textbf{let } v_1 = foo(x-1) \textbf{ in } < v_1 + foo(x-2),\ v_1 >$$

$$\widehat{foo}'(x, \widehat{r}) = \textbf{if } x \leq 1 \textbf{ then } < 1 > \textbf{ else if } x = 2 \textbf{ then } < 3,\ < 2, 1 >> \textbf{ else } < 1st(\widehat{r}) + 1st(2nd(\widehat{r})),\ < 1st(\widehat{r}) + 2nd(2nd(\widehat{r})),\ 1st(\widehat{r}) >>$$

7.3 Cost Model and Time-Space Trade-Off

The basic motivation for caching is to trade space for speed, and our primary concern is to reduce the asymptotic running time of the incremental computation. Thus, we cache values of all function applications that are useful for the incremental computation, assuming other program constructs take constant time. For example, if the value of $f(x) + g(x)$ is needed in the incremental program, then we cache the values of $f(x)$ and $g(x)$ and compute the sum from the two cached values. However, we do not assume that space is free. Each of the three stages makes an effort to reduce space consumption without adversely affecting asymptotic running time.

One could be more mindful of economizing cache space by not caching values of function applications unless they are absolutely needed. For example, if the value of $f(x) + g(x)$ is needed in the incremental program, but neither $f(x)$ nor $g(x)$ is needed separately, then we can cache just the value of $f(x) + g(x)$; coincidentally, this also improves the speed of this example by a slight constant amount.

On the other hand, we could be more mindful of constant speedup, regardless of additional space consumption, by caching the results of all subcomputations, not just function applications. For example, we could cache the values of $f(x)$, $g(x)$, and $f(x) + g(x)$ for their respective uses in the incremental program, thus saving the time to compute the sum but consuming the space to store the sum.

Other choices affecting the time-space trade-off may also be required by applications, e.g., achieving the least running time possible given a fixed-size cache space. For some applications, we may need to consider the number of times a given value is needed. Ideally, we would have a cost model for time and a cost model for space and then decide what to cache depending on the trade-off between time and space required by the application. There are standard constructions for mechanical time analysis [Rosendahl 1989; Wegbreit 1975], which can be used by our method [Liu and Teitelbaum 1995], though further study is needed. However, if we want to reduce space at possible sacrifice of speed, we must analyze the trade-off between time and space, which is a problem open for study.

8. EXAMPLES

This section gives more examples: incremental merge sort and the Fibonacci function, as well as sketches of seven other examples that are taken from attribute evaluation, combinatorics, string processing, graph algorithms, image processing, etc.

8.1 Merge Sort

Consider a sorting program and an input change operation $x \oplus y = cons(y, x)$. If the program does an insertion sort, i.e., inserts the first element into the correct position of the recursively sorted other elements, or a selection sort, i.e., selects the least element and recursively sorting the rest, then incrementalization yields

an insertion program that inserts y into the sorted result r of $sort(x)$ [Liu and Teitelbaum 1995]. While using insertion sort or selection sort on $cons(y, x)$ takes quadratic time, inserting y into r takes linear time.

Consider the merge sort program *sort* in Figure 1 and input change operation $x \oplus y = cons(y, x)$. If we are given $sort(cons(y, x)) = merge(cons(y, nil), sort(x))$, then we can directly obtain $sort'(y, r) = merge(cons(y, nil), r)$, which also takes linear time and essentially performs an insertion. However, discovering this non-trivial property of *merge* and *sort* is difficult, and proving it requires induction. Below we show that caching enables us to obtain a linear-time incremental merge sort without the need of discovering and proving this property.

—Stage I. Cache all intermediate results of *sort* using $\mathcal{E}xt$ and $\mathcal{C}lean$ and obtain the following:

```
sort(x)
= if null(x) then
        < nil, _, _, _, _, _ >
  else if null(cdr(x)) then
        < x, _, _, _, _, _ >
  else let v11 = odd(x) in
       let u1 = sort(1st(v11)) in
       let v21 = even(x) in
       let u2 = sort(1st(v21)) in
       let v = merge(1st(u1), 1st(u2)) in
       < 1st(v), v11, u1, v21, u2, v >
```

```
odd(x)  = if null(x) then < nil, _ >
          else let v1 = even(cdr(x)) in
               < cons(car(x), 1st(v1)), v1 >
even(x) = if null(x) then < nil, _ >
          else let v1 = odd(cdr(x)) in
               < 1st(v1), v1 >
merge(x, y)
        = if null(x) then < y, _, _ >
          else if null(y) then < x, _, _ >
          else if car(x) ≤ car(y) then
               let v1 = merge(cdr(x), y) in
               < cons(car(x), 1st(v1)), v1, _ >
          else let v2 = merge(x, cdr(y)) in
               < cons(car(y), 1st(v2)), _, v2 >
```

(31)

—Stage II. Derive an incremental version of $\overline{sort}$ under $\oplus$ following the approach in Liu and Teitelbaum [1995], i.e., transform $\overline{sort}(cons(y, x))$, with $\overline{sort}(x) = \bar{r}$:

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

```
= if null(x) then
        < cons(y, nil), _, _, _, _, _ >
  else let v1 = even(x) in
       let v11 = <cons(y, 1st(v1)),
                  v1> in
       let u1 = sort(1st(v11)) in
       let v2 = odd(x) in
       let v21 = <1st(v2), v2 > in
       let u2 = sort(1st(v21)) in
       let v = merge(1st(u1),
                     1st(u2)) in
       < 1st(v), v11, u1, v21, u2, v >
```

2. separate cases, replace applications of $\overline{sort}$

```
= if null(1st(r̄)) then
        < cons(y, nil), _, _, _, _, _ >
  else if null(cdr(1st(r̄))) then
       let v11 = <cons(y, nil), <nil, <nil>>> in
       let v21 = <1st(r̄), <1st(r̄), <nil>>> in
       let v = merge(cons(y, nil), 1st(r̄)) in
       <1st(v), v11, <cons(y, nil)>, v21, <1st(r̄)>,
        v >
  else let v1 = 4th(r̄) in
       let v11 = <cons(y, 1st(v1)), v1 > in
       let u1 = sort'(y, 1st(v1), 5th(r̄)) in
       let v2 = 2nd(r̄) in
       let v21 = <1st(v2), v2 > in
       let u2 = 3rd(r̄) in
       let v = merge(1st(u1), 1st(u2)) in
       <1st(v), v11, u1, v21, u2, v >
```

We obtain the function $\overline{sort}'$ below such that, if $\overline{sort}(x) = \bar{r}$, then $\overline{sort}'(y, \bar{r}) = \overline{sort}(cons(y, x))$. Note that the parameter x was dead and has been eliminated

from the definition of $\overline{sort}'$.

$$
\begin{array}{l}
\overline{sort}'(y,\bar{r}) = \textbf{ if } null(1st(\bar{r})) \textbf{ then} \\
\qquad\qquad < cons(y,nil), \_, \_, \_, \_, \_ > \\
\qquad\quad \textbf{else if } null(cdr(1st(\bar{r}))) \textbf{ then} \\
\qquad\qquad \textbf{let } v_{11} = < cons(y,nil),\ < nil, < nil >>> \textbf{ in} \\
\qquad\qquad \textbf{let } v_{21} = < 1st(\bar{r}),\ < 1st(\bar{r}), < nil >>> \textbf{ in} \\
\qquad\qquad \textbf{let } v = \overline{merge}(cons(y,nil), 1st(\bar{r})) \textbf{ in} \\
\qquad\qquad < 1st(v),\ v_{11}, < cons(y,nil) >,\ v_{21}, < 1st(\bar{r}) >,\ v > \\
\qquad\quad \textbf{else let } v_1 = 4th(\bar{r}) \textbf{ in} \\
\qquad\qquad \textbf{let } v_{11} = < cons(y, 1st(v_1)), v_1 > \textbf{ in} \\
\qquad\qquad \textbf{let } u_1 = \overline{sort}'(y, 5th(\bar{r})) \textbf{ in} \\
\qquad\qquad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\
\qquad\qquad \textbf{let } v_{21} = < 1st(v_2), v_2 > \textbf{ in} \\
\qquad\qquad \textbf{let } u_2 = 3rd(\bar{r}) \textbf{ in} \\
\qquad\qquad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in} \\
\qquad\qquad < 1st(v), v_{11}, u_1, v_{21}, u_2, v >
\end{array}
\tag{32}
$$

—Stage III. First, using the dependency analysis, for $\Pi \neq ABS$, we have

$$
\begin{array}{rcl}
\overline{sort}'^2(\Pi) & = & (null(1st(\bar{r})))^{\bar{r}}(ID) \ \cup\ ABS \ \cup \\
& & (null(cdr(1st(\bar{r}))))^{\bar{r}}(ID) \ \cup\ (1st(\bar{r}))^{\bar{r}}(\overline{merge}^2(\{1^{st}\})) \ \cup \\
& & (4th(\bar{r}))^{\bar{r}}((1st(v_1))^{v_1}((\Pi^{-2})^{-1}) \ \cup\ (\Pi^{-2})^{-2}) \ \cup \\
& & (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^2((1st(u_1))^{u_1}(\overline{merge}^1((1st(v))^{v}(\Pi^{-1}) \ \cup\ \Pi^{-6})) \ \cup\ \Pi^{-3})) \ \cup \\
& & (2nd(\bar{r}))^{\bar{r}}((1st(v_2))^{v_2}((\Pi^{-4})^{-1}) \ \cup\ (\Pi^{-4})^{-2}) \ \cup \\
& & (3rd(\bar{r}))^{\bar{r}}((1st(u_2))^{u_2}(\overline{merge}^2((1st(v))^{v}(\Pi^{-1}) \ \cup\ \Pi^{-6})) \ \cup\ \Pi^{-5})
\end{array}
$$

which is recursively defined and can be simplified, assuming $1^{st} \in \Pi$, yielding the following:

$$
\begin{array}{rcl}
\overline{sort}'^{2(0)}(\Pi) & = & ABS \\
\overline{sort}'^{2(i+1)}(\Pi) & = & \{1^{st}\} \ \cup \\
& & (4th(\bar{r}))^{\bar{r}}((1st(v_1))^{v_1}((\Pi^{-2})^{-1}) \ \cup\ (\Pi^{-2})^{-2}) \ \cup \\
& & (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}(\{1^{st}\} \ \cup\ \Pi^{-3})) \ \cup \\
& & (2nd(\bar{r}))^{\bar{r}}((1st(v_2))^{v_2}((\Pi^{-4})^{-1}) \ \cup\ (\Pi^{-4})^{-2}) \ \cup \\
& & (3rd(\bar{r}))^{\bar{r}}(\{1^{st}\} \ \cup\ \Pi^{-5}).
\end{array}
\tag{33}
$$

Limiting the depth of selection functions to be 1, we compute the closure of the transitive dependencies for $\overline{sort}'^2$ and obtain

$\Pi^{(0)} = \{1^{st}\}$ and, by (33), $\overline{sort}'^{2(i+1)}(\Pi^{(0)}) = \{1^{st}\} \cup (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}(\{1^{st}\})) \cup \{3^{rd}\}$

$$
\begin{array}{l}
\overline{sort}'^{2(0)}(\Pi^{(0)}) = ABS \\
\overline{sort}'^{2(1)}(\Pi^{(0)}) = \{1^{st}, 3^{rd}\} \\
\overline{sort}'^{2(2)}(\Pi^{(0)}) = \{1^{st}, 3^{rd}, 5^{th}\} \\
\overline{sort}'^{2(3)}(\Pi^{(0)}) = \{1^{st}, 3^{rd}, 5^{th}\}
\end{array}
$$

$\Pi^{(1)} = \{1^{st}, 3^{rd}, 5^{th}\}$ and, by (33), $\overline{sort}'^{2(i+1)}(\Pi^{(1)}) = \{1^{st}\} \cup (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}(ID)) \cup \{3^{rd}\}$

$$
\begin{array}{rcl}
\overline{sort}'^{2(i+1)}(ID) & = & \{1^{st}\} \cup \{4^{th}\} \cup (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}(ID)) \cup \\
& & \{2^{nd}\} \cup \{3^{rd}\}
\end{array}
$$

$$
\begin{array}{l}
\overline{sort}'^{2(0)}(\Pi^{(1)}) = ABS \\
\overline{sort}'^{2(1)}(\Pi^{(1)}) = \{1^{st}, 3^{rd}\} \\
\overline{sort}'^{2(2)}(\Pi^{(1)}) = \{1^{st}, 3^{rd}, 5^{th}\} \\
\overline{sort}'^{2(3)}(\Pi^{(1)}) = \{1^{st}, 3^{rd}, 5^{th}\}
\end{array}
$$

$\Pi^{(2)} = \{1^{st}, 3^{rd}, 5^{th}\}$.

Thus, we obtain the closure projection $\{1^{st}, 3^{rd}, 5^{th}\}$. This matches the intuition that the first component of $\overline{sort'}(y, \bar{r})$ depends only on $3rd(\bar{r})$ and $5th(\bar{r})$, and the third and fifth components depend only on $3rd(\bar{r})$ and $5th(\bar{r})$ too. Pruning functions $\overline{sort}$ and $\overline{sort'}$ yields

$$\begin{aligned}
\widehat{sort}_1(x) = \ & \textbf{if } null(x) \textbf{ then} \\
& \quad < nil, _, _, _, _, _ > \\
& \textbf{else if } null(cdr(x)) \textbf{ then} \\
& \quad < x, _, _, _, _, _ > \\
& \textbf{else let } u_1 = \widehat{sort}_1(odd(x)) \textbf{ in} \\
& \quad \textbf{let } u_2 = \widehat{sort}_1(even(x)) \textbf{ in} \\
& \quad < merge(1st(u_1), 1st(u_2)), _, u_1, _, u_2, _ >
\end{aligned}$$

$$\begin{aligned}
\widehat{sort}'_1(y, \hat{r}_1) = \ & \textbf{if } null(1st(\hat{r}_1)) \textbf{ then} \\
& \quad < cons(y, nil), _, _, _, _, _ > \\
& \textbf{else if } null(cdr(1st(\hat{r}_1))) \textbf{ then} \\
& \quad < merge(cons(y, nil), 1st(\hat{r}_1)), _, <cons(y, nil)>, _, <1st(\hat{r}_1)>, _ > \\
& \textbf{else let } u_1 = \widehat{sort}'_1(y, 5th(\hat{r}_1)) \textbf{ in} \\
& \quad \textbf{let } u_2 = 3rd(\hat{r}_1) \textbf{ in} \\
& \quad < merge(1st(u_1), 1st(u_2)), _, u_1, _, u_2, _ >
\end{aligned}$$

Eliminating $\_$ components and adjusting the indexing yields

$$\begin{aligned}
\widehat{sort}(x) = \ & \textbf{if } null(x) \textbf{ then } < nil > \\
& \textbf{else if } null(cdr(x)) \textbf{ then } < x > \\
& \textbf{else let } u_1 = \widehat{sort}(odd(x)) \textbf{ in} \\
& \quad \textbf{let } u_2 = \widehat{sort}(even(x)) \textbf{ in} \\
& \quad < merge(1st(u_1), 1st(u_2)),\ u_1,\ u_2 >
\end{aligned} \tag{34}$$

$$\begin{aligned}
\widehat{sort}'(y, \hat{r}) = \ & \textbf{if } null(1st(\hat{r})) \textbf{ then } < cons(y, nil) > \\
& \textbf{else if } null(cdr(1st(\hat{r}))) \textbf{ then} \\
& \quad < merge(cons(y, nil), 1st(\hat{r})),\ < cons(y, nil) >,\ < 1st(\hat{r}) >> \\
& \textbf{else let } u_1 = \widehat{sort}'(y, 3rd(\hat{r})) \textbf{ in} \\
& \quad \textbf{let } u_2 = 2nd(\hat{r}) \textbf{ in} \\
& \quad < merge(1st(u_1), 1st(u_2)),\ u_1,\ u_2 >
\end{aligned} \tag{35}$$

For x of length n, using merge sort to compute $sort(cons(y, x))$ takes $O(n \log n)$ time, but using incremental merge sort to compute $\widehat{sort}'(y, \hat{r})$ takes only $O(n)$ time, just like insertion. Note that $O(n)$ is the optimal time complexity for an applicative sorting algorithm that returns a linked list. While insertion takes $O(n)$ space to store the previously sorted list, incremental merge sort uses $O(n \log n)$ space to store also the intermediate results. We can view this as trading space for the need to discover and prove program properties.

8.2 Fibonacci Function

This example shows how our method for caching intermediate results can be used for general systematic program efficiency improvement via caching. We transform the classical exponential-time Fibonacci function into a linear-time one. To perform such optimizations, the user needs to provide the function f_0 and operation $\oplus$, but not special schemas or eurekas that decide to cache particular intermediate results.

Consider the function $fib(x)$ below that computes the xth Fibonacci number and

consider input change operation $x \oplus y = x + 1$.

$$fib(x) = \textbf{if } x \leq 1 \textbf{ then } 1 \textbf{ else } fib(x-1) + fib(x-2) \tag{36}$$

Direct application of the incrementalization method in Liu and Teitelbaum [1995] yields the function fib' below such that, if $fib(x) = r$, then $fib'(x, r) = fib(x+1)$:

$$fib'(x,r) = \textbf{if } x \leq 0 \textbf{ then } 1 \textbf{ else if } x = 1 \textbf{ then } 2 \textbf{ else } r + fib(x-1) \tag{37}$$

But $fib'(x, r)$ takes $O(2^x)$ time, no better than computing $fib(x+1)$ from scratch. Instead, we apply the cache-and-prune method, as follows:

—Stage I. Cache all intermediate results of *sort* using $\mathcal{E}xt$ and $\mathcal{C}lean$ and obtain the following:

$$\begin{aligned}\overline{fib}(x) = \ & \textbf{if } x \leq 1 \textbf{ then } <1, \_, \_> \\ & \textbf{else let } v_1 = \overline{fib}(x-1) \textbf{ in} \\ & \quad \textbf{let } v_2 = \overline{fib}(x-2) \textbf{ in} \\ & \quad <1st(v_1)+1st(v_2),\ v_1,\ v_2>\end{aligned} \tag{38}$$

—Stage II. Derive an incremental version of $\overline{fib}$ under $\oplus$ following the approach in Liu and Teitelbaum [1995], i.e., transform $\overline{fib}(x \oplus y) = \overline{fib}(x+1)$, with $\overline{fib}(x) = \bar{r}$:

1. unfold $\overline{fib}(x+1)$, simplify primitives

$$\begin{aligned}= \ & \textbf{if } x \leq 0 \textbf{ then } <1, \_, \_> \\ & \textbf{else let } v_1 = \overline{fib}(x) \textbf{ in} \\ & \quad \textbf{let } v_2 = \overline{fib}(x-1) \textbf{ in} \\ & \quad <1st(v_1)+1st(v_2),\ v_1,\ v_2>\end{aligned}$$

2. separate cases, replace applications of $\overline{fib}$

$$\begin{aligned}= \ & \textbf{if } x \leq 0 \textbf{ then } <1, \_, \_> \\ & \textbf{else if } x = 1 \textbf{ then} \\ & \quad <2,\ <1, \_, \_>,\ <1, \_, \_>> \\ & \textbf{else let } v_1 = \bar{r} \textbf{ in} \\ & \quad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\ & \quad <1st(v_1)+1st(v_2),\ v_1,\ v_2>\end{aligned}$$

We obtain function $\overline{fib}'$ such that, if $\overline{fib}(x) = \bar{r}$, then $\overline{fib}'(x, \bar{r}) = \overline{fib}(x+1)$:

$$\begin{aligned}\overline{fib}'(x,\bar{r}) = \ & \textbf{if } x \leq 0 \textbf{ then } <1, \_, \_> \\ & \textbf{else if } x = 1 \textbf{ then } <2,\ <1, \_, \_>,\ <1, \_, \_>> \\ & \textbf{else } <1st(\bar{r})+1st(2nd(\bar{r})),\ \bar{r},\ 2nd(\bar{r})>\end{aligned} \tag{39}$$

—Stage III. Using dependency analysis for $\overline{fib}'$ in a similar way as for $\overline{foo}'$ but simpler, we obtain the closure projection $\{1^{st}, 1^{st}2^{nd}\}$. To prune, we first obtain

$$\begin{aligned}\widehat{fib}_1(x) = \ & \textbf{if } x \leq 1 \textbf{ then } <1, \_, \_> \\ & \textbf{else let } v_1 = \overline{fib}(x-1) \textbf{ in} \\ & \quad \textbf{let } v_2 = \overline{fib}(x-2) \textbf{ in} \\ & \quad <1st(v_1)+1st(v_2),\ <1st(v_1), \_, \_>,\ \_>\end{aligned}$$

$$\begin{aligned}\widehat{fib}_1'(x,\hat{r}_1) = \ & \textbf{if } x \leq 0 \textbf{ then } <1, \_, \_> \\ & \textbf{else if } x = 1 \textbf{ then } <2,\ <1, \_, \_>,\ <1, \_, \_>> \\ & \textbf{else } <1st(\hat{r}_1)+1st(2nd(\hat{r}_1)),\ <1st(\hat{r}_1), \_, \_>,\ \_>\end{aligned}$$

Simplify $\widehat{fib}_1$ and $\widehat{fib}_1'$, remove $\_$ components, and lift the single component in the second component of $\widehat{fib}_1$ and $\widehat{fib}_1'$ as discussed in Section 7. We obtain

$$\begin{aligned}\widehat{fib}(x) = \ & \textbf{if } x \leq 1 \textbf{ then } <1> \\ & \textbf{else let } u_1 = fib(x-1) \textbf{ in} \\ & \quad <u_1 + fib(x-2),\ u_1>\end{aligned} \tag{40}$$

Table III. Some Examples and Their Running Times

Example	Batch	Incremental	Optimized	Note
attribute evaluation	$O(n)$	$O(P+A)$		
binomial coefficient	$O(2^i)$	$O(j)$	$O(i*j)$	
string editing	$O(3^{i+j})$	$O(i)$	$O(i*j)$	
dag path sequence	$O(2^n)$		$O(n^2)$	aux. info.
longest common subsequence	$O(2^{i+j})$		$O(i*j)$	aux. info.
row neighborhood summation	$O(n_1*n_2*m)$		$O(n_1*n_2)$	array
local neighborhood summation	$O(n^2*m^2)$		$O(n^2)$	array

$$\begin{aligned}\widehat{fib}'(x,\hat{r}) = \ & \textbf{if } x \le 0 \textbf{ then } <1> \\ & \textbf{else if } x = 1 \textbf{ then } <2,1> \\ & \textbf{else } <1st(\hat{r}) + 2nd(\hat{r}),\ 1st(\hat{r})>\end{aligned} \tag{41}$$

Clearly, $\widehat{fib}'(x,\hat{r})$ takes constant time. Note that $fib(x) = 1st(\widehat{fib}(x))$ and, if $\widehat{fib}(x) = \hat{r}$, then $\widehat{fib}(x+1) = \widehat{fib}'(x,\hat{r})$. Using definition (41) of $\widehat{fib}'$ in this last equation, we obtain a new definition for $\widehat{fib}$:

$$\begin{aligned}\widehat{fib}(x+1) = \ & \textbf{if } x \le 0 \textbf{ then } <1> \\ & \textbf{else if } x = 1 \textbf{ then } <2,1> \\ & \textbf{else let } \hat{r} = \widehat{fib}(x) \textbf{ in } <1st(\hat{r})+2nd(\hat{r}),\ 1st(\hat{r})>\end{aligned}$$

Letting $v = x+1$, we get

$$\begin{aligned}\widehat{fib}(v) = \ & \textbf{if } v \le 1 \textbf{ then } <1> \\ & \textbf{else if } v = 2 \textbf{ then } <2,1> \\ & \textbf{else let } \hat{r} = \widehat{fib}(v-1) \textbf{ in } <1st(\hat{r})+2nd(\hat{r}),\ 1st(\hat{r})>\end{aligned} \tag{42}$$

We define $fib(v) = 1st(\widehat{fib}(v))$ using definition (42) of $\widehat{fib}$. This computes the Fibonacci numbers in linear time, as desired.

8.3 More Examples

The cache-and-prune method is based on general techniques for static analyses and program transformations. They are not developed for particular program schema or particular application problems. Therefore, many examples can be improved with them. Table III lists some of them and summarizes their running times. The heading "Batch" refers to a given nonincremental program; "Incremental" refers to an incremental version of the batch program under a given input change operation; and "Optimized" refers to an optimized version of the batch program that uses an incremental version for an iteration or recursion. The column "Note" specifies techniques needed, if any, in addition to the transformations for caching and pruning.

These examples are classic problems, and no systematic method for obtaining the optimizations was previously given. They illustrate different aspects, including useful improvements, of cache-and-prune. We explain each example briefly below. Details for some of these and other examples are given in Liu et al. [Liu 1997; Liu and Stoller 1998; Liu et al. 1996].

Attribute Evaluation. Attribute grammars are widely used in programming language implementations and systems [Deransart et al. 1988; Knuth 1968; Paakki 1995]. Given an attribute grammar, a set of recursive functions can be constructed

to evaluate the attribute values for any derivation tree of the grammar [Katayama 1984]. Each function evaluates a synthesized attribute of a nonterminal, and the value of a synthesized attribute of the root symbol is the final return value of interest. We consider subtree replacement as the input change operation, given by a new subtree and a path from the root of the whole tree to the root of the subtree to be replaced.

Caching all intermediate results yields a set of extended functions that returns an attributed tree instead of just the value of a synthesized attribute at the root. Then, incrementalizing the extended functions under a subtree replacement just composes a new attributed tree from the old, based on equalities between old attributes and new attributes, evaluating only attributes whose values are affected by the subtree replacement, yielding a set of incremental recursive functions. Suppose a given batch program takes $O(n)$ time to evaluate each attribute once. The incremental program takes $O(P + A)$ time, where P (for PATH) is the path from the root of the whole tree to the root of the new subtree, and A (for AFFECTED) is the set of attributes whose values are different in the new tree [Reps et al. 1983]. This is the best that can be done in a functional language [Pugh and Teitelbaum 1989].

Binomial Coefficient and String Editing. Binomial coefficient $binomial(i, j)$, for $0 \leq j \leq i$, is the number of j-element subsets of an i-element set [Cormen et al. 1990; Partsch 1990]. A straightforward recursive program for $binomial(i, j)$ takes exponential time. Using the cache-and-prune method, we obtain a incremental program that computes $binomial(i + 1, j)$ using $binomial(i, j)$ in $O(j)$ time and $O(j)$ space. Using this incremental program, we can obtain an optimized program that computes $binomial(i, j)$ in $O(i * j)$ time and $O(j)$ space, both of which are optimal for this problem.

The string-edit problem is to find the minimum cost $edit(s_1, s_2)$ for modifying two strings s_1 of length i and s_2 of length j so that they are the same [Aho et al. 1974; Purdom and Brown 1985]. A straightforward recursion that tries all possible deletions, insertions, and substitutions takes $O(3^{\max(i,j)})$ time. Using the cache-and-prune method, we obtain an incremental program that computes $edit(s_1, cons(c, s_2))$ using $edit(s_1, s_2)$ in $O(i)$ time. Using this incremental program, we can obtain an optimized program that computes $edit(s_1, s_2)$ in $O(i * j)$ time.

Dag Path Sequence and Longest Common Subsequence. Given a directed acyclic graph, and a string whose elements are vertices in the graph, the path-sequence problem is to compute the length of the longest subsequence in the string that forms a path in the graph [Bird 1984]. Suppose $path(s)$ computes the required length for a string s of length n. A straightforward program takes $O(2^n)$ time. Using the cache-and-prune method, extended to cache also certain auxiliary information, we obtain an incremental program that computes $path(cons(c, s))$ using $path(s)$ in $O(n)$ time; using this incremental program, we can obtain an optimized program that computes $path(s)$ in $O(n^2)$ time [Liu et al. 1996].

The longest-common-subsequence problem finds the length $lcs(s_1, s_2)$ of the longest common subsequence of two given sequences s_1 of length i and s_2 of length j [Cormen et al. 1990]. A simple recursive program takes $O(2^{\max(i,j)})$ time [Cormen et al. 1990]. Using cache-and-prune, again with auxiliary information, we obtain

```
[1]  for i := 0 to n1 − m do
[2]     s[i] := 0;
[3]     for k := 0 to m − 1 do
[4]        for l := 0 to n2 − 1 do
[5]           s[i] := s[i] + a[i + k, l]
```

(a)

```
[6]   s[0] := 0;
[7]   for k := 0 to m − 1 do
[8]      s1[k] := 0;
[9]      for l := 0 to n2 − 1 do
[10]        s1[k] := s1[k] + a[k, l];
[11]     s[0] := s[0] + s1[k];
[12]  for i := 1 to n1 − m do
[13]     s1[i+m−1] := 0;
[14]     for l := 0 to n2 − 1 do
[15]        s1[i+m−1] := s1[i+m−1] + a[i+m−1, l];
[16]     s[i] := s[i−1] − s1[i−1] + s1[i+m−1]
```

(b)

Fig. 8. Programs for row-neighborhood-summation problem.

an incremental program that computes $lcs(s_1, cons(c, s_2))$ using $lcs(s_1, s_2)$ in $O(i)$ time; using the incremental program, we can obtain an optimized program that computes $lcs(s_1, s_2)$ in $O(i * j)$ time.

Row-Neighborhood-Summation and Local-Neighborhood Problems. We show how the general principle of the cache-and-prune method applies to improving imperative programs that use arrays. Given an array a with n_1 rows and n_2 columns, the row-neighborhood-summation problem computes, for each row i ($0 \le i \le n_1 - m$), the sum of the m-by-n_2 rectangle comprising rows i through $i+m-1$. The straightforward program in Figure 8(a) takes $O(n_1 n_2 m)$ time, while the efficient program in Figure 8(b) uses $O(n_1 n_2)$ time and $O(n_1)$ additional space. We sketch how to obtain the efficient program systematically following the basic ideas of cache-and-prune. The detailed analyses of loops and arrays needed to make the transformations automatic are described in Liu and Stoller [1998].

As discussed in Section 7, to optimize a loop we take the loop body as the program to be incrementalized and take the loop increment as the update operation. In this example, incrementalizing the body of the loop over k and the body of the loop over l leaves them unchanged. For the loop over i, the loop body comprises lines [2-5], and the update operation is the increment to i. Stage I introduces an array s_2 that caches the results computed by the loop over k: $s_2[i, k]$ is the sum of row $i + k$. The transformations in Liu and Stoller [1998] eliminate the redundancy in this array, replacing it with an array s_1 such that $s_1[i + k] = s_2[i, k]$; thus, $s_1[i]$ is the sum of row i. Stage II incrementalizes the computation of $s[i]$: in line [16], $s[i]$ is computed from $s[i-1]$ by subtracting the sum of row $i-1$ and adding the sum of row $i+m-1$, and in lines [13-15], the result $s_1[i+m-1]$ is maintained by summing row $i+m-1$. Stage III has no effect, since all intermediate results are used. The final optimized program uses the incrementalized loop body. The initialization in lines [6-11] computes $s[0]$ and prepares the cached results $s_1[0]$ through $s_1[m-1]$.

The row-neighborhood-summation problem is just an example of a class of problems called local-neighborhood problems, which involve computing information about local neighborhoods of objects, (pixels, rows, etc.). Such problems are common in image processing [Webb 1992; Wells 1986; Zabih 1994; Zabih and Woodfill 1994]. The cache-and-prune approach can be used to optimize programs for such

problems. This is illustrated by the row-neighborhood problem above and by the local-neighborhood-summation problem in Liu and Stoller [1998].

9. RELATED WORK AND CONCLUSION

Related work in incrementalization, as well as work related to the analysis and transformation techniques used for caching and pruning, has been discussed in Section 7. Here we compare our work with related work in program efficiency improvement using caching techniques. Caching is the basis of many techniques for developing efficient programs and optimizing programs. Bird [1980] and Cohen [1983] provide overviews. We classify most of these into three classes.

Separate Caching. In the first class, a global cache separate from a subject program is employed to record values of subcomputations that may be needed later, and certain strategies are chosen for using and managing the cache. We call this technique *separate caching*. It corresponds to exact tabulation in Bird [1980] and the large-table method in Cohen [1983]. The initial idea of separate caching, "memo" functions, proposed by Michie [1968], belongs to this class. Uses of the word "memoization" mainly refer to techniques in this class [Partsch 1990].

Since then, there has been additional work on general strategies for separate caching. For example, Hughes [1985] discusses lazy memo functions that are suitable for use in systems with lazy evaluation. Mostow and Cohen [1985] discuss issues for speeding up Interlisp programs by caching in the presence of side effects. Pugh [1988] describes improved cache replacement strategies for a simple functional language. Abadi, Lampson, and Levy [Abadi et al. 1996] study caching in the context of λ-calculus. Two trends are apparent: studying *specialized* cache strategies for classes of problems, and adding *annotations* or *specifications* to subject programs to provide hints to the cache strategies. An example of the former is the stable decomposition scheme of Pugh and Teitelbaum [1989]. Examples of the latter include work by Keller and Sleep [1986], which uses annotations for applicative languages, work by Sundaresh and Hudak [Sundaresh 1991; Sundaresh and Hudak 1991], which decides what to cache based on given input partitions of programs, and work by Hoover [1992], which uses annotations for an imperative language.

The pros and cons are well discussed by Bird [1980] and Cohen [1983]. To summarize, the idea is simple, and the subject programs are basically unchanged. But the caching methods are dynamic and thus fundamentally interpretive. Moreover, the strategies for the use and management of the separate cache cannot easily be made both general and powerful, so they can be sources of inefficiency.

Schema-Based Integrated Caching. In the second and third classes, the above drawbacks are overcome by transforming subject programs to integrate caching into the transformed programs. Techniques in the second class apply transformations based on special properties and schemas of subject programs. We call this *schema-based integrated caching*. A nice survey of most of these ideas can be found in Bird [1980]. Some uses of the word "tabulation" mainly refer to techniques in this class [Partsch 1990]. Typical examples of these techniques are dynamic programming [Aho et al. 1974], schemas of redundancies [Cohen 1983], and tupling [Chin 1993; Chin and Khoo 1993; Pettorossi 1984; 1987; Pettorossi and Proietti 1997]. Dynamic

programming applies to problems that can be divided into subproblems and solved from small subproblems to larger ones by storing and using results of smaller ones. Work on schemas of redundancies studies several forms of redundant recursive calls and their mathematical properties and provides transformations to eliminate them. Tupling looks for a recurrent pattern in computing intermediate results, groups those computed in the pattern into a tuple, and transforms the program to compute the tuple progressively.

Note that separate caching with a specialized cache strategy for a certain class of problems can be realized as schema-based integrated caching for this class of problems. More precisely, for any problem that fits into this class, we treat the corresponding program as fitting into a certain schema. We can then integrate the specialized cache strategy by transforming the corresponding program and obtain a transformed program with schema-based integrated caching. In this case, the separate caching corresponds to an interpretive mechanism; the transformation with integration is similar to compiling; and a transformed program corresponds to a compiled program.

While integrating caching into transformed programs eliminates the interpretive overhead of separate caching, a drawback of schema-based integrated caching is its lack of generality.

Principle-Based Integrated Caching. Techniques in the third class analyze and transform programs according to general principles. We call this *principle-based integrated caching*. Often, such principles are used to derive a relatively complete set of strategies and rules for programs written in a certain language, and these strategies and rules are used to transform programs. For example, the conventional strength reduction optimization [Allen 1969; Allen et al. 1981; Cocke and Kennedy 1977] identifies subcomputations such as multiplications that can be replaced with subcomputations such as additions while maintaining the values of these subcomputations. Similarly, the APTS program transformation system [Paige 1983; 1990; 1994] identifies set expressions in SETL that can be maintained using finite differencing rules [Paige and Koenig 1982].

Sometimes it is not sufficient to have only a fixed set of strategies and rules. Seeking more flexibility and broader applicability, KIDS [Smith 1990] advocates certain high-level strategies but leaves the choice of which intermediate results to maintain as manual decisions. CIP [Partsch 1990] also proposes a general strategy for caching, but it may even lead to less efficient programs.

Recently, certain principles that can directly guide program transformations have been proposed. Webber's *principle of least computation* [Webber 1993; 1995] avoids subcomputations whose values have been computed before or are not needed. Basically, first-order purely functional programs are transformed into *trace grammars*, which are *thinned* using this principle and then transformed back. The analysis used by thinning can lead to clever optimizations [Webber 1997], but it does not perform explicit incremental computation such as strength reduction. Hall's *principle of redistributing intermediate results* [Hall 1990; 1991] finds paths from subcomputations to multiple uses of their values. However, the method uses a great deal of program design knowledge, including annotations of invariants, test-case inputs, and proofs of correctness. Also, it guarantees correctness of the transformed programs only on

the test-case inputs.

Our approach to program improvement via caching falls into the third class. The intrinsic iterative computation property of programs drives the incremental computation of each iteration, which in turn drives the decision of what intermediate results to cache. This approach is a crucial complement to any incremental computation technique for achieving the goal of program improvement.

Among principle-based integrated caching methods, our approach is not limited to using a fixed set of rules for program analysis and transformation; on the contrary, the approach can even be used to derive such rules when necessary. Compared to the general approaches advocated by KIDS or CIP, our approach is more algorithmic and automatable.

There is an additional advantage of formulating the problem of program efficiency improvement as incrementalizing repetitive computations. It allows auxiliary information to be used together with intermediate results [Liu et al. 1996]. This is impossible for methods based on principles that exploit only intermediate results.

In conclusion, incremental computation has widespread applications throughout computing. This article proposes a general systematic method for caching intermediate results for incremental computation which can also be used for general program efficiency improvement. The modularity of the method lets us integrate other techniques in our framework and reuse our components for other optimizations. Although our approach is presented in terms of a simple functional language, the underlying principles are general and apply to other languages as well. A prototype system, CACHET [Liu 1995], based on our approach is under development.

ACKNOWLEDGMENT

We thank the anonymous referees for their exceptionally careful reviews and helpful comments.

REFERENCES

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

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

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

Allen, F. E. 1969. Program optimization. In *Annual Review of Automatic Programming.* Vol. 5. Pergamon Press, New York, 239–307.

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

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

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

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

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

Cartwright, R. 1984. Recursive programs as definitions in first order logic. *SIAM J. Comput. 13,* 2 (May), 374–408.

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

Chin, W.-N. and Khoo, S.-C. 1993. Tupling functions with multiple recursion parameters. In *Proceedings of the 3rd International Workshop on Static Analysis*, P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, Eds. Lecture Notes in Computer Science, vol. 724. Springer-Verlag, Berlin, 124–140.

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

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

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

Deransart, P., Jourdan, M., and Lorho, B. 1988. *Attribute Grammars: Definitions, Systems, and Bibliography*. Lecture Notes in Computer Science, vol. 323. Springer-Verlag, Berlin.

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

Gunter, C. A. 1992. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass.

Hall, R. J. 1990. Program improvement by automatic redistribution of intermediate results. Tech. Rep. AI-TR-1251, Artificial Intelligence Laboratory, MIT, Cambridge, Mass. Dec.

Hall, R. J. 1991. Program improvement by automatic redistribution of intermediate results: An overview. In *Automating Software Design*, M. R. Lowry and R. D. McCartney, Eds. AAAI Press/The MIT Press, 339–372.

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

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

Hughes, J. 1990. Compile-time analysis of functional programs. In *Research Topics in Functional Programming*, D. Turner, Ed. Addison-Wesley, Reading, Mass., 117–153.

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

Jones, S. B. and Le Métayer, D. 1989. Compile-time garbage collection by sharing analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 54–74.

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

Keller, R. M. and Sleep, M. R. 1986. Applicative caching. *ACM Trans. Program. Lang. Syst. 8,* 1 (Jan.), 88–108.

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

Knuth, D. E. 1968. Semantics of context-free languages. *Math. Syst. Theory 2,* 2 (June), 127–145.

Launchbury, J. 1989. Projection factorisations in partial evaluation. Ph.D. thesis, Department of Computing, University of Glasgow, Glasgow, Scotland.

Launchbury, J. 1991. Strictness and binding-time analysis: Two for the price of one. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. ACM, New York, 80–91.

Lawall, J. L. and Danvy, O. 1993. Separating stages in the continuation-passing style transformation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 124–136.

Liu, Y. A. 1995. CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*. IEEE CS Press, Los Alamitos, Calif., 19–26.

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

Liu, Y. A. 1998. Dependence analysis for recursive data. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*. IEEE CS Press, Los Alamitos, Calif.

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

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

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

Michie, D. 1968. "memo" functions and machine learning. *Nature 218*, 19–22.

Mogensen, T. 1989. Separating binding times in language specifications. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 12–25.

Mostow, D. J. and Cohen, D. 1985. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, San Francisco, Calif., 165–172.

Paakki, J. 1995. Attribute grammar paradigms—A high-level methodology in language implementation. *ACM Comput. Surv. 27,* 2 (June), 196–255.

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

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

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

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

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

Pettorossi, A. 1984. 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.

Pettorossi, A. 1987. Strategical derivation of on-line programs. In *Program Specification and Transformation*, L. G. L. T. Meertens, Ed. North-Holland, Amsterdam, 73–88.

Pettorossi, A. and Proietti, M. 1997. Program derivation via list introduction. In *Algorithmic Languages and Calculi*, R. Bird and L. Meertens, Eds. Chapman & Hall, London, U.K.

Plotkin, G. D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoret. Comput. Sci. 1*, 125–159.

Pugh, W. 1988. An improved cache replacement strategy for function caching. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. ACM, New York, 269–276.

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

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

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

Reps, T. and Turnidge, T. 1996. Program specialization via program slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 409–429.

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

Rosendahl, M. 1989. Automatic complexity analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 144–156.

Scott, D. S. 1982. Lectures on a mathematical theory of computation. In *Theoretical Foundations of Programming Methodology*, M. Broy and G. Schmidt, Eds. D. Reidel Publishing Company, 145–292.

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

Sundaresh, R. S. 1991. Building incremental programs using partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York, 83–93.

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

Traub, K. R. 1986. A compiler for the MIT tagged-token dataflow architecture. M.S. thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts. Appeared as Technical Report LCS TR-370, August, 1986.

Wadler, P. and Hughes, R. J. M. 1987. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 385–407.

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

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

Webber, A. B. 1993. Principled optimization of functional programs. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, N.Y. Also appeared as Technical Report TR 93-1363, June 1993.

Webber, A. B. 1997. Program analysis using binary relations. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM, New York, 249–160.

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

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

Yeh, D. and Kastens, U. 1988. Improvements on an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Not. 23,* 12, 45–50.

Zabih, R. 1994. Individuating unknown objects by combining motion and stereo. Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, Calif.

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

Received November 1996; revised August 1997; accepted November 1997