Caching Intermediate Results for Program Improvement

Yanhong A. Liu* Tim Teitelbaum*

March 1995

Abstract

A systematic approach is given for symbolically caching intermediate results useful for deriving incremental programs from non-incremental programs. We exploit a number of program analysis and transformation techniques, centered around effective caching based on its utilization in deriving incremental programs, in order to increase the degree of incrementality not otherwise achievable by using only the return values of programs that are of direct interest. Our method can be applied straightforwardly to provide a systematic approach to program improvement via caching.

1 Introduction

Incremental programs take advantage of repeated computations on inputs that differ only slightly from one another, making use of the old output in computing a new output rather than computing from scratch. Methods of incremental computation have widespread application, e.g., optimizing compilers [2, 9, 11], transformational programming [29, 32, 42], interactive editing systems [4, 38], *etc.*

In this paper, we (a) extend our previous work on deriving incremental programs [26] with a new technique for caching intermediate results, and (b) show how the technique applies to the general problem of program improvement via caching.

Deriving Incremental Programs. Given a program f and an input change $\oplus$, 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 $\oplus$.

Liu and Teitelbaum [26] give a systematic transformational approach for deriving an incremental program f' from a given program f and an input change $\oplus$. The basic idea is to identify in the computation of $f(x \oplus y)$ those subcomputations that are also performed in the computation of $f(x)$ and whose values can be retrieved from the cached result r of $f(x)$. The computation of $f(x \oplus y)$ is transformed symbolically to avoid re-performing these subcomputations by replacing them with corresponding retrievals. This efficient way of computing $f(x \oplus y)$ is captured in the definition of $f'(x, y, r)$.

Caching Intermediate Results. The above approach has a limitation. The derived program $f'(x, y, r)$ avoids only subcomputations that are performed by $f(x)$ *and* whose values can be retrieved from the cached result r of $f(x)$. There may be subcomputations of $f'(x, y, r)$ that are also performed by $f(x)$ *but* whose values can not be retrieved from r. If the values of these computations are also cached and returned, then after the input change $\oplus$, we can avoid recomputing them by retrieving their values from the extended

*The authors gratefully acknowledge the support of the Office of Naval Research under contract No. N00014-92-J-1973.
Authors' address: Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: {yanhong, tt}@cs.cornell.edu

return value of the old computation. We call the values of these computations *intermediate results useful for computing f incrementally under $\oplus$.*

Examples where intermediate results are needed for incremental computation include incremental parsing [16] and incremental attribute evaluation [24, 39, 50]. An incremental parser may cache, in addition to the derived parse tree, the $LR(0)$ state corresponding to each shift and reduction. An attribute evaluator may only return some designated synthesized attribute of the root [18], but the corresponding incremental attribute evaluator may cache the whole attributed tree.

Program Improvement via Caching. Deriving incremental programs and caching intermediate results provide a principled approach for program improvement using caching. In essence, every program computes by fixed point iteration, which is why loop optimizations are so important. A straightforward idea then is to compute the result of each iteration incrementally using the stored result of the previous iteration, which is why strength reduction [3] and related techniques [31] are crucial for performance. Observe that, most of the time, not only the result, but also the intermediate results computed in one iteration can be useful for efficiently computing the result of the next iteration. Thus, these intermediate results need to be identified, used, and maintained as well.

We can regard a loop body as a program f, and a loop increment as a change $\oplus$. Then the goal of computing loops incrementally corresponds to transforming f into f', an incremental version of f under $\oplus$, and the problem of identifying intermediate results that can be used from iteration to iteration corresponds to deciding which intermediate results are useful for computing f incrementally under $\oplus$. Once these intermediate results have been determined and the program f has been extended to a program $\hat{f}$ that returns them as well, a program $\hat{f}'$ can be derived that incrementally computes these intermediate results as well as the value of f.

This Paper. We present a clean three-stage method, called *cache-and-prune*, for caching intermediate results useful for computing f incrementally under $\oplus$. The basic idea is to (I) extend the program f to a program $\bar{f}$ that returns all intermediate results, (II) incrementalize the program $\bar{f}$ under $\oplus$ to obtain an incremental version $\bar{f}'$, and (III) using the dependencies in $\bar{f}'$, prune the extended program $\bar{f}$ to a program $\hat{f}$ that returns only the useful intermediate results. Additionally, we also prune the program $\bar{f}'$ to directly obtain a program$\hat{f}'$ that incrementally maintains only the useful intermediate results. The contributions of this paper are as follows.

1. We give a systematic approach for caching intermediates results useful for computing f incrementally under $\oplus$, and for constructing a corresponding program that incrementally maintains these intermediate results. Previous work on this relies on a fixed set of rules [3, 31], applies only to programs with certain properties or schemas [5, 10, 33, 34], or requires program annotations [14, 19, 44].

2. Our cache-and-prune method consists of three independent stages, and thus is modular. It has certain nice properties. 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. Therefore, the whole method is optimal with respect to the incremental techniques of Stage II (for which we use [26]). Stages I and III are simple, clean, and fully-automatable.

3. We develop in Stage III a backward dependency analysis that uses domain projections to specify sufficient information, which is a natural application of the techniques previously used for other analyses [22, 46]. Our projections specify specific components of compound values, rather than just heads or tails of

list values, and thus provide more accurate information. The technique may also be used to assist general program optimizations in context, like tuple elimination [45].

4. Our result can be applied straightforwardly and systematically to program improvement via caching. The classical example of the Fibonacci function, which can be improved dramatically by various caching techniques, is shown in Section 8. A comprehensive comparison with work in program improvement via caching is given in Section 9.

This paper 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 program analysis and transformation techniques used and the time/space trade-off. Several examples are given in Section 8. Finally, we discuss related work and conclude in Section 9.

2 Defining the Problem

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

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

Each constructor c, primitive function p, and user-defined function f has a fixed arity. A program is a set F 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 $x = \langle x_1,...,x_n \rangle$. The semantics of the language is strict. Figure 1 gives some example definitions.

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

$$\begin{array}{rl} foo(x) = & \textbf{if } x \le 2 \textbf{ then } 1 \\ & \textbf{else } boo(x) + foo(x-3) \\ boo(x) = & foo(x-1) + foo(x-2) \end{array}$$

$fib(x)$: compute the x-th Fibonacci number

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

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

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

Figure 1: Example function definitions

An input change $\oplus$ to a function f_0 combines an old input $x = \langle x_1,...,x_n \rangle$ and a change $y = \langle y_1,...,y_m \rangle$ to form a new input $x' = \langle x'_1,...,x'_n \rangle = x \oplus y$, where each x'_i is some function of x_j's and y_k's. For example, an input change $\oplus_1$ to the function foo or fib in Figure 1 can be defined by $x' = x \oplus_1 y = x + 1$, and an input change $\oplus_2$ to $sort$ can be $x' = x \oplus_2 y = cons(y, x)$.

We need cost models to discuss efficient computation and program improvement. In this paper, we use an asymptotic time model, and write

$$t(f(v_1, .., v_n)) \tag{2}$$

to denote the asymptotic time of computing $f(v_1, ..., v_n)$. Since only asymptotic time is of concern, it is sufficient 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 principle of trading space for speed. We assume that we have unlimited space to be used for achieving the least asymptotic time possible. The pruning saves time as well as space for computing and maintaining intermediate results that are not useful for incremental computation. There are standard constructions for mechanical time analysis [40, 48], but automatic space analysis and the trade-off between time and space are problems open for study. Related issues about the trade-off between time and space is discussed in Section 7.

Given a program f_0 and an input change $\oplus$, we can use the approach in [26] 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] Instead of trivially defining $f_0'(x, y, r)$ as $f_0(x \oplus y)$, we attempt to make $f_0'(x, y, r)$ as efficient as possible by having it use the cached result r of $f_0(x)$ as much as possible. A number of examples like outer product and selection sort are given in [26]. For the function foo in Figure 1 and input change $x \oplus_1 y = x + 1$, the function foo' given in Figure 2 can be derived. Unfortunately, computing $foo'(x, r)$ is not much faster than computing $foo(x + 1)$ from scratch.

This foo example illustrates that there are cases where we can compute the value of $f_0(x \oplus y)$ more quickly by caching and using, in addition to the value of $f_0(x)$, the intermediate results computed in $f_0(x)$. For example, the value of $foo(x - 1) + foo(x - 2)$, which could be used in computing $foo'(x, r)$, is also computed by $foo(x)$ but can not be retrieved from r. Thus, we can cache this intermediate value and use it in computing the value of $foo(x + 1)$ faster.

We need consistent notations for the mechanical transformation that caches intermediate results. We use $<>$ to denote a tuple constructed by the transformation that bundles intermediate results with the original return value, with $1st$ returning the first element, which is always the original value, and rst returning a tuple of the remaining elements, which are the corresponding intermediate results. We use nth to get the nth element of such a tuple, and we use an infix operation @ to concatenate two such tuples.

For typographical convenience, we shall always use x to refer to the previous input to f_0, r the cached result of $f_0(x)$, y the change parameter to the input x, x' the new input $x \oplus y$, and f_0' an incremental version of f_0 under $\oplus$. We use $\bar{f}_0$ to refer to the 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 refer to the pruned function that returns only the intermediate results of f_0 useful for incremental computation, $\hat{r}$ the cached result of $\hat{f}_0(x)$, and $\hat{f}_0'$ a function that incrementally maintains only the useful intermediate results.

We use the function foo in Figure 1 as a running example. At the end, we obtain the functions $\widehat{foo}$, $\widehat{boo}$, and $\widehat{foo}'$ as shown in Figure 2. Rather than computing foo or $\widehat{foo}$ from scratch using $O(3^n)$ time, $\widehat{foo}'$ computes incrementally using only $O(1)$ time.

[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 [26].

If $foo(x)$ returns r, then $foo'(x,r)$ computes $foo(x+1)$.
For x of length n, $foo'(x,r)$ takes time $O(3^n)$;
$foo(x+1)$ takes time $O(3^n)$.

$foo(x) = 1st(\widehat{foo}(x))$.
For x of length n, $\widehat{foo}(x)$ takes time $O(3^n)$;
$foo(x)$ takes time $O(3^n)$.

If $\widehat{foo}(x)$ returns $\hat{r}$, then $\widehat{foo}'(x,\hat{r})$ computes $\widehat{foo}(x+1)$.
For x of length n, $\widehat{foo}'(x,\hat{r})$ takes time $O(1)$;
$\widehat{foo}(x+1)$ takes time $O(3^n)$.

$$
\begin{array}{lcl}
foo'(x,r) & = & \textbf{if } x \le 1 \textbf{ then } 1 \\
 & & \textbf{else if } x = 2 \textbf{ then } 3 \\
 & & \textbf{else } r + foo(x-1) + foo(x-2) \\
\widehat{foo}(x) & = & \textbf{if } x \le 2 \textbf{ then } < 1 > \\
 & & \textbf{else let } v_1 = \widehat{boo}(x) \textbf{ in} \\
 & & \quad < 1st(v_1) + foo(x-3),\ v_1 > \\
\widehat{boo}(x) & = & \textbf{let } v_1 = foo(x-1) \textbf{ in} \\
 & & < v_1 + foo(x-2),\ < v_1 >> \\
\widehat{foo}'(x,\hat{r}) & = & \textbf{if } x \le 1 \textbf{ then } < 1 > \\
 & & \textbf{else if } x = 2 \textbf{ then } < 3, < 2, < 1 >>> \\
 & & \textbf{else } < 1st(\hat{r}) + 1st(2nd(\hat{r})), \\
 & & \quad < 1st(\hat{r}) + 1st(2nd(2nd(\hat{r}))),\ < 1st(\hat{r}) >>>
\end{array}
$$

Figure 2: Resulting "foo" function definitions

3 Approach

Since $f_0(x \oplus y)$ can be computed incrementally using only values that are available like the value of $f_0(x)$, we want to extend f_0 so that intermediate values computed in $f_0(x)$ that are also used in computing $f_0(x \oplus y)$ are returned as well.

Selective Caching Method. A relatively straightforward method is to mimic the derivation approach in [26], namely, to identify subcomputations of $f_0(x \oplus y)$ that are also performed in the computation of $f_0(x)$ *but* whose values can not be retrieved from the cached result r of $f_0(x)$, and transform $f_0(x)$ to cache and return these values. We call this method the *selective caching* of intermediate results. This method is as heavy-weight as the derivation approach in [26].

Let $g(x)$ capture these intermediate results of $f_0(x)$, and let $\hat{f}_0^1 = \langle f_0, g \rangle$. To compute $f_0(x \oplus y)$ incrementally, we need to maintain $g(x \oplus y)$ to support incremental computation after further input changes. Thus, we want 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 subcomputations of $g(x \oplus y)$, and thus of $\hat{f}_0^1(x \oplus y)$, that are also performed in the computation of $f_0(x)$, and thus of $\hat{f}_0^1(x)$, but whose values can not be retrieved even from $\hat{r}^1$. To capture these other subcomputations, we can apply the selective caching method again, to the extended program $\hat{f}_0^1$ and input change $\oplus$.

The above process may repeat until we obtain a program $\hat{f}_0^i$ such that all subcomputations of $\hat{f}_0^i(x \oplus y)$ that are also performed in $\hat{f}_0^i(x)$ can be retrieved from the cached result $\hat{r}^i$ of $\hat{f}_0^i(x)$. Intuitively, this process always terminates since there exists an upper bound of such $\hat{f}_0^i$'s, namely, a program that returns all intermediate results of f_0. However, the number of repetitions depends at least on f_0 and $\oplus$. Moreover, each repetition is heavy-weight. Therefore, we propose instead a simple three-stage method called *cache-and-prune*.

Cache-And-Prune Method. The cache-and-prune method consists of three stages, where a light-weight dependency analysis may be iterated a number of times in Stage III, but the heavy-weight derivation approach in [26] is applied 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 tuple containing both the intermediate results and the value of $f_0(x)$, such that

$$1st(\bar{f}_0(x)) = f_0(x) \quad \text{and} \quad t(\bar{f}_0(x)) \le t(f_0(x)). \tag{3}$$

Stage II derives a function $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$, using the approach in [26], such that if $\bar{f}_0(x) = \bar{r}$, then we have if $\bar{f}_0(x \oplus y) = \bar{r}'$, then

$$\bar{f}_0'(x, y, \bar{r}) = \bar{r}' \quad \text{and} \quad t(\bar{f}_0'(x, y, \bar{r})) \le t(\bar{f}_0(x \oplus y)) \tag{4}$$

and thus, together with (3), we have

$$1st(\bar{f}_0'(x, y, \bar{r})) = 1st(\bar{f}_0(x \oplus y)) = f_0(x \oplus y). \tag{5}$$

Stage III generates a function $\hat{f}_0$, 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 $\Pi(\bar{r})$ projects out 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 $f_0(x \oplus y)$.[2] In particular, if $f_0(x) = r$, then

$$1st(\hat{f}_0(x)) = r \quad \text{and} \quad t(\hat{f}_0(x)) \le t(f_0(x)). \tag{6}$$

Additionally, we obtain a function $\hat{f}_0'$, 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 $\Pi(\bar{r})$ as above, returns $\Pi(\bar{r}')$. This pruning is possible because $\Pi(\bar{r}')$ depends only on $\Pi(\bar{r})$, which can be easily shown using the transitivity above. With the relationship between $\hat{f}_0$ and $\bar{f}_0$, together with (3) and (4), we can prove that if $f_0(x) = r$, then we have if $\hat{f}_0(x) = \hat{r}$ and $f_0(x \oplus y) = r'$, then

$$\hat{f}_0'(x, y, \hat{r}) = \hat{f}_0(x \oplus y) \quad \text{and} \quad t(\hat{f}_0'(x, y, \hat{r})) \le t(f_0(x \oplus y)) \tag{7}$$

and thus, together with (6), we have

$$1st(\hat{f}_0'(x, y, \hat{r})) = 1st(\hat{f}_0(x \oplus y)) = r'. \tag{8}$$

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. Therefore, we do not have to conduct a derivation on $\hat{f}_0$ and $\oplus$ to obtain such an incremental function.

At the end, putting (6), (7), and (8) together, we have if $f_0(x) = r$, then

$$1st(\hat{f}_0(x)) = r \quad \text{and} \quad t(\hat{f}_0(x)) \le t(f_0(x)) \tag{9}$$

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

$$1st(\hat{f}_0'(x, y, \hat{r})) = r', \quad \hat{f}_0'(x, y, \hat{r}) = \hat{f}_0(x \oplus y), \quad \text{and} \quad t(\hat{f}_0'(x, y, \hat{r})) \le t(f_0(x \oplus y)). \tag{10}$$

i.e, the functions $\hat{f}_0$ and $\hat{f}_0'$ preserve the semantics and compute asymptotically at least as fast. 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 $f_0(x \oplus y)$ due to the transformations used in Stages II and III.

[2] Note that this is different from the partial dead code elimination in [20], where partial dead code refers to code that is dead on some but not all computation paths.

4 Stage I: Caching All Intermediate Results

Stage I transforms the program for f_0 to embed all intermediate results in the final return value. It consists of a straightforward extension transformation and administrative simplifications. Optimizations to this process are possible. Certain improvements can also be made.

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

$$\bar{f}(v_1, ..., v_n) = \mathcal{E}xt[\![e]\!] \tag{11}$$

where $\mathcal{E}xt[\![e]\!]$ extends an expression e to return the values of all function calls made in computing e, i.e., it considers subexpressions of e in applicative and left-to-right order, introduces bindings that name the results of function calls, builds up tuples of these values together with the values of the original subexpressions, and passes these values from subcomputations to enclosing computations.

The definition of $\mathcal{E}xt$ is given in Figure 3. We assume that each introduced binding uses a fresh variable name. For transforming a conditional expression, the transformation $\mathcal{G}us[\![e]\!]$ generates a tuple of _'s of length equal to the number of the function applications in e, where _ is a dummy constant that just occupies a spot. The lengths of the tuples generated by $\mathcal{G}us[\![e_2]\!]$ and $\mathcal{G}us[\![e_3]\!]$ can easily be determined statically. Actually, they are just the lengths of $rst(v_2)$ and $rst(v_3)$, respectively. This mechanism assures that the extended function returns a uniform tuple no matter what the value of the Boolean expression is, which makes the pruning stage simpler, since we do not have to consider pruning differently under different conditions.

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

Figure 3: Definition of $\mathcal{E}xt$

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

$$1st(\bar{f}(v_1, ..., v_n)) = f(v_1, ..., v_n). \tag{12}$$

The result of this transformation is a set of extended function definitions that straightforwardly embed the values of all function calls in the return values. For the functions *foo* and *boo* in Figure 1, after the extension transformation, we obtain the functions $\overline{foo}_1$ and $\overline{boo}_1$ as follows:

$$
\begin{array}{ll}
\overline{foo}_1(x) = & \mathbf{let}\ v_1 = \mathbf{let}\ v_{11} = < x > \ \mathbf{in}\ \mathbf{let}\ v_{12} = < 2 > \ \mathbf{in} \\
& \qquad\qquad < 1st(v_{11}) \leq 1st(v_{12}) > @rst(v_{11})@rst(v_{12}) \quad \mathbf{in} \\
& \mathbf{if}\ 1st(v_1)\ \ \mathbf{then}\ \ \mathbf{let}\ v_2 = < 1 > \ \mathbf{in} \\
& \qquad\qquad\qquad\qquad < 1st(v_2) > @rst(v_1)@rst(v_2)@ < \_, \_ > \\
& \qquad\qquad\qquad \mathbf{else}\ \ \mathbf{let}\ v_3 = \mathbf{let}\ v_{31} = \mathbf{let}\ v_{311} = < x > \ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ u_1 = \overline{boo}_1(1st(v_{311}))\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad < 1st(u_1), u_1 > @rst(v_{311}) \quad \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ v_{32} = \mathbf{let}\ v_{321} = \mathbf{let}\ v_{3211} = < x > \ \mathbf{in}\ \mathbf{let}\ v_{3212} = < 3 > \ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad < 1st(v_{3211}) - 1st(v_{3212})@rst(v_{3211})@rst(v_{3212}) \quad \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ u_2 = \overline{foo}_1(1st(v_{321}))\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad < 1st(u_2), u_2 > @rst(v_{321}) \quad \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad < 1st(v_{31}) + 1st(v_{32}) > @rst(v_{31})@rst(v_{32}) \quad \mathbf{in} \\
& \qquad\qquad\qquad\qquad < 1st(v_3) > @rst(v_1)@ <> @rst(v_3)
\end{array}
\tag{13}
$$

$$
\begin{array}{ll}
\overline{boo}_1(x) = & \mathbf{let}\ v_1 = \mathbf{let}\ v_{11} = \mathbf{let}\ v_{111} = < x > \ \mathbf{in}\ \mathbf{let}\ v_{112} = < 1 > \ \mathbf{in} \\
& \qquad\qquad\qquad\qquad < 1st(v_{111}) - 1st(v_{112})@rst(v_{111})@rst(v_{112}) \quad \mathbf{in} \\
& \qquad\qquad \mathbf{let}\ u_1 = \overline{foo}_1(1st(v_{11}))\ \mathbf{in} \\
& \qquad\qquad < 1st(u_1), u_1 > @rst(v_{11}) \quad \mathbf{in} \\
& \mathbf{let}\ v_2 = \mathbf{let}\ v_{21} = \mathbf{let}\ v_{211} = < x > \ \mathbf{in}\ \mathbf{let}\ v_{212} = < 2 > \ \mathbf{in} \\
& \qquad\qquad\qquad\qquad < 1st(v_{211}) - 1st(v_{212})@rst(v_{211})@rst(v_{212}) \quad \mathbf{in} \\
& \qquad\qquad \mathbf{let}\ u_2 = \overline{foo}_1(1st(v_{21}))\ \mathbf{in} \\
& \qquad\qquad < 1st(u_2), u_2 > @rst(v_{21}) \quad \mathbf{in} \\
& < 1st(v_1) + 1st(v_2) > @rst(v_1)@rst(v_2)
\end{array}
$$

The straightforward extension implemented by this transformation 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.

Administrative Simplification. We then perform administrative simplifications to *clean* up the resulting program obtained above. For each function definition $f(v_1, ..., v_n) = e$ obtained from the extension transformation, we obtain a function definition

$$f(v_1, ..., v_n) = \mathcal{C}lean[\![e]\!]\,\emptyset \tag{14}$$

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 whenever possible to enhance readability.

An information set $I_{[e]}$ at the occurrence of an expression e is a set of equations collected from the bindings introduced in the context of e. We write $e_1 \leftrightarrow e_2$ to denote that two expressions e_1 and e_2 are equal. For example, if some $f(v_1, ..., v_n)$ is defined to be e, and e is $\mathbf{let}\ v_1 = e_1\ \mathbf{in}\ \mathbf{let}\ v_2 = e_2\ \mathbf{in}\ e_3$, 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 like tuple operations and binding unfolding, as summarized in Figure 4. Given an expression e and an information set I, we say that e can be simplified to e' under I if the corresponding condition $cond(I)$ holds, and we define $\mathcal{S}imp_{\mathcal{C}lean}[\![e]\!]I = e'$; otherwise, we define $\mathcal{S}imp_{\mathcal{C}lean}[\![e]\!]I = e$.

$\mathcal{C}lean$ 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. We just present the case of $\mathcal{S}ubl_{\mathcal{C}lean}$ working on the ith subexpression of the top-level construct and condition it on whether the

expression e	expression e'	condition $cond(I)$
$e_1@e_2$	$<e_{11},...,e_{1n_1},e_{21},...,e_{2n_2}>$	$e_1 \leftrightarrow <e_{11},...,e_{1n_1}> \in I$ and $e_2 \leftrightarrow <e_{21},...,e_{2n_2}> \in I$
$1st(e)$	e_1	$e \leftrightarrow <e_1,e_2,...,e_n> \in I$
$rst(e)$	$<e_2,...,e_n>$	$e \leftrightarrow <e_1,e_2,...,e_n> \in I$
let $v = e_1$ **in** e_2	$e_2[e_1/v]$	v is introduced by the ext. transformation and occurs at most once in e_2

Figure 4: Simplification

subexpressions 1 through $i-1$ have been *reduced*. Operationally, we say that a subexpression is reduced if it is the result of having already applied $\mathcal{C}lean$ for the subexpression at that position; otherwise, it is not reduced. 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 expression, then $\mathcal{S}ubl_{\mathcal{C}lean}$ 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 into the information set.

$$
\begin{array}{ll}
\mathcal{S}ubl_{\mathcal{C}lean}[\![g(e_1,...,e_n)]\!]\ I \quad \text{where } g \text{ is } c,\ p, \text{ or } f & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![g(e_1,...,e_{i-1},e_i',e_{i+1},...,e_n)]\!]\ I & \text{if } e_1,...,e_{i-1} \text{ are reduced, not } \textbf{let}\text{, but} \\
\quad\quad \text{where } e_i' = \mathcal{C}lean[\![e_i]\!]\ I & \quad e_i \text{ is not reduced} \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1' \textbf{ in } g(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)]\!]\ I & \text{if } e_1,...,e_{i-1} \text{ are reduced, not } \textbf{let}, \\
\quad\quad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \quad e_i \text{ is reduced, but } e_i \text{ is } \textbf{let } v = e_1' \textbf{ in } e_2' \\
\quad = g(e_1,...,e_n) & \text{otherwise} \\
\\
\mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I & \text{if } e_1 \text{ is not reduced} \\
\quad\quad \text{where } e_1' = \mathcal{C}lean[\![e_1]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1' \textbf{ in if } e_2' \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I & \text{if } e_1 \text{ is reduced, and } e_1 \text{ is } \textbf{let } v = e_1' \textbf{ in } e_2' \\
\quad\quad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \\
\quad = \textbf{if } e_1 \textbf{ then } \mathcal{C}lean[\![e_2]\!]\ I \textbf{ else } \mathcal{C}lean[\![e_3]\!]\ I & \text{otherwise} \\
\\
\mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v = e_1' \textbf{ in } e_2]\!]\ I & \text{if } e_1 \text{ is not reduced} \\
\quad\quad \text{where } e_1' = \mathcal{C}lean[\![e_1]\!]\ I & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{let } v' = e_1' \textbf{ in let } v = e_2' \textbf{ in } e_2]\!]\ I & \text{if } e_1 \text{ is reduced, and } e_1 \text{ is } \textbf{let } v' = e_1' \textbf{ in } e_2' \\
\quad\quad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \\
\quad = \mathcal{S}ubl_{\mathcal{C}lean}[\![\textbf{if } e_1' \textbf{ then let } v = e_2' \textbf{ in } e_2 \textbf{ else let } v = e_3' \textbf{ in } e_2]\!]\ I & \text{if } e_1 \text{ is reduced, and } e_1 \text{ is } \textbf{if } e_1' \textbf{ then } e_2' \textbf{ else } e_3' \\
\quad\quad \text{where } e_1' \text{ is reduced and is not } \textbf{let} & \\
\quad = \textbf{let } v = e_1 \textbf{ in } \mathcal{C}lean[\![e_2]\!]\ I' & \text{otherwise} \\
\quad\quad \text{where } I' = I \cup \{v \leftrightarrow e_1\} \text{ if } v \text{ is introduced, and } I' = I \text{ otherwise} &
\end{array}
$$

Figure 5: Definition of $\mathcal{S}ubl_{\mathcal{C}lean}$

Finally, we define the function $\mathcal{C}lean$ as in (15). 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.

$$
\mathcal{C}lean[\![e]\!]I = e'', \quad \text{where} \quad e'' = \mathcal{S}imp_{\mathcal{C}lean}[\![e']\!]I \quad \text{and} \quad e' = \begin{cases} \mathcal{S}ubl_{\mathcal{C}lean}[\![e]\!]I & \text{if } e \text{ is not } v \\ e & \text{otherwise} \end{cases} \tag{15}
$$

$\mathcal{C}lean$ cleans out only some of the bindings introduced by the extension transformation and lifts some bindings and conditions. The resulting functions $\bar{f}$ still satisfy the properties stated around (12), i.e., the formula (3) holds.

After cleaning, we obtain a set of extended function definitions that are simpler, easier to read, and also easier for the subsequent stages to process. For the functions $\overline{foo}_1$ and $\overline{boo}_1$ in (13), after the cleaning transformation, we obtain the functions $\overline{foo}$ and $\overline{boo}$ as follows:

$$\begin{array}{ll}\overline{foo}(x) = & \textbf{if } x \leq 2 \textbf{ then } <1,\_,\_> \\ & \textbf{else let } u_1 = \overline{boo}(x) \textbf{ in} \\ & \quad \textbf{let } u_2 = \overline{foo}(x-3) \textbf{ in} \\ & \quad <1st(u_1){+}1st(u_2),\ u_1,\ u_2>\end{array} \qquad \begin{array}{ll}\overline{boo}(x) = & \textbf{let } u_1 = \overline{foo}(x-1) \textbf{ in} \\ & \textbf{let } u_2 = \overline{foo}(x-2) \textbf{ in} \\ & <1st(u_1){+}1st(u_2),\ u_1,\ u_2>\end{array} \tag{16}$$

Optimizations. An obvious optimization can be incorporated into the extension transformation, i.e, we can introduce bindings only for subexpressions that contain function applications. Thus, there would be fewer tuple operations for passing intermediate results and fewer bindings to be unwound or lifted, leaving less work for the administrative simplifications.

To do this, we replace the transformation $\mathcal{E}xt$ with the transformation $\mathcal{E}xt1$ in Figure 6. The notion of *reduced* for $\mathcal{E}xt1$ is similar to that for $\mathcal{S}ubl_{\mathcal{C}lean}$. We use $cf(e)$ to denote that the expression e contains a function application, and $ncf(e)$ to denote that e contains no function application.

$$\begin{array}{ll}
\mathcal{E}xt1[\![g(e_1,...,e_n)]\!]\ A \quad \text{where } g \text{ is } c,\ p, \text{ or } f & \\
\quad = \textbf{let } v_i = \mathcal{E}xt1[\![e_i]\!]\ \emptyset \textbf{ in } \mathcal{E}xt1[\![g(e_1,...,e_n)]\!]\ (A \cup \{\langle v_i, e_i\rangle\}) & \text{if } e_1,...,e_{i-1} \text{ are reduced, } e_i \text{ is not, and } cf(e_i) \\
\quad = \mathcal{E}xt1[\![g(e_1,...,e_n)]\!]\ A & \text{if } e_1,...,e_{i-1} \text{ are reduced, } e_i \text{ is not, and } ncf(e_i) \\
\quad = <g(e'_1,...,e'_n)> @e''_1@...@e''_n & \text{if } e_1,...,e_n \text{ are reduced, and } g \text{ is } c \text{ or } p \\
\quad = \textbf{let } v = \bar{g}(e'_1,...,e'_n) \textbf{ in } \ <1st(v)> @e''_1@...@e''_n@<v> & \text{otherwise, i.e., if } e_1,...,e_n \text{ are reduced, and } g \text{ is } f \\
\qquad \text{where } e'_i = \begin{cases} 1st(v_i) & \text{if } \langle v_i, e_i\rangle \in A \\ e_i & \text{otherwise} \end{cases} \ \text{and } e''_i = \begin{cases} rst(v_i) & \text{if } \langle v_i, e_i\rangle \in A \\ <> & \text{otherwise} \end{cases} \ \text{for } i = 1..n &
\end{array}$$

$$\begin{array}{ll}
\mathcal{E}xt1[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ A & \\
\quad = \textbf{let } v_1 = \mathcal{E}xt1[\![e_1]\!]\ \emptyset \textbf{ in } \mathcal{E}xt1[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ \{\langle v_1, e_1\rangle\} & \text{if } e_1 \text{ is not reduced, and } cf(e_1) \\
\quad = \mathcal{E}xt1[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ \emptyset & \text{if } e_1 \text{ is not reduced, and } ncf(e_1) \\
\quad = \textbf{if } e'_1 \textbf{ then } e^*_2 \textbf{ else } e^*_3 & \text{otherwise} \\
\qquad \text{where } e^*_2 = \begin{cases} \textbf{let } v_2 = \mathcal{E}xt1[\![e_2]\!]\ \emptyset \textbf{ in } \ <1st(v_2)> @e''_1@rst(v_2)@\mathcal{G}us[\![e_3]\!] & \text{if } cf(e_2) \\ <e_2> @e''_1@\mathcal{G}us[\![e_3]\!] & \text{otherwise} \end{cases} & \\
\qquad\qquad\quad e^*_3 = \begin{cases} \textbf{let } v_3 = \mathcal{E}xt1[\![e_3]\!]\ \emptyset \textbf{ in } \ <1st(v_3)> @e''_1@\mathcal{G}us[\![e_2]\!]@rst(v_3) & \text{if } cf(e_3) \\ <e_3> @e''_1@\mathcal{G}us[\![e_2]\!] & \text{otherwise} \end{cases} & \\
\qquad \text{where } e'_1 = \begin{cases} 1st(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ e_1 & \text{otherwise} \end{cases} \ \text{and } e''_1 = \begin{cases} rst(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ <> & \text{otherwise} \end{cases} &
\end{array}$$

$$\begin{array}{ll}
\mathcal{E}xt1[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ A & \\
\quad = \textbf{let } v_1 = \mathcal{E}xt1[\![e_1]\!]\ \emptyset \textbf{ in } \mathcal{E}xt1[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ \{\langle v_1, e_1\rangle\} & \text{if } e_1 \text{ is not reduced, and } cf(e_1) \\
\quad = \mathcal{E}xt1[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ \emptyset & \text{if } e_1 \text{ is not reduced, and } ncf(e_1) \\
\quad = \textbf{let } v = e'_1 \textbf{ in } e^*_2 & \text{otherwise} \\
\qquad \text{where } e^*_2 = \begin{cases} \textbf{let } v_2 = \mathcal{E}xt1[\![e_2]\!]\ \emptyset \textbf{ in } \ <1st(v_2)> @e''_1@rst(v_2) & \text{if } cf(e_2) \\ <e_2> @e''_1 & \text{otherwise} \end{cases} & \\
\qquad \text{where } e'_1 = \begin{cases} 1st(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ e_1 & \text{otherwise} \end{cases} \ \text{and } e''_1 = \begin{cases} rst(v_1) & \text{if } \langle v_1, e_1\rangle \in A \\ <> & \text{otherwise} \end{cases} &
\end{array}$$

Figure 6: Definition of $\mathcal{E}xt1$

For the functions foo and boo in Figure 1, after the optimized extension transformation, we obtain the functions $\overline{foo}_2$ and $\overline{boo}_2$ as follows:

$$
\begin{array}{ll}
\overline{foo}_2(x) = & \textbf{if } x \leq 2 \textbf{ then } <1, \_, \_> \\
& \textbf{else let } v_3 = \textbf{ let } v_{31} = \textbf{ let } u_1 = \overline{boo}_2(x) \textbf{ in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad <1st(u_1), u_1> @ <> \quad \textbf{in} \\
& \qquad\qquad\qquad\quad \textbf{let } v_{32} = \textbf{ let } u_2 = \overline{foo}_2(x-3) \textbf{ in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad <1st(u_2), u_2> @ <> \quad \textbf{in} \\
& \qquad\qquad\qquad\quad <1st(v_{31}) + 1st(v_{32})> @rst(v_{31})@rst(v_{32}) \quad \textbf{in} \\
& \qquad\quad <1st(v_3)> @ <> @rst(v_3) \\
\\
\overline{boo}_2(x) = & \textbf{let } v_1 = \textbf{ let } u_1 = \overline{foo}_2(x-1) \textbf{ in} \\
& \qquad\qquad <1st(u_1), u_1> @ <> \quad \textbf{in} \\
& \textbf{let } v_2 = \textbf{ let } u_2 = \overline{foo}_2(x-2) \textbf{ in} \\
& \qquad\qquad <1st(u_2), u_2> @ <> \quad \textbf{in} \\
& <1st(v_1) + 1st(v_2)> @rst(v_1)@rst(v_2)
\end{array}
\tag{17}
$$

Then, after the cleaning transformation on them, we obtain the same functions $\overline{foo}$ and $\overline{boo}$ as in (16).

Improvements. We can make certain improvements to the above brute-force caching of all intermediate results. We discuss three of them.

First, before applying the extension transformation, we can lift common subcomputations in both branches of a conditional expression. This simplifies programs in general. Common subcomputations appear only once instead of twice in two branches of a conditional expression. Also, it is easier to reason about the results of these subcomputations, since they are independent of the value of the condition. For caching all intermediate results, this lifting saves the extension transformation from caching these common subcomputations in difference components under different conditions, which makes it easier to reason about using these values for incremental computation.

Second, we can avoid caching values of function applications that are embedded in the values of their enclosing applications, since these omitted values can be retrieved from the results of the enclosing computations.

To do this, we first compute the value-embedding relations. We use $Mf(f, i)$ to indicate whether the value of v_i is embedded in the value of $f(v_1, ..., v_n)$, and we use $Me(e, v)$ to indicate whether the value of variable v is embedded in the value of expression e. They must satisfy the following safety requirements:

$$
\begin{array}{l}
\text{if } Mf(f,i) = true, \text{ then } \exists f_i^{-1} \text{ such that, if } u = f(v_1, ..., v_n), \text{ then } v_i = f_i^{-1}(u) \\
\text{if } Me(e,v) = true, \text{ then } \exists e_v^{-1} \text{ such that, if } u = e, \text{ then } v = e_v^{-1}(u)
\end{array}
\tag{18}
$$

For each function definition $f(v_1, ..., v_n) = e_f$, we define $Mf(f, i) = Me(e_f, v_i)$, and we define Me as in Figure 7, where it may refer to $Mf(f, i)$ recursively. For a primitive function application, $\exists p_i^{-1}$ denotes $true$ if p has an inverse for the ith argument, and $false$ otherwise. For a conditional expression, $in_{if}^{e_1}$ denotes $true$ if the value of e_1 can be determined statically or inferred from the value of **if** e_1 **then** e_1 **else** e_3, and $false$ otherwise. For example, $in_{if}^{e_1}$ is true if e_1 is just T (for true) or F (for false), or if the two branches of the conditional expression return applications of two different constructors. We can easily show, by recursion induction based on each rule, that the safety requirements are satisfied. To compute $Mf(f, i)$ for any f and i, we start with $Mf(f, i) = true$ for every f and i and iterate using the above definitions to compute the greatest fixed point (assuming $false \sqsubseteq true$). The iteration always terminates since these definitions are monotonic and the true-false domain is finite.

Then, for each function definition $f(v_1, ..., v_n) = e_f$, we associate an embedding tag $Mtag$ with each subexpression e of e_f indicating whether the value of e is embedded in the value of e_f. These tags can be defined in a similar fashion to the above definitions: We define $Mtag(e_f) = true$, and define the $true$

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

Figure 7: Definition of Me

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

Figure 8: Definition of $Mtag$

values of $Mtag$ for subexpressions e of e_f as in Figure 8; the value of other subexpressions of e_f are defined to be $false$. These tags can be computed directly once the above embedding relations are computed. With these embedding tags, we can compute, for each function definition $f(v_1,...,v_n) = e_f$, whether all intermediate results of e_f are embedded in the value of e_f, and we use $Mall_f$ to denote this. We define, for each $f(v_1,...,v_n) = e_f$,

$$
Mall_f = \bigwedge_{\text{all function applications } g(e_1,...,e_n) \text{ occurring in } e_f} Mtag(g(e_1,...,e_m)) \wedge Mall_g \tag{19}
$$

where $Mtag$ is with respect to the value of e_f. To compute $Mall_f$ for any f, we start with $Mall_f = true$ for all f and iterate using the definitions in (19).

Now, we modify the extension transformation to omit caching the value of a function application all of whose intermediate results (including the value of the application itself) are embedded in the value of its enclosing application. For the unoptimized transformation $\mathcal{E}xt$ in Figure 3, everything remains the same except for a function application $f(e_1,...,e_n)$ when both $Mall_f = true$ and $Mtag(f(e_1,...,e_n)) = true$:

$$
\begin{array}{rl}
\mathcal{E}xt[\![f(e_1,...,e_n)]\!] = & \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } \cdots \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\
& < f(1st(v_1),\ldots,1st(v_n)) > @rst(v_1)@\cdots@rst(v_n)
\end{array} \tag{20}
$$

Similarly, for the optimized transformation $\mathcal{E}xt1$ in Figure 6, under this condition, we define

$$
\begin{array}{ll}
\mathcal{E}xt1[\![g(e_1,...,e_n)]\!] & \\
\quad = < g(e_1',\ldots,e_n')) > @e_1''@\cdots@e_n'' & \text{if } e_1,...,e_n \text{ are reduced, and } g \text{ is } f
\end{array} \tag{21}
$$

With this improvement, we can obtain that, if caching all intermediate results for a set F of function definitions gives us a set $\bar{F}$ of function definitions, then caching all intermediate results again for $\bar{F}$ would give us the same set $\bar{F}$, i.e., the transformation for caching all intermediate results is idempotent.

The third improvement is to avoid making an extended version for a function f all of whose intermediate results are embedded in the value of f (i.e., $Mall_f = true$, which includes the case where f does not contain function applications), and reference the return value of f directly rather than having to refer to the first component of the extended version. Therefore, for the transformation $\mathcal{E}xt$ on a function application $f(e_1, ..., e_n)$, if $Mall_f = true$ and $Mtag(f(e_1, ..., e_n)) = true$, we define $\mathcal{E}xt$ as in (20); else if $Mall_f = true$ but $Mtag(f(e_1, ..., e_n)) = false$, we define

$$\mathcal{E}xt[\![f(e_1, ..., e_n)]\!] = \begin{array}{l} \textbf{let } v_1 = \mathcal{E}xt[\![e_1]\!] \textbf{ in } \cdots \textbf{ let } v_n = \mathcal{E}xt[\![e_n]\!] \textbf{ in} \\ \textbf{let } v = f(1st(v_1), \ldots, 1st(v_n)) \textbf{ in} \\ < v > @rst(v_1)@ \cdots @rst(v_n)@ < v > \end{array} \tag{22}$$

otherwise we define $\mathcal{E}xt$ as in Figure 3. Similar modifications can be made to the transformation $\mathcal{E}xt1$.

For the functions foo and boo of our running example, these improvements would eventually give us the same functions $\overline{foo}$ and $\overline{boo}$ as in (16).

5 Stage II: Incrementalization

Stage II derives a function $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$. Basically, one may identify subcomputations in the expanded $\bar{f}_0(x \oplus y)$ whose values can be retrieved from the cached result $\bar{r}$ of $\bar{f}_0(x)$, replace them by corresponding retrievals, and capture the resulting way of computing $\bar{f}_0(x \oplus y)$ in the incremental version $\bar{f}_0'(x, y, \bar{r})$. Such a derivation method is given in [26], and, depending on the power one expects from the derivation, the method can be made semi-automatic or fully-automatic.

The details of the derivation approach are not the subject of this paper. However, two concerns specific to the prune-and-cache method and relating the different stages are addressed here.

First, secondary to the goal of making the incremental program $\bar{f}_0'(x, y, \bar{r})$ as fast as possible, we want to make it use as few different intermediate results in $\bar{r}$ as possible. To do this, we require that the derivation not use intermediate results that are embedded in the results of enclosing computations so that the unused intermediate results can be pruned out by Stage III. The second improvement in Stage I avoids caching intermediate results that can be statically determined to be embedded in the results of enclosing computations. We may do better by addressing the issue also in Stage II, where we may have more powerful reasoning support.

Second, not only do we want $\bar{f}_0'(x, y, \bar{r})$ to be no slower than $f_0(x \oplus y)$, as can be guaranteed with the approach in [26], but we also want it to be no slower than $f_0'(x, y, r)$. To assure this, we require that the derivation replace a subcomputation in the expanded $\bar{f}_0(x \oplus y)$ by a retrieval from an intermediate result in $\bar{r}$ other than $1st(\bar{r})$ only if the subcomputation is also a subcomputation in $\bar{f}_0(x)$.[3] This requirement helps assure that caching intermediate results is worthwhile, i.e., the time spent in maintaining intermediate results will not surpass that saved by using them, as will be explained in Section 6.1.

Consider the function $\overline{foo}$ that caches all intermediate results of foo in (16). To derive an incremental version of $\overline{foo}$ under $\oplus_1$ using [26], we transform $\overline{foo}(x \oplus_1 y) = \overline{foo}(x + 1)$, with $\overline{foo}(x) = \bar{r}$:

1. unfold $\overline{foo}(x{+}1)$, simplify primitive operations on $x{+}1$

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

2. separate cases, replace applications of $\overline{foo}$ by retrievals

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

[3] In practice, this is the best any derivation method could do without the power of a general theorem prover [26].

and we obtain an incremental function $\overline{foo}'$ such that, if $\overline{foo}(x) = \bar{r}$, then $\overline{foo}'(x, \bar{r}) = \overline{foo}(x+1)$, as follows:

$$\begin{array}{ll} \overline{foo}'(x,\bar{r}) = & \textbf{if } x \leq 1 \textbf{ then } <1,\_,\_> \\ & \textbf{else if } x = 2 \textbf{ then } <3, \;\; <2,<1,\_,\_>,<1,\_,\_>>, \;\; <1,\_,\_>> \\ & \textbf{else } <1st(\bar{r})+1st(2nd(\bar{r})), \;\; <1st(\bar{r})+1st(2nd(2nd(\bar{r}))),\; \bar{r},\; 2nd(2nd(\bar{r}))>, \;\; 3rd(2nd(\bar{r}))> \end{array} \tag{23}$$

Clearly, $\overline{foo}'(x, \bar{r})$ computes $\overline{foo}(x + 1)$ in only $O(1)$ time.

6 Stage III: Pruning

The input to the pruning stage is $\bar{f}_0$, a function that caches all intermediate results of f_0 obtained from Stage I, and $\bar{f}_0'$, an incremental version of $\bar{f}_0$ under $\oplus$ obtained from Stage II, together with a set of other function definitions used in computing $\bar{f}_0$ and $\bar{f}_0'$, obtained from Stage I and II. The goal is to prune $\bar{f}_0$, so that it returns only the value of f_0 and the intermediate results useful for incremental computation under $\oplus$, and prune $\bar{f}_0'$, so that it incrementally computes only the value of f_0 and the useful intermediate results.

To achieve this goal, we analyze the function $\bar{f}_0'$ to determine the components of $\bar{r}$, the value of $\bar{f}_0(x)$, on which $1st(\bar{f}_0'(x, y, \bar{r}))$, the value of $f_0(x \oplus y)$, transitively depends. Two issues arise as we need to maintain these components: transitive dependencies and cost. We first depict the transitive dependencies and address the cost issue. Then we give an algorithm that computes the needed components based on a dependency analysis using domain projections [41, 12]. With this result, we prune the function $\bar{f}_0$ to return only the intermediate results that are useful for computing $1st(\bar{f}_0'(x, y, \bar{r}))$. In addition, we prune the function $\bar{f}_0'$ to incrementally maintain only the useful intermediate results.

6.1 Maintaining Intermediate Results: Transitive Dependency and Cost

Transitive Dependency. The function application $\bar{f}_0'(x, y, \bar{r})$ returns the value of $f_0(x \oplus y)$ in the first component and all corresponding intermediate results in the other components. To determine which components in the value $\bar{r}$ are needed for incremental computation, we start with the first component in the value of $\bar{f}_0'(x, y, \bar{r})$, the value of $f_0(x \oplus y)$, and find out the components of $\bar{r}$ on which this value depends. These components may include those other than the first one of $\bar{r}$. To support incremental computation after further input changes, we need to maintain these components of $\bar{f}_0'(x, y, \bar{r})$ as well as the first component. They may depend on even other components of $\bar{r}$, forming a kind of transitive dependency.

Figure 9 illustrates the transitive dependencies for the example foo under change $\oplus_1$. By definitions of foo and boo and associativity of '+', we have

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

Thus, to compute the value v_1' of $foo(x{+}1)$, $\overline{foo}'$ uses the value v_1 of $foo(x)$ and the intermediate result v_2 of $boo(x)$ returned by $\overline{foo}(x)$. Therefore, the corresponding value v_1' of $foo(x{+}1)$ and the intermediate result v_2' of $boo(x+1)$ need to be maintained. The value v_1' of $foo(x+1)$ has just been considered. To compute the intermediate result v_2' of $boo(x+1)$, $\overline{foo}'$ uses the value v_1 of $foo(x)$ and the intermediate result v_3 of $foo(x-1)$ returned by $\overline{foo}(x)$. Therefore, the corresponding value v_1' of $foo(x+1)$ and the intermediate result v_3' of $foo(x)$ also need to be maintained. Again, the value v_1' of $foo(x{+}1)$ has just been considered. To compute the value v_3' of $foo(x)$, $\overline{foo}'$ just uses the value v_1 of $foo(x)$ returned by $\overline{foo}(x)$.

Thus, to summarize, the value v_1' of $foo(x{+}1)$ transitively depends on the components of intermediate results corresponding to 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 components of intermediate results are not needed and therefore do not need to be computed or maintained; they can be be pruned out.

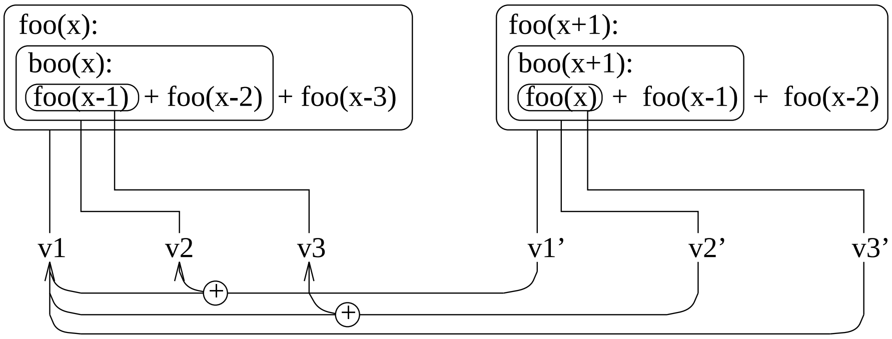

Figure 9: Dependencies for the function foo

Cost. Is it always true that the time spent in maintaining intermediate results will not surpass that saved by using them?

First, we consider the problem in general. Given a way of computing a function f, let g be a function that computes some intermediate results of f in the way f does, and let $\hat{f} = \langle f, g \rangle$. Suppose $f'(x, y, r)$ computes $f(x \oplus y)$ given $r = f(x)$, and $\hat{f}'(x, y, r)$ computes $\hat{f}(x \oplus y)$ given $\hat{r} = \hat{f}(x)$. Then in general, it is not true that $t(\hat{f}'(x, y, \hat{r})) \leq t(f'(x, y, r))$. This is mainly because f, and thus g, could be arbitrary. This is true even if all these functions compute with the best asymptotic time. What it says is that maintaining arbitrary intermediate results may not be worthwhile for incremental computation.

But, consider the particular functions f' and $\hat{f}'$ derived using the derivation approach. Recall the second requirement in incrementalization: the derivation replaces a subcomputation in the expanded $\hat{f}_0(x \oplus y)$ by a retrieval from an intermediate result in $\hat{r}$ other than $1st(\hat{r})$ only if the subcomputation is also a subcomputation in $\hat{f}_0(x)$. Thus, suppose we compute $\hat{f}(x \oplus y)$ using the cached result $\hat{r}$ of $\hat{f}(x)$, and suppose computing $1st(\hat{f}(x \oplus y))$, i.e., $f(x \oplus y)$, uses a subcomputation $g(x)$ in $f(x)$, and the value of $g(x)$ can be retrieved from $\hat{r}$ but not $1st(\hat{r})$, i.e., r. Then, on the one hand, the value of $g(x \oplus y)$ needs to be maintained by $\hat{f}'(x, y, \hat{r})$; on the other hand, if we compute $f(x \oplus y)$ using only the cached result r of $f(x)$, then the subcomputation $g(x)$ remains in $f'(x, y, r)$, i.e., $f'(x, y, r)$ has the cost of recomputing $g(x)$.

Now, for intermediate results of f like the value of g above, 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 polynomial time or exponential time but with linear exponent, then we have

$$t(\hat{f}'(x, y, \hat{r})) \leq t(f'(x, y, r)). \tag{24}$$

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 (24), we notice that

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

We conclude that, with the conditions above, using and maintaining intermediate results is always asymptotically at least as fast. Therefore, in order to achieve as fast incremental computation as possible, we should compute the closure of the transitive dependencies for maintaining intermediate results.

6.2 Dependency Analysis Using Projections

We first describe our use of projections 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 components of $\bar{r}$ are needed for computing certain 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 of interest D contains $\perp$, indicating a computation diverges, values d returned by functions in the original program for 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 bounded, since it is dynamically determined. 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. If a subcomputation involves $\_$, then the result of that subcomputation is $\_$, but the result of the parent computation need not be $\_$. For any value d in domain D, $\perp \sqsubseteq d$. For two values d_1 and d_2 other than $\perp$'s in D, $d_1 \sqsubseteq d_2$ iff

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

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

A non-bottom projection Π of interest here can be represented as a set of selection functions π, each of which is a sequence of 1^{st}, 2^{nd}, ..., n^{th}. 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 $\Pi_{(i)}$ to denote the set $\{\pi \mid \pi\, i^{th} \in \Pi\}$, i.e., $\Pi_{(i)}$ is the part of Π that considers the ith component. With the set representation, a projection $\Pi = ID$ iff $\epsilon \in \Pi$ or $\Pi_{(i)} = ID$ for $i = 1..n$ for arguments of Π of length n. A projection $\Pi = ABS$ iff $\Pi = \emptyset$. 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's, $\Pi_1 \sqsubseteq \Pi_2$ iff

$$\Pi_1 = ABS, \quad \Pi_2 = ID, \quad \text{or} \quad \Pi_{1(i)} \sqsubseteq \Pi_{2(i)} \text{ for } i = 1..n \text{ for arguments of } \Pi_1 \text{ and } \Pi_2 \text{ of length } n.$$

Dependency Analysis. To compute which components of $\bar{r}$ are needed for computing certain components of $\bar{f_0}'(x, y, \bar{r})$, we apply a backward dependency analysis to the program for $\bar{f_0}'$.

Following the style of [46], 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 must satisfy is: if $\Pi_i = f^i\, \Pi$ then

$$\Pi(f(v_1, ..., v_i, ..., v_n)) \sqsubseteq f(v_1, ..., \Pi_i(v_i), ..., v_n) \tag{25}$$

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

$$\Pi(e) \sqsubseteq e[\Pi'(v)/v] \tag{26}$$

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

$$e^v\,BOT = BOT \qquad \text{and} \qquad e^v\,ABS = ABS. \tag{27}$$

For $\Pi \neq BOT, ABS$, we give the definition of $e^v\,\Pi$ in Figure 10. Note that, the argument Π to e^v must be ID if e is a variable whose value is not a constructed tuple, or an application of a constructor or a primitive function that is not $<>$ or ith. We can easily show that each rule guarantees sufficient information. Thus, the sufficiency conditions are satisfied by recursion induction.

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

Figure 10: Definition of $e^v\,\Pi$ for $\Pi \neq BOT, ABS$

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 Π of $\bar f_0{}'(x, y, \bar r)$ is needed.

To compute $f^i\,\Pi$ for some f^i and $\Pi \neq BOT, ABS$ (otherwise, we can use (27)), 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 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 every 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.

Computing the Closure of the Transitive Dependencies. To compute the components 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

$$\{1^{st}\} \sqsubseteq \Pi \ \ \text{and} \ \ \Pi(\bar f_0{}'(x, y, \bar r)) \sqsubseteq \bar f_0{}'(x, y, \Pi(\bar r)). \tag{28}$$

Of course, the projection $\Pi = ID$ is always a solution to (28). 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

$$\begin{array}{lcl} \Pi^{(0)} & = & \{1^{st}\} \\ \Pi^{(i+1)} & = & \Pi^{(i)} \ \cup \ \bar{f_0}'^{\,i_{\bar r}}\Pi^{(i)} \end{array} \tag{29}$$

and compute the least fixed point of Π. In other words, 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, i.e., the largest number of function applications in a function definition in the program for 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 the program for $\bar{f_0}'$. Let n be the number of function definitions, and 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 of 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 $na2^c$. Now, let s_f be the maximum number of transformers used in a transformer definition, i.e., the number of function applications in a function definition. Being careful, we can recompute each $f^i\Pi$ only when any computed projections used by $f^i\Pi$ change, 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 $na2^c cs_f$. Each such computation takes at most sc time, where s be the maximum size of a function definition, i.e., the number of subexpressions in the defining expression, and c is the time needed to compute operations, such as union, on two projections. Therefore, the total time is at most $na2^c c^2 ss_f$.

If we limit 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 big.

Now that the above estimate includes the computations of all $f^i\Pi$, computing the dependency closure takes at most c projection unions, each taking at most c time. Thus, the total time of closure computation can be no worse than the above bound.

Example. Applying the dependency analysis to the function $\overline{foo}'$ in (23), we get

$$\begin{array}{lcl} \overline{foo'}^{2}\Pi & = & (x \le 1)^{\bar r}\ I\!D \ \cup (<1,\_,\_>)^{\bar r}\Pi \ \cup \\ & & (x = 2)^{\bar r}\ I\!D \ \cup (<3,<2,<1,\_,\_>,<1,\_,\_>>,<1,\_,\_>>)^{\bar r}\Pi \ \cup \\ & & (<1st(\bar r)+1st(2nd(\bar r)),\ <1st(\bar r)+1st(2nd(2nd(\bar r))),\ \bar r,\ 2nd(2nd(\bar r))>,\ 3rd(2nd(\bar r))>)^{\bar r}\Pi \\ & = & (1st(\bar r)+1st(2nd(\bar r)))^{\bar r}\Pi_{(1)} \ \cup \\ & & (1st(\bar r)+1st(2nd(2nd(\bar r))))^{\bar r}\Pi_{(2)(1)} \ \cup \ (\bar r)^{\bar r}\Pi_{(2)(2)} \ \cup \ (2nd(2nd(\bar r)))^{\bar r}\Pi_{(2)(3)} \ \cup \\ & & (3rd(2nd(\bar r)))^{\bar r}\Pi_{(3)} \end{array}$$

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,

$$\begin{array}{lcl} \overline{foo}'^{2}\{1^{st}\} & = & \{1^{st}, 1^{st}2^{nd}\} \\ \overline{foo}'^{2}\{1^{st}2^{nd}\} & = & \{1^{st}, 1^{st}2^{nd}2^{nd}\} \\ \overline{foo}'^{2}\{1^{st}2^{nd}2^{nd}\} & = & \{1^{st}\} \end{array}$$

which illustrates the dependencies depicted in Figure 9. An example where the dependency transformer is defined recursively is shown in the merge sort example in Section 8. Now, we compute the projection for the closure of the transitive dependencies:

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

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

6.3 Pruning Under the Closure Projection.

We have obtained a closure projection Π such that $1^{st} \in \Pi$ and $\Pi(\bar{f}_0'(x,y,\bar{r})) \sqsubseteq \bar{f}_0'(x,y,\Pi(\bar{r}))$. Now, we prune the extended function $\bar{f}_0$ to get a function $\hat{f}_0$ such that $\Pi(\bar{f}_0(x)) \sqsubseteq \hat{f}_0(x)$, and prune the incremental function $\bar{f}_0'$ to get a function $\hat{f}_0'$ 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 only want to do this if Π is *ID*, otherwise 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, and thereby 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 $\bar{f}_0$ (respectively $\bar{f}_0'$) is needed. The definition and computation of the associated projections can be done in a fashion similar to the dependency analysis. For the program for $\bar{f}_0'$ and the closure projection Π, the final projection computed associated with each variable will be the same as computed for the variable using dependency analysis.

When the computation reaches the limit of the ascending chain of projections, subexpressions associated with *ID* are left unchanged in the resulting function, 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 $\hat{f}_0(x) = \Pi(\bar{f}_0(x))$ as anticipated in Section 3. Nevertheless, the resulting $\hat{f}_0$ is still good enough to guarantee (9). We can just project $\Pi(\bar{r})$ out of the return value of $\hat{f}_0(x)$. But 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 (10). As a matter of fact, we intend to use the function $\hat{f}_0$ only once to get the initial value, and then use the function $\hat{f}_0'$ repeatedly to compute all successive values. Recall that $\hat{f}_0'$ incrementally computes the desired output and the corresponding intermediate results, as shown in (10).

Consider the functions $\overline{foo}$ and $\overline{boo}$ in (16). Only $\{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}$ of $\overline{foo}(x)$ is needed, therefore only $\{1^{st}, 1^{st}2^{nd}\}$ of $\overline{boo}(x)$ is needed, and therefore only $\{1^{st}\}$ of $\overline{foo}(x-3)$ is needed. Thus, $\widehat{boo}_1(x)$ returns only $\{1^{st}, 1^{st}2^{nd}\}$ of $\overline{boo}(x)$, and $\widehat{foo}_1(x)$ returns only $\{1^{st}\}$ of $\overline{foo}(x)$ and the result $\{1^{st}, 1^{st}2^{nd}\}$ of $\widehat{boo}_1(x)$, as follows:

$$
\begin{array}{ll}
\widehat{foo}_1(x) = & \textbf{if } x \le 2 \textbf{ then } <1, \_, \_> \\
& \textbf{else let } u_1 = \widehat{boo}_1(x) \textbf{ in} \\
& \quad \textbf{let } u_2 = \widehat{foo}_1(x-3) \textbf{ in} \\
& \quad <1st(u_1)+1st(u_2),\ <1st(u_1), <1st(2nd(u_1)), \_, \_>, \_>,\ \_>
\end{array}
\qquad
\begin{array}{ll}
\widehat{boo}_1(x) = & \textbf{let } u_1 = \widehat{foo}_1(x-1) \textbf{ in} \\
& \textbf{let } u_2 = \widehat{foo}_1(x-2) \textbf{ in} \\
& <1st(u_1)+1st(u_2),\ <1st(u_1), \_, \_>,\ \_>
\end{array}
\tag{30}
$$

Consider the function $\overline{foo}'$ in (23). Only $\{1^{st}, 1^{st}2^{nd}, 1^{st}2^{nd}2^{nd}\}$ of $\overline{foo}'(x,y,\bar{r})$ is needed. Therefore, $\widehat{foo}_1(x,y,\hat{r}_1)$

returns only the corresponding components. We have, if $\widehat{foo}_1(x) = \hat{r}_1$, then $\widehat{foo}_1'(x, \hat{r}_1) = \widehat{foo}_1(x+1)$.

$$\begin{array}{ll} \widehat{foo}_1'(x,\hat{r}_1) = & \textbf{if } x \le 1 \textbf{ then } <1,\_,\_> \\ & \textbf{else if } x=2 \textbf{ then } <3,\ <2,<1,\_,\_>,<\_,\_,\_>>,\ <\_,\_,\_>> \\ & \textbf{else } <1st(\hat{r}_1)+1st(2nd(\hat{r}_1)),\ <1st(\hat{r}_1)+1st(2nd(2nd(\hat{r}_1))),<1st(\hat{r}_1),\_,\_>,\_>,\ \_> \end{array} \tag{31}$$

Simplification. After the replacements, a number of simplifications can be applied to the resulting functions: (a) unfolding a **let** expression if a binding variable occurs at most once in the body due to some replacements by _'s, (b) combining unnecessarily split components resulting from some replacements for variables whose values are constructed tuples, (c) lifting common selection computations to avoid unnecessarily computing a compound value and using only part of it, and (d) replacing occurrences of $1st(\hat{f}(e_1,...,e_n))$ by occurrences of $f(e_1,...,e_n)$.

For the function $\widehat{foo}_1$ in (30), we unfold the binding for u_2, replace the occurrence of $1st(\widehat{foo}_1(x-3))$ by $foo(x-3)$, and merge separate components of u_1. For the function $\widehat{boo}_1$ in (30), we unfold the binding for u_2, replace the occurrence of $1st(\widehat{foo}_1(x-2))$ by $foo(x-2)$, and lift $1st(u_1)$. We obtain

$$\begin{array}{ll} \widehat{foo}_2(x) = & \textbf{if } x \le 2 \textbf{ then } <1,\_,\_> \\ & \textbf{else let } u_1 = \widehat{boo}_2(x) \textbf{ in} \\ & \quad <1st(u_1)+foo(x-3),\ u_1,\ \_> \end{array} \qquad \begin{array}{ll} \widehat{boo}_2(x) = & \textbf{let } v_1 = foo(x-1) \textbf{ in} \\ & <v_1+foo(x-2),\ <v_1,\_,\_>,\ \_> \end{array} \tag{32}$$

The function $\widehat{foo}_1'$ remains the same.

Finally, we can eliminate _ components. But we must be careful if such a component precedes a non-_ component in a tuple, since our selectors ith's follow the indexing, which need to be changed accordingly. In particular, if k of the components preceding a component i are eliminated from a tuple, 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 $\hat{f}_0'$. At the end, we obtain the function $\hat{f}_0$, which caches only the useful intermediate results for incremental computation under $\oplus$, and the function $\hat{f}_0'$, which incrementally maintains only the useful intermediate results.

Theses simplifications and eliminations can be fully automated. For the functions $\widehat{foo}_2$ and $\widehat{boo}_2$ in (32) and $\widehat{foo}_1'$ in (31), we eliminate unnecessary _ components and obtain

$$\begin{array}{ll} \widehat{foo}(x) = & \textbf{if } x \le 2 \textbf{ then } <1> \\ & \textbf{else let } u_1 = \widehat{boo}(x) \textbf{ in} \\ & \quad <1st(u_1)+foo(x-3),\ u_1> \end{array} \qquad \begin{array}{ll} \widehat{boo}(x) = & \textbf{let } v_1 = foo(x-1) \textbf{ in} \\ & <v_1+foo(x-2),\ <v_1>> \end{array} \tag{33}$$

and

$$\begin{array}{ll} \widehat{foo}'(x,\hat{r}) = & \textbf{if } x \le 1 \textbf{ then } <1> \\ & \textbf{else if } x=2 \textbf{ then } <3,\ <2,\ <1>>> \\ & \textbf{else } <1st(\hat{r})+1st(2nd(\hat{r})),\ <1st(\hat{r})+1st(2nd(2nd(\hat{r}))),\ <1st(\hat{r})>>> \end{array} \tag{34}$$

These are the pruned extended functions $\widehat{foo}$ and $\widehat{boo}$ as given in Figure 2. The overall effect is that only $\{1^{st}\}$ and part of $\{2^{nd}\}$ are returned; and for the part of $\{2^{nd}\}$, only $\{1^{st}\}$ and part of $\{2^{nd}\}$ is returned; and for the part of $\{2^{nd}\}$ of $\{2^{nd}\}$, only $\{1^{st}\}$ is returned.

7 Discussion

We have obtained not only the extended function $\hat{f}_0$, which caches appropriate intermediate results, but also the corresponding function $\hat{f}_0'$ that incrementally maintains these intermediate results. The functions $\hat{f}_0$ and

$\hat{f_0}'$ preserve the semantics of computations and compute asymptotically at least as fast, as described in (9) and (10).

The cache-and-prune method is parameterized with respect to the derivation approach used in Stage II, and thus it is modular. The maximality provided by the caching in Stage I and minimality by the pruning in Stage III also give us the optimality of the method with respect to the derivation approach used in Stage II.

Transformation and Analysis Techniques. In cooperation with the approach for deriving incremental programs, we achieve the goal of identifying and maintaining intermediate results useful for incremental computation. A number of program analysis and transformation techniques are used for caching and pruning. These transformations can be fully automated. We summarize relevant techniques here.

First, the transformation $\mathcal{E}xt$ is similar to the construction of call-by-value complete recursive programs by Cartwright [6]. However, a call-by-value computation sequence returned by such a program is a flat list of all intermediate results, while our extended function returns a computation tree, a structure that mirrors the hierarchy of function calls. The transformations for in Stage I also mimics the CPS transformations in some aspects [35, 23]: sequencing subexpressions, naming intermediate results, passing the collected information, and performing administrative reductions on the resulting program. However, they are simpler than the CPS transformations since the collected intermediate results are passed directly to the return values, rather than to the continuation functions.

Second, the backward dependency analysis and pruning transformations in Stage III use domain projections to specify sufficient information, which is natural and thus simple. Other uses of projections include the strictness analysis by Wadler and Hughes [46], where necessary information needs to be specified and thus accounts for some complications, and binding time analysis by Launchbury [21], which is a forward analysis and is proved equivalent to strictness analysis [22]. The necessity interpretation by Jones and Le Métayer [17] is in the same spirit of our analysis, where their notion of necessity patterns correspond to our notion of projections. While necessity patterns specify heads and tails of list values, our projections specify specific components of tuple values and thus provide more accurate information.

Since the dependency analysis and pruning transformations simply eliminate dead components and related computations on compound values, it would be useful for general program optimizations in context. For example, in many functional programs, we create compound values only to take them apart somewhere else, and perhaps we only use some of the components. It would be nice to avoid constructing and passing the unnecessary components. Related work is done in optimizing compilers that eliminate unnecessary tuple constructions and destructions in functional programs; for example, the Id compiler [45] does tuple elimination. We think our analyses and transformations provide a straightforward solution to such problems. For us, it is more lightweight than trying to adopt any of the existing techniques.

There are a couple of analyses and transformations not yet mentioned that we believe could be incorporated in our framework. First, type analysis is very useful for many program manipulations, e.g., for the incrementalization in Stage II. We could easily equip the transformations in Stage I and III with corresponding manipulations needed for types. Second, Stage III replaces irrelevant components with the constant $\_$'s and performs a number of simplifications, where further manipulation with projections may help perform more simplifications like *component lifting.* For example, if we lift the single component in the second component of the second component of $\widehat{foo}$ in (33) and $\widehat{foo}'$ in (34), and simplify the selection $1st(2nd(2nd(\hat{r})))$

in $\widehat{foo}'$ to be $2nd(2nd(\hat{r}))$, we obtain $\widehat{foo}$ in (33), and $\widehat{boo}$ and $\widehat{foo}'$ as follows:

$$\begin{array}{ll}\widehat{boo}(x) = & \textbf{let } v_1 = foo(x-1) \textbf{ in} \\ & <v_1 + foo(x-2),\ v_1>\end{array} \qquad \begin{array}{ll}\widehat{foo}'(x,\hat{r}) = & \textbf{if } x \le 1 \textbf{ then } <1> \\ & \textbf{else if } x = 2 \textbf{ then } <3,\ <2,1>> \\ & \textbf{else } <1st(\hat{r})+1st(2nd(\hat{r})),\ <1st(\hat{r})+2nd(2nd(\hat{r})),\ 1st(\hat{r})>>\end{array}$$

Cost Model and Time/Space Trade-Off. The basic motivation for caching is to trade space for speed. Ideally, we would have a cost model for time and a cost model for space, and decide what to cache depending on the trade-off between time and space required by the application.

This paper assumes that we have unlimited space to be used for achieving the least asymptotic time possible, and thus we cache only the useful values of function applications, 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. Note that we assume that space is unlimited, not that it is free. Each of the three stages make an effort to reduce space consumption without adversely affecting asymptotic time performance.

One could be more mindful of economizing cache space by avoiding 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 $f(x) + g(x)$; coincidently, this also improves the speed of this example by a slight constant amount.

On the other hand, we could be more mindful of constant speed-up regardless of additional space consumption by caching the values of all program constructs, not just function applications. For example, we would cache the values of $f(x)$, $g(x)$, and $f(x)+g(x)$ respectively 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 of the time-space trade-off may also be required by applications, for example, a fixed cache space for achieving the least running time possible. For some applications, we may need to take into account the number of times a given value is needed.

8 Examples

Fibonacci Function. The definition of Fibonacci function fib is as given in Figure 1. If we apply the derivation approach of [26] on fib and $\oplus_1$, we obtain an incremental function fib' such that, if $fib(x) = r$, then $fib'(x, r) = fib(x + 1)$:

$$\begin{array}{ll}fib'(x,r) = & \textbf{if } x \le 0 \textbf{ then } 1 \\ & \textbf{else if } x = 1 \textbf{ then } 2 \\ & \textbf{else } r + fib(x-1)\end{array}$$

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

I. We apply the optimized extension transformation $\mathcal{E}xt1$ and obtain a function $\overline{fib}_1$:

$$\begin{array}{lllll}\overline{fib}_1(x) = & \multicolumn{4}{l}{\textbf{if } x \le 1 \textbf{ then } <1,\_,\_>} \\ & \textbf{else let } v_3 = & \textbf{let } v_{31} = & \textbf{let } v_1 = \overline{fib}_1(x-1) \textbf{ in} & \\ & & & <1st(v_1), v_1> & \textbf{in} \\ & & \textbf{let } v_{32} = & \textbf{let } v_2 = \overline{fib}_1(x-2) \textbf{ in} & \\ & & & <1st(v_2), v_2> & \textbf{in} \\ & & \multicolumn{3}{l}{<1st(v_{31})+1st(v_{32})> @rst(v_{31})@rst(v_{32})} \\ & \multicolumn{4}{l}{\quad <1st(v_3)> @ <> @ <rst(v_3)>}\end{array}$$

We apply the cleaning transformation and obtain an extended function $\overline{fib}$:

$$\begin{array}{rl} \overline{fib}(x) = & \textbf{if } x \le 1 \textbf{ then } <1,\_,\_> \\ & \textbf{else let } v_1 = \overline{fib}(x-1) \textbf{ in} \\ & \quad \textbf{let } v_2 = \overline{fib}(x-2) \textbf{ in} \\ & \quad <1st(v_1)+1st(v_2),\ v_1,\ v_2> \end{array} \tag{35}$$

II. We derive an incremental version of $\overline{fib}$ under $\oplus_1$ using [26], i.e., we transform $\overline{fib}(x \oplus_1 y) = \overline{fib}(x+1)$, with $\overline{fib}(x) = \bar{r}$:

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

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

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

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

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

$$\begin{array}{rl} \overline{fib}'(x,\bar{r}) = & \textbf{if } x \le 0 \textbf{ then } <1,\_,\_> \\ & \textbf{else if } x = 1 \textbf{ then } <2,\ <1,\_,\_>,\ <1,\_,\_>> \\ & \textbf{else } <1st(\bar{r})+1st(2nd(\bar{r})),\ \bar{r},\ 2nd(\bar{r})> \end{array} \tag{36}$$

III. Using the 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:

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

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

Then, we 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:

$$\begin{array}{rl} \widehat{fib}(x) = & \textbf{if } x \le 1 \textbf{ then } <1> \\ & \textbf{else let } u_1 = fib(x-1) \textbf{ in} \\ & \quad <u_1 + fib(x-2),\ u_1> \end{array} \tag{37}$$

$$\begin{array}{rl} \widehat{fib}'(x,\hat{r}) = & \textbf{if } x \le 0 \textbf{ then } <1> \\ & \textbf{else if } x = 1 \textbf{ then } <2,1> \\ & \textbf{else } <1st(\hat{r}) + 2nd(\hat{r}),\ 1st(\hat{r})> \end{array} \tag{38}$$

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

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

Letting $v = x+1$, we get

$$\begin{array}{rl} \widehat{fib}(v) = & \textbf{if } v \le 1 \textbf{ then } <1> \\ & \textbf{else if } v = 2 \textbf{ then } <2,1> \\ & \textbf{else let } \hat{r} = \widehat{fib}(v-1) \textbf{ in } <1st(\hat{r})+2nd(\hat{r}),\ 1st(\hat{r})> \end{array} \tag{39}$$

Finally, we define $fib(v) = 1st(\widehat{fib}(v))$ using the definition of $\widehat{fib}$ in (39). Clearly, this computes the Fibonacci function in linear time, as desired.

Merge Sort. The definition of merge sort function *sort* is as given in Figure 1. We consider the input change $x \oplus_2 y = cons(y, x)$.

I. We cache all intermediate results of *sort* and obtain the extended functions:

$$
\begin{array}{l}
\overline{sort}(x) = \textbf{if } null(x) \textbf{ then} \\
\qquad < nil, \_, \_, \_, \_, \_ > \\
\quad \textbf{else if } null(cdr(x)) \textbf{ then} \\
\qquad < x, \_, \_, \_, \_, \_ > \\
\quad \textbf{else let } v_{11} = \overline{odd}(x) \textbf{ in} \\
\qquad \textbf{let } u_1 = \overline{sort}(1st(v_{11})) \textbf{ in} \\
\qquad \textbf{let } v_{21} = \overline{even}(x) \textbf{ in} \\
\qquad \textbf{let } u_2 = \overline{sort}(1st(v_{21})) \textbf{ in} \\
\qquad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in} \\
\qquad < 1st(v), v_{11}, u_1, v_{21}, u_2, v >
\end{array}
\qquad
\begin{array}{l}
\overline{odd}(x) = \textbf{if } null(x) \textbf{ then } < nil, \_ > \\
\quad \textbf{else let } v_1 = \overline{even}(cdr(x)) \textbf{ in} \\
\qquad < cons(car(x), 1st(v_1)),\ v_1 > \\
\overline{even}(x) = \textbf{if } null(x) \textbf{ then } < nil, \_ > \\
\quad \textbf{else let } v_1 = \overline{odd}(cdr(x)) \textbf{ in} \\
\qquad < 1st(v_1),\ v_1 > \\
\overline{merge}(x, y) = \textbf{if } null(x) \textbf{ then } < y, \_, \_ > \\
\quad \textbf{else if } null(y) \textbf{ then } < x, \_, \_ > \\
\quad \textbf{else if } car(x) \leq car(y) \textbf{ then} \\
\qquad \textbf{let } v_1 = \overline{merge}(cdr(x), y) \textbf{ in} \\
\qquad < cons(car(x), 1st(v_1)),\ v_1, \_ > \\
\quad \textbf{else let } v_2 = \overline{merge}(x, cdr(y)) \textbf{ in} \\
\qquad < cons(car(y), 1st(v_2)), \_,\ v_2 >
\end{array}
\qquad (40)
$$

II. We derive an incremental version of $\overline{sort}$ under $\oplus_2$ using [26], i.e., we transform $\overline{sort}(x \oplus y) = \overline{sort}(cons(y, x))$, with $\overline{sort}(x) = \bar{r}$:

$$
\begin{array}{l}
\text{1. unfold } \overline{sort}(cons(y,x)) \text{, simplify primitives} \\
= \ \textbf{if } null(x) \textbf{ then} \\
\qquad < cons(y, nil), \_, \_, \_, \_, \_ > \\
\quad \textbf{else let } v_1 = \overline{even}(x) \textbf{ in} \\
\qquad \textbf{let } v_{11} = < cons(y, 1st(v_1)),\ v_1 > \textbf{ in} \\
\qquad \textbf{let } u_1 = \overline{sort}(1st(v_{11})) \textbf{ in} \\
\qquad \textbf{let } v_2 = \overline{odd}(x) \textbf{ in} \\
\qquad \textbf{let } v_{21} = < 1st(v_2), v_2 > \textbf{ in} \\
\qquad \textbf{let } u_2 = \overline{sort}(1st(v_{21})) \textbf{ in} \\
\qquad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in} \\
\qquad < 1st(v), v_{11}, u_1, v_{21}, u_2, v >
\end{array}
\qquad
\begin{array}{l}
\text{2. separate cases, replace applications of } \overline{sort} \text{ by retrievals} \\
= \ \textbf{if } null(1st(\bar{r})) \textbf{ then} \\
\qquad < cons(y, nil), \_, \_, \_, \_, \_ > \\
\quad \textbf{else if } null(cdr(1st(\bar{r}))) \textbf{ then} \\
\qquad \textbf{let } v_{11} = < cons(y, nil),\ < nil,\ < nil >>> \textbf{ in} \\
\qquad \textbf{let } v_{21} = < 1st(\bar{r}),\ < 1st(\bar{r}),\ < nil >>> \textbf{ in} \\
\qquad \textbf{let } v = \overline{merge}(cons(y, nil), 1st(\bar{r})) \textbf{ in} \\
\qquad < 1st(v),\ v_{11}, < cons(y, nil) >,\ v_{21}, < 1st(\bar{r}) >,\ v > \\
\quad \textbf{else let } v_1 = 4th(\bar{r}) \textbf{ in} \\
\qquad \textbf{let } v_{11} = < cons(y, 1st(v_1)), v_1 > \textbf{ in} \\
\qquad \textbf{let } u_1 = \overline{sort}'(y, 1st(v_1), 5th(\bar{r})) \textbf{ in} \\
\qquad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\
\qquad \textbf{let } v_{21} = < 1st(v_2), v_2 > \textbf{ in} \\
\qquad \textbf{let } u_2 = 3rd(\bar{r}) \textbf{ in} \\
\qquad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in} \\
\qquad < 1st(v), v_{11}, u_1, v_{21}, u_2, v >
\end{array}
$$

and obtain the function $\overline{sort}'$ such that, if $\overline{sort}(x) = \bar{r}$, then $\overline{sort}'(y, \bar{r}) = \overline{sort}(cons(y, x))$:

$$
\begin{array}{l}
\overline{sort}'(y, \bar{r}) = \ \textbf{if } null(1st(\bar{r})) \textbf{ then} \\
\qquad < cons(y, nil), \_, \_, \_, \_, \_ > \\
\quad \textbf{else if } null(cdr(1st(\bar{r}))) \textbf{ then} \\
\qquad \textbf{let } v_{11} = < cons(y, nil),\ < nil,\ < nil >>> \textbf{ in} \\
\qquad \textbf{let } v_{21} = < 1st(\bar{r}),\ < 1st(\bar{r}),\ < nil >>> \textbf{ in} \\
\qquad \textbf{let } v = \overline{merge}(cons(y, nil), 1st(\bar{r})) \textbf{ in} \\
\qquad < 1st(v),\ v_{11}, < cons(y, nil) >,\ v_{21}, < 1st(\bar{r}) >,\ v > \\
\quad \textbf{else let } v_1 = 4th(\bar{r}) \textbf{ in} \\
\qquad \textbf{let } v_{112} = < cons(y, 1st(v_1)), v_1 > \textbf{ in} \\
\qquad \textbf{let } u_1 = \overline{sort}'(y, 5th(\bar{r})) \textbf{ in} \\
\qquad \textbf{let } v_2 = 2nd(\bar{r}) \textbf{ in} \\
\qquad \textbf{let } v_{212} = < 1st(v_2), v_2 > \textbf{ in} \\
\qquad \textbf{let } u_2 = 3rd(\bar{r}) \textbf{ in} \\
\qquad \textbf{let } v = \overline{merge}(1st(u_1), 1st(u_2)) \textbf{ in} \\
\qquad < 1st(v), v_{112}, u_1, v_{212}, u_2, v >
\end{array}
\qquad (41)
$$

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 for $1^{st} \in \Pi$. With $\Pi_{(1)} = ID$ and $\overline{merge}^{1}\{1^{st}\} = \overline{merge}^{2}\{1^{st}\} = ID$, we have

$$\begin{array}{rcl} \overline{sort}'^{2(0)}\Pi & = & ABS \\ \overline{sort}'^{2(i+1)}\Pi & = & \{1^{st}\} \;\cup \\ & & (4th(\bar{r}))^{\bar{r}}((1st(v_1))^{v_1}\Pi_{(2)(1)} \;\cup\; \Pi_{(2)(2)}) \;\cup \\ & & (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}(\{1^{st}\} \;\cup\; \Pi_{(3)})) \;\cup \\ & & (2nd(\bar{r}))^{\bar{r}}((1st(v_2))^{v_2}\Pi_{(4)(1)} \;\cup\; \Pi_{(4)(2)}) \;\cup \\ & & (3rd(\bar{r}))^{\bar{r}}(\{1^{st}\} \;\cup\; \Pi_{(5)}) \end{array} \tag{42}$$

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

$$\begin{array}{lrcl} \Pi^{(0)} = \{1^{st}\} & \text{and, by (42), } \overline{sort}'^{2(i+1)}\Pi^{(0)} & = & \{1^{st}\} \cup\; (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)}\{1^{st}\}) \;\cup \{3^{rd}\} \\ & \overline{sort}'^{2(0)}\Pi^{(0)} & = & ABS \\ & \overline{sort}'^{2(1)}\Pi^{(0)} & = & \{1^{st}, 3^{rd}\} \\ & \overline{sort}'^{2(2)}\Pi^{(0)} & = & \{1^{st}, 3^{rd}, 5^{th}\} \\ & \overline{sort}'^{2(3)}\Pi^{(0)} & = & \{1^{st}, 3^{rd}, 5^{th}\} \\ \Pi^{(1)} = \{1^{st}, 3^{rd}, 5^{th}\} & \text{and, by (42), } \overline{sort}'^{2(i+1)}\Pi^{(1)} & = & \{1^{st}\} \cup\; (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)} ID) \;\cup \{3^{rd}\} \\ & \overline{sort}'^{2(i+1)} ID & = & \{1^{st}\} \cup \{4^{th}\} \cup\; (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)} ID) \;\cup \{2^{nd}\} \cup \{3^{rd}\} \\ & \overline{sort}'^{2(0)}\Pi^{(1)} & = & ABS \\ & \overline{sort}'^{2(1)}\Pi^{(1)} & = & \{1^{st}, 3^{rd}\} \\ & \overline{sort}'^{2(2)}\Pi^{(1)} & = & \{1^{st}, 3^{rd}, 5^{th}\} \\ & \overline{sort}'^{2(3)}\Pi^{(1)} & = & \{1^{st}, 3^{rd}, 5^{th}\} \\ \Pi^{(2)} = \{1^{st}, 3^{rd}, 5^{th}\} & & & \end{array}$$

Thus, we get the closure projection $\{1^{st}, 3^{rd}, 5^{th}\}$. Actually, for this example, we could directly see that

$$\overline{sort}'^{2}\{1^{st}, 3^{rd}, 5^{th}\} \;=\; \{1^{st}\} \;\cup\; (5th(\bar{r}))^{\bar{r}}(\overline{sort}'^{2(i)} ID) \;\cup\; (3rd(\bar{r}))^{\bar{r}} ID \;\subseteq\; \{1^{st}, 5^{th}, 3^{rd}\}$$

which matches the intuition that the first component of $\overline{sort}'$ depends only on $3rd(\bar{r})$ and $5th(\bar{r})$, and the third and fifth components depend on $3rd(\bar{r})$ and $5th(\bar{r})$ too. Now, we prune the functions $\overline{sort}$ and $\overline{sort}'$, and we obtain

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

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

Finally, we eliminate _ components, adjust the indexing, and obtain

$$\begin{aligned}
\widehat{sort}(x) = \;& \textbf{if } null(x) \textbf{ then } < nil > \\
& \textbf{else if } null(cdr(x)) \textbf{ then } < x > \\
& \textbf{else let } u_1 = \widehat{sort}(odd(x)) \textbf{ in} \\
& \quad\;\; \textbf{let } u_2 = \widehat{sort}(even(x)) \textbf{ in} \\
& \quad\;\; < merge(1st(u_1), 1st(u_2)),\ u_1,\ u_2 >
\end{aligned} \tag{43}$$

$$\begin{aligned}
\widehat{sort}'(y, \hat{r}) = \;& \textbf{if } null(1st(\hat{r})) \textbf{ then } < cons(y, nil) > \\
& \textbf{else if } null(cdr(1st(\hat{r}))) \textbf{ then} \\
& \quad\;\; < merge(cons(y, nil), 1st(\hat{r})),\ < cons(y, nil) >,\ < 1st(\hat{r}) >> \\
& \textbf{else let } u_1 = \widehat{sort}'(y, 3rd(\hat{r})) \textbf{ in} \\
& \quad\;\; \textbf{let } u_2 = 2nd(\hat{r}) \textbf{ in} \\
& \quad\;\; < merge(1st(u_1), 1st(u_2)),\ u_1,\ u_2 >
\end{aligned} \tag{44}$$

For x of length n, merge sort $sort$ takes $O(n \log n)$ time. Incremental merge sort $\widehat{sort}'$ takes only $O(n)$ time, although it uses $O(n \log n)$ space to store the intermediate results of the previous sort.

Attribute Evaluation Using Katayama Functions. Given an attribute grammar, a set of recursive functions can be constructed to evaluate the attribute values for any derivation tree of the grammar [18]. Basically, each function evaluates a synthesized attribute of a non-terminal, and the value of a synthesized attribute of the root symbol is the final return value of interest. Thus, for the given grammar, the set of recursive functions takes a derivation tree of the grammar as input, and returns the value of a synthesized attribute at the root as output.

We consider subtree replacement as the input change, given by a new subtree and a path from the root of the whole tree to the root of the subtree to be replaced.

First, caching all intermediate results leads to a set of extended recursive functions that returns an attributed tree instead of just the value of an synthesized attribute at the root. Then, incrementalizing the set of extended functions under a subtree replacement is just composing a new attributed tree from the old one, evaluating only attributes whose values are affected by the subtree replacement, yielding a set of incremental recursive functions.

Suppose a given batch attribute evaluation program evaluates each attribute only once, then the derived incremental program computes in $O(|\text{PATH}| + |\text{AFFECTED}|)$ time, where PATH is the path from the root of the whole tree to the root of the new subtree, and AFFECTED is the set of attributes whose values are different in the new tree than in the old after the subtree replacement [39].

9 Related Work and Conclusion

The cache-and-prune method uses a number of program analysis and transformation techniques that have been summarized in Section 7. Here we compare our work with related work in program improvement using caching techniques. Caching has been the basis of many techniques for developing efficient programs and optimizing programs. Bird [5] and Cohen [10] provide nice overviews. Most of these techniques fall into one of the following 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 the "exact tabulation" in [5] and the "large-table method" in [10]. The initial idea of memoization, "memo" functions, proposed by

Michie [27], belongs to this class. Thus, some uses of the word "memoization" mainly refers to techniques in this class [32].

In recent years, there has been additional work on general strategies for separate caching. For example, Hughes [15] discusses lazy memo-functions that are suitable for use in systems with lazy evaluation. Mostow and Cohen [28] discuss some issues for speeding up Interlisp programs by caching in the presence of side effects. Pugh [36] provides some improved cache replacement strategies for a simple functional language. Two trends seem obvious: studying *specialized* cache strategies for classes of problems, and adding *annotations* or certain *specifications* to subject programs that provide hints to the cache strategies. An example of the former is the stable decomposition scheme proposed by Pugh and Teitelbaum [37], who also advocate a closer study of using memoization for incremental evaluation. Examples of the latter include work by Keller and Sleep [19], which proposes annotating applicative languages, work by Sundaresh and Hudak [44, 43], which decides what to cache based on given input partitions of programs, and work by Hoover [14], which proposes annotating an imperative language.

The pros and cons of separate caching are well discussed by Bird [5] and Cohen [10]. To summarize, the idea is simple, and the subject programs are basically unchanged. But the caching methods are dynamic, and thus are fundamentally interpretive. Moreover, the strategies for the use and management of the separate cache are hard to be both general and powerful at the same time, and therefore are 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 [5], following which some uses of the word "tabulation" mainly refer to techniques in this class [32]. Typical examples of these techniques are dynamic programming [1], schemas of redundancies [10], and tupling [7, 8, 33, 34]. 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 used for 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 like 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 the 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 optimizing-compiler technique of strength reduction [3] identifies subcomputations like multiplications that can be replaced with subcomputations like additions while maintaining the values of these subcomputations. Similarly, the APTS program transformation system [29, 30] identifies set expressions in SETL that can be maintained using finite differencing rules [31].

Sometimes it is not sufficient to have only a fixed set of strategies and rules. Seeking more flexibility and broader applicability, KIDS [42] advocates certain high-level strategies but leaves the choice of which intermediate results to maintain to manual decisions. CIP [32] 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 [47] 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 heavy inference engine for thinning leads to some nice and clever optimizations but is computationally exorbitant. Hall's principle of redistributing intermediate results [13] finds paths from subcomputations to multiple uses of their values. However, the method uses a great deal of program design knowledge, including annotations of invariances, test-case inputs, and proofs of correctness. Also, it guarantees correctness of the transformed programs only on the test-case inputs.

Our approach to the problem of program improvement via caching is a principled approach that integrates caching in the transformed programs. 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. The 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 transformations. On the contrary, we can even use our approach to derive such rules when necessary. Compared to the general approaches advocated by KIDS or CIP, our approach is more algorithmic and automatable. A prototype system CACHET based on our approach is under development. We expect all the analyses and transformations presented in this paper to be fully automated.

Although we present the approach in a first-order functional language, the underlying principle is general and can be applied to other languages as well, e.g., higher-oder functional languages, functional languages with lazy semantics, and especially imperative languages with complex data structures and side effects. We have given an example [25] where the principle is applied to improve imperative programs with arrays for the local neighborhood problems in image processing [49, 51]. Further application of our principles to language with these features is a subject for future study.

References

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

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

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

[4] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The *Pan* language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.

[5] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[6] R. Cartwright. Recursive programs as definitions in first order logic. *SIAM Journal on Computing*, 13(2):374–408, May 1984.

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

[8] W.-N. Chin and S.-C. Khoo. Tupling functions with multiple recursion parameters. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proceedings of the 3rd International Workshop on Static Analysis*, pages 124–140. Springer-Verlag, September 1993. LNCS 724.

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

[10] N. H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

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

[12] C. A. Gunter. *Semantics of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1992.

[13] R. J. Hall. Program improvement by automatic redistribution of intermediate results: An overview. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, chapter 14, pages 339–372. AAAI Press/The MIT Press, 1991. Proceedings of the Workshop on Automating Software Design, AAAI '88.

[14] R. Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN '92 Conference on PLDI*, pages 261–272, California, June 1992.

[15] J. Hughes. Lazy memo-functions. In *Proceedings of the 2nd Conference on FPCA*, pages 129–146, Nancy, France, September 1985. Springer-Verlag. LNCS 201.

[16] F. Jalili and J. H. Gallier. Building friendly parsers. In *Conference Record of the 9th Annual ACM Symposium on POPL*, pages 196–206, Albuquerque, New Mexico, January 1982.

[17] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the 4th International Conference on FPCA*, pages 54–74, London, September 1989.

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

[19] R. M. Keller and M. R. Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, January 1986.

[20] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on PLDI*, pages 147–158, Orlando, Florida, June 1994.

[21] J. Launchbury. Projection factorisations in partial evaluation. Ph.d. thesis, Department of Computing, University of Glasgow, 1989.

[22] J. Launchbury. Strictness and binding-time analysis: Two for the price of one. In *Proceedings of the ACM SIGPLAN '91 Conference on PLDI*, pages 80–91, Toronto, Ontario, Canada, June 1991.

[23] J. L. Lawall and O. Danvy. Separating stages in the continuation-passing style transformation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 124–136, January 1993.

[24] P. Lipps, U. Möncke, M. Olk, and R. Wilhelm. Attribute (re)evaluation in OPTRAN. *Acta Informatica*, 26:213–239, 1988.

[25] Y. A. Liu and T. Teitelbaum. Incremental computation for transformational software development. Technical Report TR 95-1499, Department of Computer Science, Cornell University, Ithaca, New York, March 1995.

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

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

[28] D. J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the Ninth IJCAI*, pages 165–172, Los Angeles, August 1985.

[29] R. Paige. Transformational programming – applications to algorithms and systems. In *Conference Record of the 10th Annual ACM Symposium on POPL*, pages 73–87, January 1983.

[30] R. Paige. Symbolic finite differencing - part I. In *Proceedings of the 3rd ESOP*, pages 36–56, Copenhagen, Denmark, May 1990. Springer-Verlag. LNCS 432.

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

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

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

[34] A. Pettorossi. Strategical derivation of on-line programs. In L. G. L. T.Meertens, editor, *Program Specification and Transformation*, pages 73–88, Amsterdam, 1987. The Netherlands: North-Holland. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, April 1986.

[35] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[36] W. Pugh. An improved cache replacement strategy for function caching. In *Proceedings of the ACM '88 Conference on LFP*, pages 269–276, 1988.

[37] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on POPL*, pages 315–328, January 1989.

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

[39] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

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

[41] D. S. Scott. Lectures on a mathematical theory of computation. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292. D. Reidel Publishing Company, 1982. Lecture Notes of 1981 Marktoberdorf Summer School on Theoretical Foundations of Programming Methodology, directed by F.L. Bauer, E.W. Dijkstra, and C.A.R. Hoare.

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

[43] R. S. Sundaresh. Building incremental programs using partial evaluation. In *Proceedings of the Symposium on PEPM*, pages 83–93, Yale University, June 1991. Published as SIGPLAN Notices, 26(9).

[44] R. S. Sundaresh and P. Hudak. Incremental computation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on POPL*, pages 1–13, January 1991.

[45] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Master's Thesis LCS TR-370, Department of Electrical Engineering and Computer Science, MIT, August 1986.

[46] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on FPCA*, pages 385–407, Portland, Oregon, September 1987. LNCS 274.

[47] A. B. Webber. A formal definition of unnecessary computation in functional programs. Technical Report TR 92-1260, Department of Computer Science, Cornell University, Ithaca, New York, January 1992.

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

[49] W. M. Wells, III. Efficient synthesis of Gaussian filters by cascaded uniform filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):234–239, March 1986.

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

[51] R. Zabih. Individuating unknown objects by combining motion and stereo. Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, California, 1994.