Selectively caching intermediate results for incremental computation

Yanhong A. Liu*

Department of Computer Science, Cornell University, Ithaca, New York 14853

1 Introduction

Incremental computation takes advantage of repeated computations on inputs that slightly differ from one another, arranges outputs to be computed in an incremental fashion rather than from scratch. It has wide applications throughout the world of computing, e.g., program improvement [1, 18, 25] and interactive systems [10, 24].

Given a program f and an input change $\oplus$, a program f' that computes the value of $f(x \oplus y)$ efficiently by using the value of $f(x)$ is called an incremental version of f under $\oplus$. A systematic approach has been given for deriving such incremental programs [11]. It identifies in the computation of $f(x \oplus y)$ subcomputations whose values are also computed in computing $f(x)$ and can be efficiently retrieved from the cached result r of $f(x)$, and transforms $f(x \oplus y)$ symbolically to avoid recomputing these values. This efficient way of computing $f(x \oplus y)$ is captured in the program $f'(x, y, r)$.

The above approach has a limitation. The program $f'(x, y, r)$ avoids *only* subcomputations whose values are computed in computing $f(x)$ *and* can be retrieved from the result r of $f(x)$. There may be subcomputations of $f'(x, y, r)$ whose values are computed in computing $f(x)$ *but* can not be retrieved from r. If the values of these computations are also returned, then after input change $\oplus$, we can avoid recomputing them by retrieving them from the extended 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 include incremental parsing [7] and incremental attribute evaluation [10, 24, 27]. For example, sometimes in attribute evaluation, only a synthesized attribute of the root is of interest [8], but a whole attributed tree needs to be cached for incremental evaluation.

This paper presents a systematic approach that symbolically expands the computation of $f(x \oplus y)$ to selectively collect intermediate results useful for computing f incrementally under $\oplus$ and constructs a program $\hat{f}$ that returns these intermediate results in addition to the return value of f. The approach consists of four transformation steps: (1) expanding the computation of $f(x \oplus y)$, (2) selectively collecting subcomputations in the expanded computation whose values are computed in computing $f(x)$ but can not be retrieved from the value of $f(x)$, (3) tagging in the computation of $f(x)$ subcomputations whose values equal those of the collected subcomputations, and (4) constructing a function $\hat{f}$ that expands f to return the values of the tagged subcomputations.

Our approach follows lines different from all existing approaches, and is general and powerful. It decomposes a complicated task into small separate steps, thus is modular. The approach can also be used for general program improvements [12].

The paper is organized as follows. Section 2 formally describes the problem. Section 3 presents the four step transformations. Section 4 discusses related issues. Section 5 compares with related work and concludes.

2 Formalizing the Problem

We use a first-order functional programming language, with expressions given by the grammar:

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

*This research is supported by ONR contract No. N00014-92-J-1973. Author's Email: yanhong@cs.cornell.edu

Each constructor c, primitive function p, and user-defined function f has a fixed arity. A program is a set of recursive function definitions of the form $f(v_1, ..., v_n) = e$ 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.

sum minimums of nonempty sublists of counting numbers:

$sumin(x) =$ **if** $null(x)$ **then** 0
  else $min(car(x)) + sumin(cdr(x))$

$min(z) =$ **if** $null(cdr(z))$ **then** $car(z)$
  else let $s = min(cdr(z))$ **in**
    if $car(z) \leq s$ **then** $car(z)$ **else** s

cons elements in y to corresponding sublists in x; if there are more elements in y than sublists in x, ignore extra elements; if less, leave extra sublists:

$new(y, x) =$ **if** $null(x)$ **then** nil
  else if $null(y)$ **then** m
  else $cons(\,cons(car(y), car(x)),$
    $new(cdr(y), cdr(x)))$

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 $\oplus$ to $sumin$ can be $x' = x \oplus y = new(y, x)$.

Given a program f_0 and a change $\oplus$, we can use the approach in [11] to derive an incremental version f'_0 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 efficiently.[1] For the function $sumin$ and input change $x \oplus y = new(y, x)$, the function $sumin'$ in Figure 2 can be derived. Unfortunately, computing $sumin'(x, y, r)$ is not much faster than computing $sumin(new(y, x))$ from scratch. This is an example where we can compute the value of $f_0(x \oplus y)$ faster by caching and using, in addition to the value of $f_0(x)$, the intermediate results of $f_0(x)$. For example, the values of $min(car(x))$ and $sumin(cdr(x))$ are intermediate results of $sumin(x)$ that can be used to compute $sumin(new(y, x))$ faster.

We need cost models for time and space. This paper assumes an asymptotic time model. Assuming all primitive functions take constant time, 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. This paper assumes we have unlimited space to be used for achieving the least asymptotic time possible.

We use $<>$ to denote a tuple constructed by the transformation to bundle intermediate results with the original return value, with $1st$ returning the first element, the original value, and rst returning a tuple of the remaining elements, the corresponding intermediate results. We use nth to get the nth element of such a tuple, and an infix operation @ to concatenate two such tuples.

We use the function $sumin$ in Figure 1 as a running example. At the end, we obtain the functions $\widehat{sumin}$ and $\widehat{sumin}'$ in Figure 2. For a list x of length m, and each sublist of length n, both $sumin(new(y, x))$ and $\widehat{sumin}(new(y, x))$ take $O(mn)$ time, but $\widehat{sumin}'(y, \hat{r})$ takes $O(m)$ time.

3 Transformations for Caching Intermediate Results

3.1 Step 1. Expansion

The first step uses a transformation $\mathcal{E}xp$ to expand the computation $f_0(x \oplus y)$ symbolically. The goal is to decompose the computation of f_0 on $x \oplus y$ into computations on x and y separately so that computations on x alone whose values are also computed in computing $f_0(x)$ but can not be retrieved from the return value r of $f_0(x)$ can be cached and returned.

The transformation $\mathcal{E}xp$ considers subexpressions of $f_0(x \oplus y)$ in applicative order, simplifies primitive function applications, specializes two branches of conditional expressions, unwinds unnecessary binding expressions, unfolds function applications, and lifts conditionals and bindings out of conditionals, bindings, and arguments of applications. To avoid infinite unfolding, $\mathcal{E}xp$ unfolds function applications once for each function on each computation path. In particular, if the function application is $f_0(e_x \oplus e_y)$, where e_x does not depend on x and e_y depends on y as little as possible,

[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 [11].

If $sumin(x) = r$,
  then $sumin'(x,y,r) = sumin(new(y,x))$.
For x of size $m \times n$,
  $sumin'(x,y,r)$ takes time $O(mn)$;
  $sumin(new(y,x))$ takes time $O(mn)$.

$sumin(x) = 1st(\widehat{sumin}(x))$.
For x of size $m \times n$,
  $\widehat{sumin}(x)$ takes time $O(mn)$;
  $sumin(x)$ takes time $O(mn)$.

If $\widehat{sumin}(x) = \hat{r}$,
  then $\widehat{sumin}'(y,\hat{r}) = \widehat{sumin}(new(y,x))$.
For x of size $m \times n$,
  $\widehat{sumin}'(y,\hat{r})$ takes time $O(m)$;
  $\widehat{sumin}(new(y,x))$ takes time $O(mn)$.

$$
\begin{array}{lll}
sumin'(x,y,r) & = & \textbf{if } null(x) \textbf{ then } 0 \\
 & & \textbf{else if } null(y) \textbf{ then } r \\
 & & \textbf{else let } s = min(car(x)) \textbf{ in} \\
 & & \quad \textbf{if } car(y) \leq s \textbf{ then} \\
 & & \quad\; car(y) + sumin(new(cdr(y), cdr(x))) \\
 & & \quad \textbf{else } s + sumin(new(cdr(y), cdr(x))) \\
\widehat{sumin}(x) & = & \textbf{if } null(x) \textbf{ then } < 0 > \\
 & & \textbf{else let } u_1 = min(car(x)) \textbf{ in} \\
 & & \quad \textbf{let } u_2 = \widehat{sumin}(cdr(x)) \textbf{ in} \\
 & & \quad < u_1 + 1st(u_2),\ u_1,\ u_2 > \\
\widehat{sumin}'(y,\hat{r}) & = & \textbf{if } null(rst(\hat{r})) \textbf{ then } < 0 > \\
 & & \textbf{else if } null(y) \textbf{ then } \hat{r} \\
 & & \textbf{else let } u_2 = \widehat{sumin}'(cdr(y), 3rd(\hat{r})) \textbf{ in} \\
 & & \quad \textbf{if } car(y) \leq 2nd(\hat{r}) \\
 & & \quad \textbf{then } < car(y) + 1st(u_2),\ car(y),\ u_2 > \\
 & & \quad \textbf{else } < 2nd(\hat{r}) + 1st(u_2),\ 2nd(\hat{r}),\ u_2 >
\end{array}
$$

Figure 2: Resulting function definitions

then $\mathcal{E}xp$ replaces it by $f_0'(e_x, e_y, f_0(e_x))$, in the hope that the value of $f_0(e_x)$ can be retrieved from the (extended) cached result of $f_0(x)$.

The transformation $\mathcal{E}xp$ corresponds approximately to the transformations $\mathcal{G}_o$ and $\mathcal{D}_t$ in [11], but with a different motivation. Namely, the motivation here is to expose, as many as possible, subcomputations of $f_0(x)$ that are useful for computing $f_0(x \oplus y)$ incrementally *without* worrying about the cost of the expanded computation of $f_0(x \oplus y)$. We use $e_{f_0(x \oplus y)}$ to denote the resulting expression of this first step:

$$e_{f_0(x \oplus y)} = \mathcal{E}xp[\![f_0(x \oplus y)]\!] \tag{1}$$

Example. We expand $sumin(new(y,x))$. First, we unfold new and lift conditionals out of the argument of $sumin$. We get

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

Then, in the first branch, we specialize $sumin$ on argument nil; in the second branch, we leave $sumin(x)$ unchanged since its value can be retrieved from r without further expansion; in the third branch, we unfold $sumin$ and simplify the resulting expression using $car(cons(a,b)) = a$ and $cdr(cons(a,b)) = b$. We get

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

In the third branch, we unfold min and simplify the resulting expression, and we get

$$
\begin{array}{l}
\textbf{if } null(x) \textbf{ then } 0 \\
\textbf{else if } null(y) \textbf{ then } sumin(x) \\
\textbf{else (let } s = min(car(x)) \textbf{ in if } car(y) \leq s \textbf{ then } car(y) \textbf{ else } s) \\
\quad + sumin(new(cdr(y), cdr(x)))
\end{array}
$$

and then, we lift the binding and the conditional out of the argument of plus, and we get

$$
\begin{array}{l}
\textbf{if } null(x) \textbf{ then } 0 \\
\textbf{else if } null(y) \textbf{ then } sumin(x) \\
\textbf{else let } s = min(car(x)) \textbf{ in} \\
\quad \textbf{if } car(y) \leq s \textbf{ then } car(y) + sumin(new(cdr(y), cdr(x))) \\
\quad \textbf{else } s + sumin(new(cdr(y), cdr(x)))
\end{array}
$$

Now, in the third branch, we do not further unfold the applications of min, new, and $sumin$ since other applications of them have been unfolded on the corresponding computation paths. In particular, the applications of $sumin$ are instances of the original application $sumin(new(y,x))$, and thus we replace them by $sumin'(cdr(x), cdr(y), sumin(cdr(x)))$. Finally, we get the expanded $sumin(new(y, x))$ as follows:

$$\begin{array}{l} \textbf{if } null(x) \textbf{ then } 0 \\ \textbf{else if } null(y) \textbf{ then } sumin(x) \\ \textbf{else let } s = min(car(x)) \textbf{ in} \\ \quad \textbf{if } car(y) \le s \textbf{ then } car(y) + sumin'(cdr(x), cdr(y), sumin(cdr(x))) \\ \quad \textbf{else } s + sumin'(cdr(x), cdr(y), sumin(cdr(x))) \end{array} \tag{2}$$

3.2 Step 2. Collection

The second step uses a transformation $\mathcal{C}ol$ to collect subexpressions of $e_{f_0(x \oplus y)}$ obtained in Step 1 whose values are also computed in computing $f_0(x)$ but can not be retrieved from the cached result r of $f_0(x)$. The basic idea is to keep track of all the subcomputations performed in $f_0(x)$ and those whose values can be retrieved from r, and make use of the difference.

Consider any subexpression e of $e_{f_0(x \oplus y)}$. Let $I_{[e]}$, *information set* at the occurrence of e, be the set of equations representing the context of e. Let $R_{[e]}$, *retrieving set* at the occurrence of e, be the set of expressions that are computed in computing $f_0(x)$ under I_e and whose values can be retrieved from the cached result r. In particular, each expression in a retrieving set is associated with the list of function applications that are unfolded to obtain the expression, and for convenience, we just use the list operations in our language. For example,

$$I_{[e_{f_0(x \oplus y)}]} = \emptyset \qquad\qquad R_{[e_{f_0(x \oplus y)}]} = \{\langle f_0(x), nil \rangle\}$$

and, if $e_{f_0(x \oplus y)}$ is **if** e_1 **then** e_2 **else let** $v = e_3$ **in** e_4, and $f_0(x)$ is defined as **if** e_1 **then** $cons(e_5, e_6)$ **else** e_7, then

$$I_{[e_2]} = \{e_1 \leftrightarrow T\} \qquad\qquad I_{[e_4]} = \{e_1 \leftrightarrow F, v \leftrightarrow e_3\}$$

$$R_{[e_2]} = \{\langle f_0(x), nil \rangle,\ \langle cons(e_5, e_6),\ list(f_0(x)) \rangle,\ \langle e_5,\ list(f_0(x)) \rangle,\ \langle e_6,\ list(f_0(x)) \rangle\}$$

In general, if $I_{[e_2]} \supset I_{[e_1]}$, then $R_{[e_2]} = \mathcal{R}(R_{[e_1]}, I_{[e_2]})$, where $\mathcal{R}$ is a closure function that specializes expressions in $R_{[e_1]}$ under $I_{[e_2]}$ and extracts retrievable subexpressions (in a way similar to the function $\mathcal{C}$ in [13]).

Now, the tuple $\langle e, l, I_e \rangle$ is collected if (a) there is $\langle e_x, l \rangle \in R_e$ and a subexpression e'_x of e_x that is a function application such that $e \leftrightarrow^*_{I_e} e'_x$, and (b) there is no $\langle e_x, l' \rangle \in R_e$ such that $e \leftrightarrow^*_{I_e} e_x$. We use $\mathcal{S}el$ to denote the function that, given e, I_e, and R_e, returns the set containing the above tuple if the two conditions hold and the empty set otherwise.

The transformation $\mathcal{C}ol$, defined in Figure 3, collects the above tuples by recursively examining subexpressions. It takes an expression, an information set, and a retrieving set, and returns a collected set of tuples. We use $S_{f_0(x \oplus y)}$ to denote the resulting set obtained by applying $\mathcal{C}ol$ to $e_{f_0(x \oplus y)}$ with empty information set and retrieving set $\{\langle f_0(x), nil \rangle\}$:

$$S_{f_0(x \oplus y)} = \mathcal{C}ol[\![e_{f_0(x \oplus y)}]\!]\ \emptyset\ \{\langle f_0(x), nil \rangle\} \tag{3}$$

$$\begin{array}{lcl} \mathcal{C}ol[\![v]\!]\ I\ R & = & \emptyset \\ \mathcal{C}ol[\![g(e_1, \ldots, e_n)]\!]\ I\ R \quad \text{where } g \text{ is } c \text{ or } p & = & (\mathcal{C}ol[\![e_1]\!]IR) \cup \ldots \cup (\mathcal{C}ol[\![e_n]\!]IR) \\ \mathcal{C}ol[\![f(e_1, \ldots, e_n)]\!]\ I\ R & = & \mathcal{S}el(f(e_1, \ldots, e_n), I, R) \cup (\mathcal{C}ol[\![e_1]\!]IR) \cup \ldots \cup (\mathcal{C}ol[\![e_n]\!]IR) \\ \mathcal{C}ol[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ I\ R & = & (\mathcal{C}ol[\![e_1]\!]IR) \cup (\mathcal{C}ol[\![e_2]\!]I_2R_2) \cup (\mathcal{C}ol[\![e_3]\!]I_3R_3) \\ & & \text{where } I_2 = I \cup \{e_1 \leftrightarrow T\},\ I_3 = I \cup \{e_1 \leftrightarrow F\}, \\ & & \qquad R_i = \mathcal{R}(R, I_i),\ \text{for } i = 2, 3 \\ \mathcal{C}ol[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\ I\ R & = & (\mathcal{C}ol[\![e_1]\!]IR) \cup (\mathcal{C}ol[\![e_2]\!]I_2R_2) \\ & & \text{where } I_2 = I \cup \{v \leftrightarrow e_1\},\ R_2 = \mathcal{R}(R, I_2) \end{array}$$

Figure 3: Definition of $\mathcal{C}ol$

A possible improvement to the $\mathcal{C}ol$ transformation is to avoid collecting subexpressions whose values can be retrieved from other subexpressions that have been collected. Basically, we can keep an extra set argument parallel to R that contains these other subexpressions, and check the union of this set and R instead of only R in the conditions (a) and (b) above.

An the end of this step, if $S_{f_0(x \oplus y)}$ is empty, then there is no subcomputation in $f_0(x)$ whose value is useful for $f_0(x \oplus y)$ but can not be retrieved from the cached result r. Therefore, f_0 does not need to be extended to cache any such values; or, if we try to extend f_0 using the transformations that follow, we would simply obtain a function that is identical to f_0.

Example. We examine the expression for the expanded $sumin(new(y,x))$ in (2) and collect function applications whose values are also computed in computing $sumin(x)$ but can not be retrieved from the cached result r of $sumin(x)$.

In the second branch of (2), it is straightforward that $sumin(x)$ is computed by the old computation, and its value is directly retrievable from r. Therefore, nothing needs to be collected.

In the third branch of (2), we have $I = \{null(x) \leftrightarrow F, null(y) \leftrightarrow F\}$. Obviously, the values of $min(car(x))$ and $sumin(cdr(x))$ would be useful for the incremental computation. Notice that, when $null(x)$ is false, $sumin(x)$ is specialized to $min(car(x)) + sumin(cdr(x))$. Thus, $min(car(x))$ and $sumin(cdr(x))$ are actually computed in computing $sumin(x)$, just that their values can not be retrieved from the cached result of $sumin(x)$, since plus does not have inverse. Therefore, we get the collected set

$$\{\langle min(car(x)), list(sumin(x)), I\rangle, \langle sumin(cdr(x)), list(sumin(x)), I\rangle\} \tag{4}$$

3.3 Step 3. Tagging

This step uses a transformation $\mathcal{T}ag$ to tag, corresponding to the subexpressions collected in Step 2, subcomputations of $f_0(x)$ whose values need to be returned as well as the value of $f_0(x)$.

The transformation $\mathcal{T}ag$ is defined in Figure 4. It takes an expression, an information set, and a collected set of tuples, examines subexpressions recursively, unfolds functions when necessary, and, if the expression is a function application whose value equals that of the expression in a collected tuple t and the information set does not conflict with the information set in t, tags this function application. The function $\mathcal{A}$ used for function applications is defined by $\mathcal{A}(S,e) = \{\langle e', cdr(l), I\rangle \mid \langle e', l, I\rangle \in S,\ car(l) = e\}$, where e is a function application. The function $\mathcal{B}$ used for conditional expressions and binding expressions is defined by $\mathcal{B}(S,I) = \{\langle e, l, I'\rangle \mid \langle e, l, I'\rangle \in S,\ I' \wedge I \neq F\}$, where $I_1 \wedge I_2 \neq F$ means that the conjunctions of equations in I_1 and I_2 do not imply false. We use $\mathcal{T}ic$ to denote the function that, given e and S, returns true if there is $\langle e', l, I\rangle$ in S such that $e' \leftrightarrow^*_I e$ and false otherwise.

$\mathcal{T}ag[\![v]\!]IS \quad = \quad v$

$\mathcal{T}ag[\![g(e_1,\ldots,e_n)]\!]IS$ where g is c or p
$\quad = \quad g(\mathcal{T}ag[\![e_1]\!]IS, \ldots, \mathcal{T}ag[\![e_n]\!]IS)$

$\mathcal{T}ag[\![f(e_1,\ldots,e_n)]\!]IS$ where f is defined by $f(v_1,\ldots,v_n) = e$
$\quad = \quad f(\mathcal{T}ag[\![e_1]\!]IS, \ldots, \mathcal{T}ag[\![e_n]\!]IS)$ if $\mathcal{A}(S, f(e_1,\ldots,e_n)) = \emptyset$
$\quad = \quad f_1(u_1,\ldots,u_m)$ otherwise
where $u_1,\ldots,u_m$ are all the free variables in $e_1,\ldots,e_n$,
f_1 is a new function defined by $f_1(u_1,\ldots,u_m) =$
let $v_1 = \mathcal{T}ag[\![e_1]\!]IS$ **in** … **let** $v_n = \mathcal{T}ag[\![e_n]\!]IS$ **in**
$\mathcal{T}ag[\![e]\!]I\,\mathcal{A}(S, f(e_1,\ldots,e_n))$
and the resulting application of f or f_1 is tagged if $\mathcal{T}ic(f(e_1,\ldots,e_n), S)$ is true

$\mathcal{T}ag[\![$**if** e_1 **then** e_2 **else** $e_3]\!]IS \quad = \quad$ **if** $\mathcal{T}ag[\![e_1]\!]IS$ **then** $\mathcal{T}ag[\![e_2]\!]I_2S_2$ **else** $\mathcal{T}ag[\![e_3]\!]I_3S_3$
where $I_2 = I \cup \{e_1 \leftrightarrow T\}$, $I_3 = I \cup \{e_1 \leftrightarrow F\}$, $S_i = \mathcal{B}(S, I_i)$, for $i = 2,3$

$\mathcal{T}ag[\![$**let** $v = e_1$ **in** $e_2]\!]IS \quad = \quad$ **let** $v = \mathcal{T}ag[\![e_1]\!]IS$ **in** $\mathcal{T}ag[\![e_2]\!]I_2S_2$
where $I_2 = I \cup \{v \leftrightarrow e_1\}$, $S_2 = \mathcal{B}(S, I_2)$

Figure 4: Definition of $\mathcal{T}ag$

Applying $\mathcal{T}ag$ to $f_0(x)$ with empty information set and collected set $S_{f_0(x\oplus y)}$ obtained from Step 2, we obtain a function f_0^{tag} and possibly other functions defined by $\mathcal{T}ag$:

$$f_0^{tag}(x) = \mathcal{T}ag[\![f_0(x)]\!]\ \emptyset\ S_{f_0(x\oplus y)} \tag{5}$$

Example. Applying $\mathcal{T}ag$ on $sumin(x)$ with $\emptyset$ and the collected set in (4), we obtain $sumin^{tag}$ defined below. Here we underline tagged functions applications whose values need to be cached in addition to the original return value.

$$sumin^{tag}(x) = \begin{array}{l} \textbf{if } null(x) \textbf{ then } 0 \\ \textbf{else } \underline{min(car(x))} + \underline{sumin(cdr(x))} \end{array} \tag{6}$$

All other function definitions have no subexpressions tagged.

3.4 Step 4. Extension

This last step uses a transformation $\mathcal{E}xt$ to construct a function $\hat{f}_0$ that extends the function f_0 to return the intermediate results of the subexpressions tagged in Step 3.

The transformation $\mathcal{E}xt$, defined in Figure 5, extends an expression to return the values of tagged function applications in computing the expression. It examines subexpressions recursively, introduces variables to bind the values of tagged function applications, and builds up lists to embed the values in the return values of enclosing expressions. The set argument to $\mathcal{E}xt$ is used to assist the collection of intermediate results. The first component of the value of an extended function $\hat{f}$ is always the value of the function f, and the rest are the corresponding intermediate results. We assume that each binding introduced uses a fresh variable name. We use $ctf(e)$ to denote that e contains a tagged function application, and $nctf(e)$ to denote that e contains no tagged function application.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{E}xt\llbracket g(e_1,...,e_n)\rrbracket\ A \qquad \text{where } g \text{ is } c,\ p, \text{ or } f \text{ defined by } f(v_1,...,v_n) = e} \\
\quad = \textbf{let } u_i = \mathcal{E}xt\llbracket e_i\rrbracket\ \emptyset \textbf{ in } \mathcal{E}xt\llbracket g(e_1,...,e_n)\rrbracket\ (A \cup \{\langle u_i, e_i\rangle\}) & \text{if } e_1,...,e_{i-1} \text{ are extended, } e_i \text{ is not, and } ctf(e_i) \\
\quad = \mathcal{E}xt\llbracket g(e_1,...,e_n)\rrbracket\ A & \text{if } e_1,...,e_{i-1} \text{ are extended, } e_i \text{ is not, and } nctf(e_i) \\
\quad = <g(e'_1,...,e'_n)>\ @e''_1@...@e''_n & \text{if } e_1,...,e_n \text{ are extended, } g \text{ is } c,\ p, \text{ or un-tagged } f \\
\quad = \textbf{let } u = f(e'_1,...,e'_n) \textbf{ in } <u,u>\ @e''_1@...@e''_n & \text{if } e_1,...,e_n \text{ are extended, } g \text{ is tagged } f \text{ but } nctf(e) \\
\quad = \textbf{let } u = \hat{f}(e'_1,...,e'_n) \textbf{ in } <1st(u),u>\ @e''_1@...@e''_n & \text{otherwise} \\
\multicolumn{2}{l}{\qquad \text{and define } \hat{f}(v_1,...,v_n) = \mathcal{E}xt\llbracket e\rrbracket\ \emptyset \text{ if } f \text{ is encountered the first time}} \\
\\
\multicolumn{2}{l}{\mathcal{E}xt\llbracket \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3\rrbracket\ A} \\
\quad = \textbf{let } u_1 = \mathcal{E}xt\llbracket e_1\rrbracket\ \emptyset \textbf{ in } \mathcal{E}xt\llbracket \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3\rrbracket\ \{\langle u_1, e_1\rangle\} & \text{if } e_1 \text{ is not extended, and } ctf(e_1) \\
\quad = \mathcal{E}xt\llbracket \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3\rrbracket\ \emptyset & \text{if } e_1 \text{ is not extended, and } nctf(e_1) \\
\quad = \textbf{if } e'_1 \textbf{ then } e^*_2 \textbf{ else } e^*_3 & \text{otherwise} \\
\\
\multicolumn{2}{l}{\mathcal{E}xt\llbracket \textbf{let } v = e_1 \textbf{ in } e_2\rrbracket\ A} \\
\quad = \textbf{let } u_1 = \mathcal{E}xt\llbracket e_1\rrbracket\ \emptyset \textbf{ in } \mathcal{E}xt\llbracket \textbf{let } v = e_1 \textbf{ in } e_2\rrbracket\ \{\langle u_1, e_1\rangle\} & \text{if } e_1 \text{ is not extended, and } ctf(e_1) \\
\quad = \mathcal{E}xt\llbracket \textbf{let } v = e_1 \textbf{ in } e_2\rrbracket\ \emptyset & \text{if } e_1 \text{ is not extended, and } nctf(e_1) \\
\quad = \textbf{let } v = e'_1 \textbf{ in } e^*_2 & \text{otherwise}
\end{array}$$

$$\text{where } e'_i = \begin{cases} 1st(u_i) & \text{if } \langle u_i, e_i\rangle \in A \\ e_i & \text{otherwise} \end{cases} \text{ and } e''_i = \begin{cases} rst(u_i) & \text{if } \langle u_i, e_i\rangle \in A \\ <> & \text{otherwise} \end{cases} \text{ for } i = 1..n$$

$$e^*_i = \begin{cases} \textbf{let } u_i = \mathcal{E}xt\llbracket e_i\rrbracket\ \emptyset \textbf{ in } <1st(u_i)>\ @e''_1@rst(u_i) & \text{if } ctf(e_i) \\ <e_i>\ @e''_1 & \text{otherwise} \end{cases} \text{ for } i = 2,3$$

Figure 5: Definition of $\mathcal{E}xt$

We use $\hat{f}_0$ to denote the resulting function constructed by extending the function f_0:

$$\hat{f}_0(x) = \mathcal{E}xt\llbracket e_0\rrbracket\ \emptyset \tag{7}$$

where e_0 is the tagged definition of $f_0^{tag}(x)$, i.e., f_0^{tag} is defined by $f_0^{tag}(x) = e_0$.

Administrative simplifications are performed on the resulting function definitions: simplifying operations on lists that are built up for passing intermediate results, unwinding binding expressions that become unnecessary as a result of simplifying their subexpressions, and lifting bindings out of enclosing expressions.

Example. For the tagged definition of $sumin(x)$ in (6), we define a function $\widehat{sumin}$ that extends $sumin$ to return the intermediate results of the tagged function applications. We obtain the extended function $\widehat{sumin}$ given in Figure 2.

Using the approach for deriving incremental programs in [11], we can derive an incremental version $\widehat{sumin}'$ of $\widehat{sumin}$ under new. The derivation expands $\widehat{sumin}(new(y,x))$ and replaces subcomputations with retrievals from the cached result $\hat{r}$ (Refer to the expansion of Step 1 and the resulting expression in (2)):

$$
\begin{array}{rl}
\widehat{sumin}'(y,\hat{r}) = & \textbf{if } null(rst(\hat{r})) \textbf{ then } <0> \\
& \textbf{else if } null(y) \textbf{ then } \hat{r} \\
& \textbf{else let } u_1 = 2nd(\hat{r}) \textbf{ in} \\
& \quad \textbf{let } u_2 = \widehat{sumin}'(cdr(y), 3rd(\hat{r})) \textbf{ in} \\
& \quad\quad \textbf{if } car(y) \leq u_1 \\
& \quad\quad \textbf{then } < car(y) + 1st(u_2),\ car(y),\ u_2 > \\
& \quad\quad \textbf{else } \ < u_1 + 1st(u_2),\ u_1,\ u_2 >
\end{array}
\qquad (8)
$$

Unfolding the binding for u_1, we obtain the final definition of $\widehat{sumin}'$ given in Figure 2.

4 Discussion

The value of an application of f_0 can always be projected out of the first component of value of the corresponding application of $\hat{f}_0$, i.e., $f_0(x) = 1st(\hat{f}_0(x))$. Also, $\hat{f}_0(x)$ computes as fast as $f_0(x)$ asymptotically since they perform essentially the same computation. For the incremental computation, we have $\hat{f}_0'(x,y,\hat{r}) = \hat{v}$ if $\hat{f}_0(x \oplus y) = \hat{v}$, and $\hat{f}_0'(x,y,\hat{r})$ computes at least as fast as $\hat{f}_0(x \oplus y)$ asymptotically. Thus, $1st(\hat{f}_0'(x,y,\hat{r})) = v$ if $f_0(x \oplus y) = v$, and $\hat{f}_0'(x,y,\hat{r})$ computes at least as fast as $f_0(x \oplus y)$ asymptotically. The approach is optimal with respect to the functions $\mathcal{S}el$ in $\mathcal{C}ol$ and $\mathcal{T}ic$ in $\mathcal{T}ag$ in the sense that exactly those intermediate results that are selected by $\mathcal{S}el$ and then tagged by $\mathcal{T}ic$ are returned by the extended function $\hat{f}_0$.

Improvement. To compute the extended function $\hat{f}_0$ incrementally under $\oplus$, $\hat{f}_0'$ must maintain the corresponding intermediate results. It may be that these intermediate results can be maintained more efficiently by using intermediate results of f_0 other than those cached by $\hat{f}_0$. To find and cache these other intermediate results, we can apply the transformations again on the function $\hat{f}_0$ and $\oplus$ and obtain an further extended function $\hat{\hat{f}}_0$. This process repeats until a further extended function caches no more intermediate results than the previous extended function.

Another possible improvement is as follows. The techniques proposed in this paper complements the derivation approach in [11] to increase incrementality; but an enhancement to that derivation approach that has broader applicability has been proposed in [13]. We believe that corresponding improvement can be made to the techniques in this paper to complement the enhanced derivation approach.

5 Related Work and Conclusion

One class of caching techniques in incremental computation use the idea of memoization [14]. This class is called "exact tabulation" in [2] and "large-table method" in [3]. Recent work includes [6, 15, 22]. Additionally, [23] studies *specialized* cache strategies for classes of problems, and [9, 5] add *annotations* to subject programs. These techniques are based on dynamic methods, and thus are hard to be both general and powerful at the same time, which are sources of inefficiency.

To overcome the above drawback, people transform subject programs to integrate caching statically into the transformed programs. Some of the techniques apply transformations based on special schemas of subject programs and thus lack certain generality. Examples are schemas of redundancies [3] and tupling [20, 21]. Some others transform programs under certain general principles and, often, a set of rules are derived from such principles and are used for the transformation. Examples are the conventional strength reduction technique [1] and finite differencing technique [18] used in the APTS system [16, 17]. Seeking more flexibility than a fixed set of rules, KIDS [25] and CIP [19] use certain high-level strategies, but it is hard to automate them; also [26, 4] propose certain principles that can directly guide program transformations, but they employ heavy inference engines that are computationally exorbitant.

Our method of caching intermediate results integrates caching statically into the transformed programs under a general principle, namely, it transforms the program to cache the results of previous subcomputations that are not retrievable from the previous result but are useful for performing the new computation incrementally. The method is not limited to using a fixed set of rules. It is algorithmic and automatable. Although we present our method in a first-order functional language, the underlying principle is general and can be applied to other languages as well. A prototype system CACHET based on the approach is under development.

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

[11] Y. A. Liu. Deriving incremental programs. In *Towards The Future-Proceedings of Third International Conference for Young Computer Scientists*, pages 9.57–9.60, Beijing, China, July 1993. Tsinghua University Press.

[12] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on on PEPM*, La Jolla, California, June 1995.

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

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

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

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