Systematic Derivation of Incremental Programs

Yanhong A. Liu* Tim Teitelbaum*

August 1994

Abstract

A systematic approach is given for deriving incremental programs from non-incremental programs written in a standard functional programming language. We exploit a number of program analysis and transformation techniques and domain-specific knowledge, centered around effective utilization of caching, in order to provide a degree of incrementality not otherwise achievable by a generic incremental evaluator.

1 Introduction

Incremental programs take advantage of repeated computations on inputs that differ only slightly from one another, avoiding unnecessary duplication of common computations. Given a program f and a certain input change $\oplus$, a program f' that computes the value of $f(x \oplus y)$ efficiently by making use of the value of $f(x)$ is called an incremental version of f under $\oplus$. The parameter y can be regarded as a change δx to the input x. Methods of incremental computation have widespread applications, e.g., loop optimizations in optimizing compilers [1, 25, 9, 10] and transformational programming [27, 39], interactive systems like editors [3, 36] and programming environments [35, 22], and dynamic systems like distributed databases [8, 20] and real-time systems [46].

A comprehensive guide to the literature on incremental computation has appeared in [34]. Despite the relatively diverse categories discussed in [34], most of the work can be divided into three classes.

The first class includes particular incremental algorithms designed for particular problems dealing with particular input changes. Examples are incremental parsing [15, 18], incremental attribute evaluation [36, 49], incremental data-flow analysis [38], incremental circuit evaluation [2], incremental constraint solving [45, 13], *etc.* The study of dynamic graph algorithms, e.g., [50], can be viewed as falling into this class. Although efforts in this class are directed towards particular incremental algorithms, they apply to a broad class of problems, e.g., any attribute grammar, any circuit, *etc.*

In the second class, rather than manually developing particular incremental algorithms, application programs are run in a general incremental execution framework so that incremental computation is achieved automatically, e.g., incremental attribute evaluation frameworks [35], incremental computation via function caching [33], formal program manipulations using traditional partial evaluation [43, 42], incremental lambda reduction [12], the change detailing network of INC [51], incremental computation as a program abstraction [16], *etc.* In this class, often no explicitly incremental version of an application program is derived and run autonomously by a standard evaluator. Moreover, any input change to an application program is mapped to whatever the framework can handle, which is fixed for each framework. Therefore, these solutions to the incremental computation problem for particular applications are not readily comparable with explicitly derived incremental algorithms such as those in the first class.

In the third class, systematic approaches are studied to derive explicitly incremental programs from non-incremental programs using program transformation techniques like finite differencing [26, 31]. Examples

*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

are high level iterators [10], finite differencing of set expressions in SETL [31], optimizing relational database operations [17, 28], incremental fixed point computation [7], differentiation of functional programs in KIDS [39, 40], *etc.* In most of these works, programs are written in very high-level languages with aggregate data structures, e.g., sets and bags, and fixed rules are offered for transforming aggregate operations into more efficient incremental operations. In other work, only high-level strategies are proposed. What is not provided is an effective procedure for deriving incremental programs from non-incremental programs written in a standard language like Lisp.

Our work attacks the problem of discovering incrementality for programs written in a standard functional programming language. We give an effective procedure for deriving incremental programs from non-incremental programs written in a standard functional programming language. The basic derivation idea is to expand the computation of f on the new input so that subcomputations whose values can be efficiently retrieved from the previously computed result of f are replaced by corresponding retrievals. We exploit a number of program analysis and transformation techniques and domain-specific knowledge, centered around effective utilization of caching, in order to provide a degree of incrementality not otherwise achievable by a generic incremental evaluator. We also show how our approach can be extended to address caching auxiliary information for increasing incrementality.

The paper is organized as follows. Section 2 defines the derivation problem. Section 3 outlines the derivation procedure. Basic techniques for incrementalization by simplification and replacement are described in Section 4, with emphasis on techniques for discovering incrementality. Analysis and manipulation of recursive function applications are discussed in Section 5. Section 6 summarizes the derivation procedure and addresses a number of important issues, including correctness, termination, auxiliary information, and mechanization. A number of examples are given in Section 7. Finally, we compare our approach with closely related work and conclude in Section 8.

2 Defining the Problem

For simplicity of exposition, we use a simple first-order functional programming language. The expressions of our language are given by the following grammar:

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

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 $x = \langle x_1, ..., x_n \rangle$.

Example definitions are given in Figure 1. Each constructor c, primitive function p, and user-defined function f has a fixed arity. In general, $\bar{c_i}^1$ denotes the i-th selector corresponding to the constructor c. The semantics of the language is strict.

An input change $\oplus$ to the 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, in Figure 1, an input change $\oplus$ to the function out can be defined by $\langle C', R' \rangle = \langle C, R \rangle \oplus \langle i, a \rangle = \langle C, insert(i, a, R) \rangle$. For typographical convenience, we shall always use x to refer to the previous input to f_0, r the cached result of $f_0(x)$, and y the change parameter to the input x.

We are only interested in using a cached result if we can save time by doing so. Accordingly, we need a time model $\mathcal{T}$ such that $\mathcal{T}(e)$ describes the time needed to compute expression e. The function $\mathcal{T}$ can be obtained from standard constructions [47, 37]. In general, given two expressions e_1 and e_2, it is not decidable whether e_2 computes faster than e_1 for given values of their variables. Therefore, we say $\mathcal{T}(e_2) \leq \mathcal{T}(e_1)$ if

$out(C,R)$: compute the outer product of lists C and R

$$
\begin{aligned}
out(C,R) &= \textbf{if } null(C) \textbf{ then } nil \\
&\quad\ \textbf{else } cons(row(car(C),R),\ out(cdr(C),R)) \\
row(c,R) &= \textbf{if } null(R) \textbf{ then } nil \\
&\quad\ \textbf{else } cons(c * car(R),\ row(c,cdr(R)))
\end{aligned}
$$

$insert(i,a,R)$: insert a in list R at position i

$$
\begin{aligned}
insert(i,a,R) = &\ \textbf{if } i \leq 1 \textbf{ then } cons(a,R) \\
&\ \textbf{else if } null(R) \textbf{ then } cons(a,nil) \\
&\ \textbf{else } cons(car(R), \\
&\qquad\qquad insert(i-1,a,cdr(R)))
\end{aligned}
$$

Figure 1: Example function definitions

we can *effectively* confirm the inequality. Suppose $v_1,...,v_k$ are all the variables in e_1 and e_2 and P is some predicate on these variables, we write

$$t(e_2) \leq_P t(e_1) \tag{2}$$

to denote that we can effectively decide there is a constant k such that, for any values of $v_1,...,v_k$, if P holds then $\mathcal{T}(e_2) \leq k\mathcal{T}(e_1)$, and we say that e_2 is *asymptotically at least as fast* as e_1. During our derivation, P always represents the equations that hold at the occurrence of the expression currently under consideration; therefore it will be omitted for simplicity.

Given a program f_0 and an input change $\oplus$, we aim to derive 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] Obviously, we can trivially define $f_0'(x,y,r)$ to be $f_0(x \oplus y)$, but this is of no interest. The goal is to make f_0' as efficient as possible by having it use the cached result r of $f_0(x)$ as much as possible.

We will use the example in Figure 1 as a running example. At the end, we will obtain the incremental functions shown in Figure 2.

If $out(C,R)$ returns r, then
$out'(C,i,a,r)$ computes $out(C,insert(i,a,R))$.

For C of length m and R of length n,
$out'(C,i,a,r)$ takes time $O(m * \min(i,n))$;
$out(C,insert(i,a,R))$ takes time $O(m*n)$.

$$
\begin{aligned}
out'(C,i,a,r) &= \textbf{if } null(r) \textbf{ then } nil \\
&\quad\ \textbf{else } cons(row'(car(C),i,a,car(r)), \\
&\qquad\qquad out'(cdr(C),i,a,cdr(r))) \\
row'(c,i,a,r_1) &= \textbf{if } i \leq 1 \textbf{ then } cons(c*a,\ r_1) \\
&\quad\ \textbf{else if } null(r_1) \textbf{ then } cons(c*a,\ nil) \\
&\quad\ \textbf{else } cons(car(r_1), \\
&\qquad\qquad row'(c,i-1,a,cdr(r_1)))
\end{aligned}
$$

Figure 2: Derived function definitions

3 Overview of the Derivation Procedure

The basic derivation idea is to symbolically expand the computation of $f_0(x \oplus y)$ and replace subcomputations whose values can be efficiently retrieved from the cached result r of $f_0(x)$ by corresponding retrievals.

Derivation Procedure. The derivation procedure recursively follows function applications in the computation of $f_0(x \oplus y)$ and aims to replace these applications by uses of new functions introduced to compute the applications incrementally.

To introduce a new function f' to compute a function application $f(e_1,...,e_n)$ incrementally, we collect an information set I_f describing the context of the application, and a cache set C_f indicating how the values of certain relevant computations can be retrieved from a cached result under certain conditions. Then, we

[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 as discussed in Section 5.

obtain a definition of f' by the following three steps. First, we unfold [6] (also called expand [48]) the application. Second, we incrementalize the unfolded application. Basically, we consider each subexpression e of the unfolded application in applicative order and (a) collect an information set $I_{[e]}$ from e's context based on I_f, and extend the cache set C_f under the condition that the facts in $I_{[e]}$ are valid, (b) recursively apply this procedure if e is a function application, (c) apply simplification using $I_{[e]}$ and replacement by efficient retrieval using the extended C_f. Third, we eliminate dead code mainly related to dead parameters of f'. If the function f' so obtained is suitably fast, then $f(e_1, ..., e_n)$ can be replaced by an application of f'. Other applications of f that are subsequently analyzed may also be replaced by applications of this f', if appropriate.

The derivation procedure starts by considering the function application $f_0(x \oplus y)$, with an empty information set and a cache set containing only $f_0(x) = r$. We maintain a global data structure for the set D of functions introduced during the derivation procedure. We take special care of recursive function applications to help the derivation procedure terminate naturally and, at the same time, discover as much incrementality as possible. When finished, we have the original set of functions F and the set D of functions introduced during the derivation procedure, including f'_0. We then eliminate dead functions in F and D not needed in computing f'_0.

A function $\mathcal{I}nc\mathcal{A}pply$ implements the recursive procedure on a function application $f(e_1, ..., e_n)$ with information set I, cache set C, and global definition set D:

$$\mathcal{I}nc\mathcal{A}pply[\![f(e_1, ..., e_n)]\!]\; I\; C\; D = \langle f'(e'_1, ..., e'_m),\;\; D' \rangle \tag{3}$$

f' is an introduced function such that $f'(e'_1, ..., e'_m)$ computes $f(e_1, ..., e_n)$ incrementally under I and C. The global set D may be extended to D' as a side effect.

Two Main Issues. The derivation procedure has two main tasks. First, incrementalizing an unfolded function application, i.e., discovering and replacing subcomputations whose values can be efficiently retrieved from cached results. Second, analyzing recursive function applications and introducing incremental versions that are used to replace these applications.

The first task corresponds to maintaining cache sets under collected information sets at subexpressions of an unfolded application and applying simplification and replacement to these subexpressions using these sets. The second task corresponds to maintaining a global set of functions introduced to compute function applications incrementally and replacing function applications with appropriate applications of these introduced functions.

The two main issues are addressed in Sections 4 and 5, respectively. Finally, the derivation procedure is summarized in Section 6.

4 Incrementalization

We define two notions, *information sets* and *cache sets*. Given an information set I and an initial cache set C relevant to a function application, we describe how to use them in incrementalizing the unfolded application, i.e., collecting information sets at subexpressions, extending the cache set with respect to the collected information sets, and using them to simplify subexpressions and replace subexpressions whose values can be efficiently retrieved from cached results.

4.1 Information Sets and Simplification

An information set $I_{[e]}$ at the occurrence of an expression e is a collection of equations that hold 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 $f_0(x \oplus y)$

is unfolded to be expression e as follows:

$$\textbf{let } v = e_1 \textbf{ in } (\textbf{ if } v = 0 \textbf{ then } 0 \textbf{ else } e_2 \,) \textbf{ end}$$

then $I_{[e]} = \emptyset$, $I_{[e_1]} = \emptyset$, $I_{[v=0]} = \{v \leftrightarrow e_1\}$, and $I_{[e_2]} = \{v \leftrightarrow e_1, v = 0 \leftrightarrow F\}$.

Given an information set at a top-level expression, it is simple to compute information sets at occurrences of subexpressions:

if e is $g(e_1, ..., e_n)$, where g is a constructor or a (primitive) function, then $I_{[e_i]} = I_{[e]}$ for $i = 1..n$;
if e is **if** e_1 **then** e_2 **else** e_3, then $I_{[e_1]} = I_{[e]}$, $I_{[e_2]} = I_{[e]} \cup \{e_1 \leftrightarrow T\}$, and $I_{[e_3]} = I_{[e]} \cup \{e_1 \leftrightarrow F\}$;
if e is **let** $v = e_1$ **in** e_2 **end**, then $I_{[e_1]} = I_{[e]}$, and $I_{[e_2]} = I_{[e]} \cup \{v \leftrightarrow e_1\}$.

An underlying logic L_0 is used to make inferences based on the facts in an information set. We require that L_0 be *compatible* with the semantics of the programming language we are using [14], i.e., if two expressions are proved to be equal under L_0, then they compute the same value. In this paper, we assume that a theorem prover based on L_0 is available, and we write $e_1 \leftrightarrow_I^* e_2$ to denote that a finite proof that e_1 equals e_2 can be found by the theorem prover using equations in set I.

Simplification. We can simplify expressions using information sets and the underlying logic, as summarized in Figure 3. Given an expression e and an information set I, we say *e can be simplified under I to e'* if the corresponding condition $cond(I)$ holds.

expression e	expression e'	condition $cond(I)$
v	c	$v \leftrightarrow_I^* c$
$c(e_1, ..., e_n)$	e_c	$e \leftrightarrow_I^* c(\bar{c}_1^{-1}(e_c), ..., \bar{c}_n^{-1}(e_c))$ and $t(e_c) \le t(e)$
$p(e_1, ..., e_n)$	e_p	e can be simplified to e_p under I using properties of p
$f(e_1, ..., e_n)$	$e_f\,[e_1/v_1, \ldots, e_n/v_n]$	e can be unfolded under I and f is defined as $f(v_1,...,v_n) = e_f$
if e_1 **then** e_2 **else** e_3	e_2	$e_1 \leftrightarrow_I^* T$
	e_3	$e_1 \leftrightarrow_I^* F$
	e_2 (or e_3)	$e_2 \leftrightarrow_I^* e_3$ and $t(e_2) \le t(e_3)$ (or $t(e_3) \le t(e_2)$)
let $v = e_1$ **in** e_2 **end**	$e_2[v_1/v]$	$e_1 \leftrightarrow_I^* v_1$
	$e_2[e_1/v]$	e can be unfolded under I

Figure 3: Simplification

Basically, the simplification is as conventional, except with the identity relation generalized everywhere to the equality under I relation. Simplification of a function application $f(e_1, ..., e_n)$ unfolds the application if the resulting expression (or its context) can be computed as least as fast (through appropriate simplification corresponding to the context). To automate this in practice, heuristic conditions such as the followings are used:

1) f is not recursively defined, and unfolding $f(e_1, ..., e_n)$ does not duplicate non-trivial computations, i.e., for each i, either $t(e_i) \le t(v_i)$ or v_i occurs at most once on every (syntactic) execution path in e_f.

2) $f(e_1, ..., e_n)$ is an argument to a primitive function p, and this application of p can be simplified after unfolding f using properties of p.

Simplification of a binding expression **let** $v = e_1$ **in** e_2 **end** unfolds it to $e_2[v_1/v]$ if e_1 equals a variable v_1 under I. Otherwise, it treats the expression as $f(e_1)$, with f defined by $f(v) = e_2$.

Given an expression e and an information set I, we define $\mathcal{S}imp[\![e]\!]I$ to be e' if e can be simplified under I to e', and e otherwise.

4.2 Cache Sets and Replacement

A cache set C for an unfolded application is a set of tuples $\langle e_1, e_2, I\rangle$ such that

1) expression e_1 depends only on x, expression e_2 depends only on r, and,
2) if the equations in information set I hold, then e_1 and e_2 compute the same value.

For example, if $f_0(x \oplus y)$ is unfolded to be e, then the initial cache set for e is $\{\langle f_0(x), r, \emptyset\rangle\}$.

Intuitively, an element $\langle e_1, e_2, I\rangle$ in a cache set C says that if the equations in I hold, then the value of e_1 can be retrieved from a cached result by computing e_2. Given a cache set and an occurrence of an expression e with information set $I_{[e]}$, we can extend the cache set at e under $I_{[e]}$. This extension requires techniques for discovering more expressions whose values can be retrieved from cached results, i.e., discovering incrementality, as described below.

Discovering Incrementality. The schematic diagrams of Figure 4 help explain the basic ideas. The leftmost rectangle depict the expanded computation of $f_0(x \oplus y)$. Clearly, if $f_0(x)$ occurs anywhere as a

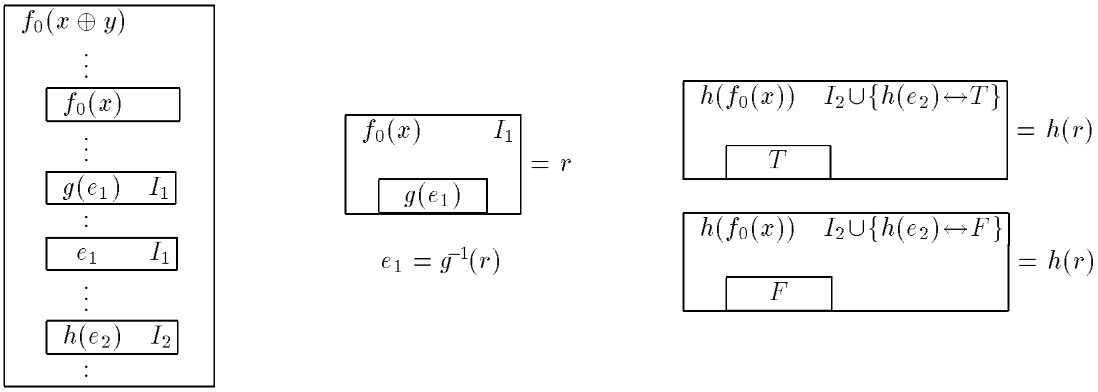

Figure 4: Discovering incrementality

subcomputation, then its value can be straightforwardly retrieved from r. However, we seek to discover other subcomputations whose values can also be retrieved from r. Suppose $g(e_1)$ occurs somewhere as a subcomputation and it is not $f_0(x)$. If we collect the context information I_1 at the occurrence of $g(e_1)$, and find that $f_0(x)$ can be specialized to $g(e_1)$ under I_1, as depicted in the middle rectangle, then the value of $g(e_1)$ at the occurrence can also be retrieved from r. Moreover, if g is a function with an inverse g^{-1}, then the value of e_1 can be retrieved from $g^{-1}(r)$, wherever I_1 holds. In a special situation, suppose $h(e_2)$ occurs as a subcomputation but neither $h(e_2)$ nor e_2 is $f_0(x)$, $g(e_1)$, or e_1. If h is a Boolean valued function defined on all inputs, and $h(f_0(x))$ can be specialized to true (false) when $h(e_2)$ equals true (false), as depicted in the right rectangles, then the value of $h(e_2)$ can be retrieved from $h(r)$.

The specializations shown in the middle and right rectangles in Figure 4 employ an auxiliary specializer $\mathcal{G}$. Given an expression e and an information set I, $\mathcal{G}[\![e]\!]I$ specializes e under I, and whenever e computes a value, $\mathcal{G}[\![e]\!]I$ computes the same value. The specialization is achieved by unfolding and simplification, and will be defined at the end of this section. Here, we use $\mathcal{G}$ as a subroutine to help discover more subcomputations whose values can be retrieved from cached results.

We formalize the basic ideas as follows. Given a cache set C and an occurrence of an expression e with information set $I_{[e]}$, we can extend C at e under $I_{[e]}$ to be $\mathcal{C}(C, e, I_{[e]})$, where $\mathcal{C}(C, e, I)$ is called the *closure of C at e under I* and is defined as follows. Given C, e, and I, let

$$\begin{array}{l} F_1(C) = \{\langle e_1', \quad e_2, \quad I\rangle \mid \langle e_1, e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![e_1]\!]I = e_1'\}, \\ F_2(C) = \{\langle e_1', g^{-1}(e_2), I\rangle \mid \langle e_1, e_2, I'\rangle \in C,\ I \Rightarrow I',\ e_1 = g(e_1')\},\ \text{and} \\ F_3(C) = \{\langle e,\ \ h(e_2),\ I\rangle \mid \langle e_1, e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![h(e_1)]\!](I \cup \{e \leftrightarrow {}^T_F\}) = {}^T_F\}, \end{array} \tag{4}$$

where g is a constructor or primitive function that has a known inverse g^{-1}, and h is a Boolean valued function defined on all inputs. Note that g and h can be straightforwardly generalized to include functions with two or more parameters. As a particular example for F_2, if $e_1 = c(e_1', ..., e_n')$, then $\langle e_i', \bar{c}_i^{-1}(e_2), I\rangle \in F_2(C)$ for $i = 1..n$. Just as $F_2(C)$ extends C when e_1 is a constructor or a special primitive function application, the following sets extend C when e_1 is a binding expression or a special conditional expression:

$$\begin{array}{l} F_{21}(C) = \{\langle e_2'', e_2, I\rangle \mid \langle \mathbf{let}\ v = e_1'\ \mathbf{in}\ e_2'\ \mathbf{end}, e_2, I'\rangle \in C,\ I \Rightarrow I',\ e_2'[e_1'/v] = e_2''\}, \\ F_{22}(C) = \{\langle e_3', e_2, I\rangle \mid \langle \mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3', e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![e_3']\!](I \cup \{e_1' \leftrightarrow T\}) = e_2'\}, \\ F_{23}(C) = \{\langle e_2', e_2, I\rangle \mid \langle \mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3', e_2, I'\rangle \in C,\ I \Rightarrow I',\ \mathcal{G}[\![e_2']\!](I \cup \{e_1' \leftrightarrow F\}) = e_3'\}. \end{array} \tag{5}$$

We let $F_2(C)$ also include the elements in these sets, and define $\mathcal{C}(C, e, I)$ to be $C \cup C' \cup F_3(C \cup C')$, where C' is the least set such that $F_1(C) \subseteq C'$ and $F_2(C') \subseteq C'$.

The set C' can be computed using a worklist algorithm. First, initialize C' to be $\emptyset$ and worklist L to be $F_1(C)$. Then, repeatedly move any element $\langle e_1, e_2, I'\rangle$ from L to C' and, if e_1 is $g(e_1')$, add $\langle e_1', g^{-1}(e_2), I'\rangle$ into L, if e_1 is **let** $v = e_1'$ **in** e_2' **end**, add $\langle e_2'[e_1'/v], e_2, I'\rangle$ into L, *etc.* This stops when L is empty, at which time we have obtained the final set C'. Optimizations to the computation of the closure are possible. For example, we can group elements that have the same information sets into units, and maintain a tree of these units so that, if $I_1 \Rightarrow I_2$, then the unit with I_1 is a descendent of the one with I_2. Every time we compute the closure, we only need to look at elements in the units that are closest to the leaves and whose information sets are implied by the current information set.

Example. Using the example in Figure 1, let e be the unfolded application of $out(C, insert(i, a, R))$

$$\begin{array}{l} \mathbf{if}\ null(C)\ \mathbf{then}\ nil \\ \mathbf{else}\ cons(row(car(C), insert(i, a, R)),\ out(cdr(C), insert(i, a, R))) \end{array} \tag{6}$$

with information set $I_{[e]} = \emptyset$ and initial cache set $C_{out} = \{\langle out(C,R), r, \emptyset\rangle\}$.

Let e_1 be the false branch of e. Given $C_{out} = \{\langle out(C,R), r, \emptyset\rangle\}$, consider extending C_{out} at e_1 with $I_{[e_1]} = \{null(C) \leftrightarrow F\}$. Specializing $out(C, R)$ under $I_{[e_1]}$, we get the expression e_2 below:

$$cons(row(car(C), R),\ out(cdr(C), R)) \tag{7}$$

Thus,

$$\mathcal{C}(C_{out}, e_1, I_{[e_1]}) = C_{out} \cup \{\langle e_2, r, I_{[e_1]}\rangle,\ \langle row(car(C),R), car(r), I_{[e_1]}\rangle,\ \langle out(cdr(C),R), cdr(r), I_{[e_1]}\rangle\}. \tag{8}$$

Given $C_{out} = \{\langle out(C,R), r, \emptyset\rangle\}$, consider extending C_{out} at the Boolean subexpression $null(C)$ in e with $I_{[null(C)]} = \emptyset$. When $null(C) \leftrightarrow T$ holds, $out(C, R)$ is specialized to nil and thus $null(out(C, R))$ equals T; when $null(C) \leftrightarrow F$ holds, $out(C, R)$ is specialized to e_2 and thus $null(out(C, R))$ equals F. Thus

$$\mathcal{C}(C_{out}, null(C), \emptyset) = C_{out} \cup \{\langle null(C), null(r), \emptyset\rangle\}. \tag{9}$$

Replacement. We say that expression e *can be replaced by* e' *under* I *and* C, denoted as $e \rightarrow^*_{IC} e'$, if

$$(\,\exists \langle e_1, e', I_1\rangle \in C\,)\ [\,e \leftrightarrow^*_I e_1 \ \wedge\ I \Rightarrow I_1 \ \wedge\ t(e') \le t(e)\,].$$

Given a non-conditional and non-binding expression e, an information set I, and a cache set C, if e can be replaced by e' under I and C, then we do so. Otherwise, we extend the cache set to be $\mathcal{C}(C,e,I)$, and, if e can be replaced by e'' under I and the extended cached set $\mathcal{C}(C,e,I)$, then we do so. As a result, the cache set may be extended as a side effect of a replacement. We define a function $\mathcal{R}epl$ for replacement as follows:

$$\mathcal{R}epl[\![e]\!]IC = \begin{cases} \langle e, C\rangle & \text{if } e \text{ is a conditional or binding expression} \\ \langle e', C\rangle & \text{else if } e \rightarrow^*_{IC} e' \\ \langle e'', C'\rangle & \text{else if } e \rightarrow^*_{IC'} e'',\ \ \text{where } C' = \mathcal{C}(C,e,I) \\ \langle e, C'\rangle & \text{otherwise,}\ \ \text{where } C' \text{ is as above} \end{cases} \tag{10}$$

Another use of a cache set for replacement is as follows. Suppose an expression e can not be replaced by any expression under I and C, but

$$\exists\,\langle \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3,\ e_4,\ I_1\rangle \in C, \quad I \Rightarrow I_1$$

such that e can be replaced by e_T (respectively e_F) under $I \cup \{e_1 \leftrightarrow T\}$ (respectively $I \cup \{e_1 \leftrightarrow F\}$) and the correspondingly extended cache set. Then we can replace e by $\textbf{if}\, e_1 \,\textbf{then}\, e_T \,\textbf{else}\, e_F$ provided $t(\textbf{if}\, e_1 \,\textbf{then}\, e_T \,\textbf{else}\, e_F) \le t(e)$. For example, if e is e_3, and e_1 takes unit time, then we can replace e by $\textbf{if}\, e_1 \,\textbf{then}\, e \,\textbf{else}\, e_4$. We extend the function $\mathcal{R}epl$ for replacement to include this case, i.e., we replace the last case of (10) by the following two cases:

$$\begin{cases} \langle \textbf{if } e_1' \textbf{ then } e_T \textbf{ else } e_F,\ C'''\rangle \\ \qquad \text{if } \exists\,\langle \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3,\ e_4,\ I_1\rangle \in C',\ I \Rightarrow I_1,\ t(\textbf{if } e_1' \textbf{ then } e_T \textbf{ else } e_F) \le t(e) \\ \qquad\qquad \text{where } e_1' = \begin{cases} e_1'' & \text{if } e_1 \rightarrow^*_{IC''} e_1'' \\ e_1 & \text{otherwise} \end{cases},\ \ \text{where } C'' = \mathcal{C}(C', e_1, I) \\ \qquad\qquad\qquad e_B = \begin{cases} e_B' & \text{if } e \rightarrow^*_{I_B C'''} e_B' \\ e & \text{otherwise} \end{cases},\ \ \text{where } I_B = I \cup \{e_1' \leftrightarrow B\},\ \ \text{for } B = T, F \\ \qquad\qquad\qquad C''' = \mathcal{C}(C'', e, I_T) \cup \mathcal{C}(C'', e, I_F) \\ \langle e, C''''\rangle \quad \text{otherwise,}\ \ \text{where } C'''' \text{ is } C''' \text{ as above if it is computed and } C' \text{ otherwise} \end{cases} \tag{11}$$

4.3 Incrementalization Using Simplification and Replacement

To incrementalize an unfolded application, a function $\mathcal{I}nc$ applies simplification and replacement on subexpressions in applicative order. The cache set for the current unfolded application may be extended as a side effect of replacement using $\mathcal{R}epl$. In particular, $\mathcal{I}nc$ calls $\mathcal{I}nc\mathcal{A}pply$ to consider subexpressions that are function applications. The global set of introduced functions may be extended as a side effect of using $\mathcal{I}nc\mathcal{A}pply$.

We refer to the application of simplification and replacement by $\mathcal{I}nc$ as *reduction*. Thus $\mathcal{I}nc$ does innermost leftmost reduction. If a subexpression is reduced to a conditional expression, then the condition is lifted out of the enclosing-expression. Similarly, if a subexpression is reduced to a binding expression, then the binding is lifted. A function $\mathcal{S}ubl$ is used by $\mathcal{I}nc$ to recursively reduce subexpressions and perform necessary lifting, as defined in Figure 5.

The presentation of $\mathcal{S}ubl$ is simplified by omitting detailed control structures that sequence $\mathcal{S}ubl$ through its subexpressions. We just present the case of $\mathcal{S}ubl$ working on the ith subexpression of the top-level construct and condition it on that the 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 $\mathcal{I}nc$ for the subexpression at that position; otherwise, it is not reduced. For a conditional expression $\textbf{if}\, e_1 \,\textbf{then}\, e_2 \,\textbf{else}\, e_3$, $\mathcal{I}nc$ reduces e_2 (respectively e_3) with the assumption that e_1 equals true (respectively false) added into the information set.

Name	Transformation	Condition
(g)	$\mathcal{S}ubl[\![g(e_1, ..., e_n)]\!]\ I\ C\ D$ where g is c, p, or f	
(g_i)	$= \mathcal{S}ubl[\![g(e_1, ..., e_{i-1}, e'_i, e_{i+1}, ..., e_n)]\!]\ I\ C'\ D'$	if $e_1, ..., e_{i-1}$ are reduced, not **if** or **let**, but
	where $\langle e'_i, C', D' \rangle = \mathcal{I}nc[\![e_i]\!]\ I\ C\ D$	e_i is not reduced
(g_{i-if})	$= \mathcal{S}ubl[\![$**if** e'_1 **then** $g(e_1, ..., e_{i-1}, e'_2, e_{i+1}, ..., e_n)$	if $e_1, ..., e_{i-1}$ are reduced, not **if** or **let**,
	else $g(e_1, ..., e_{i-1}, e'_3, e_{i+1}, ..., e_n)]\!]\ I\ C\ D$	e_i is reduced, but e_i is **if** e'_1 **then** e'_2 **else** e'_3
	where e'_1 is reduced and is not **if** or **let**	
(g_{i-let})	$= \mathcal{S}ubl[\![$**let** $v = e'_1$ **in** $g(e_1, ..., e_{i-1}, e'_2, e_{i+1}, ..., e_n)$ **end**$]\!]\ I\ C\ D$	if $e_1, ..., e_{i-1}$ are reduced, not **if** or **let**,
	where e'_1 is reduced and is not **if** or **let**	e_i is reduced, but e_i is **let** $v = e'_1$ **in** e'_2 **end**
(g_n)	$= \langle g(e_1, ..., e_n), C, D \rangle$	otherwise
(if)	$\mathcal{S}ubl[\![$**if** e_1 **then** e_2 **else** $e_3]\!]\ I\ C\ D$	
(if_1)	$= \mathcal{S}ubl[\![$**if** e'_1 **then** e_2 **else** $e_3]\!]\ I\ C'\ D'$	if e_1 is not reduced
	where $\langle e'_1, C', D' \rangle = \mathcal{I}nc[\![e_1]\!]\ I\ C\ D$	
(if_{1-if})	$= \mathcal{S}ubl[\![$**if** e'_1 **then** (**if** e'_2 **then** e_2 **else** e_3)	if e_1 is reduced, and e_1 is **if** e'_1 **then** e'_2 **else** e'_3
	else (**if** e'_3 **then** e_2 **else** e_3)$]\!]\ I\ C\ D$	
	where e'_1 is reduced and is not **if** or **let**	
(if_{1-let})	$= \mathcal{S}ubl[\![$**let** $v = e'_1$ **in** (**if** e'_2 **then** e_2 **else** e_3) **end**$]\!]\ I\ C\ D$	if e_1 is reduced, and e_1 is **let** $v = e'_1$ **in** e'_2 **end**
	where e'_1 is reduced and is not **if** or **let**	
(if_n)	$= \langle$**if** e_1 **then** e'_2 **else** e'_3, C'', $D'' \rangle$	otherwise
	where $\langle e'_2, C', D' \rangle = \begin{cases} \langle e_2, C, D \rangle & \text{if } e_1 \leftrightarrow^*_I F \\ \mathcal{I}nc[\![e_2]\!]\ (I \cup \{e_1 \leftrightarrow T\})\ C\ D & \text{otherwise} \end{cases}$	
	$\langle e'_3, C'', D'' \rangle = \begin{cases} \langle e_3, C', D' \rangle & \text{if } e_1 \leftrightarrow^*_I T \\ \mathcal{I}nc[\![e_3]\!]\ (I \cup \{e_1 \leftrightarrow F\})\ C'\ D' & \text{otherwise} \end{cases}$	
(let)	$\mathcal{S}ubl[\![$**let** $v = e_1$ **in** e_2 **end**$]\!]\ I\ C\ D$	
(let_1)	$= \mathcal{S}ubl[\![$**let** $v = e'_1$ **in** e_2 **end**$]\!]\ I\ C'\ D'$	if e_1 is not reduced
	where $\langle e'_1, C', D' \rangle = \mathcal{I}nc[\![e_1]\!]\ I\ C\ D$	
(let_{1-if})	$= \mathcal{S}ubl[\![$**if** e'_1 **then** (**let** $v = e'_2$ **in** e_2 **end**)	if e_1 is reduced, and e_1 is **if** e'_1 **then** e'_2 **else** e'_3
	else (**let** $v = e'_3$ **in** e_2 **end**)$]\!]\ I\ C\ D$	
	where e'_1 is reduced and is not **if** or **let**	
(let_{1-let})	$= \mathcal{S}ubl[\![$**let** $v' = e'_1$ **in** (**let** $v = e'_2$ **in** e_2 **end**) **end**$]\!]\ I\ C\ D$	if e_1 is reduced, and e_1 is **let** $v' = e'_1$ **in** e'_2 **end**
	where e'_1 is reduced and is not **if** or **let**	
(let_n)	$= \langle$**let** $v = e_1$ **in** e'_2 **end**, C', $D' \rangle$	otherwise
	where $\langle e'_2, C', D' \rangle = \mathcal{I}nc[\![e_2]\!]\ (I \cup \{v \leftrightarrow e_1\})\ C\ D$	

Figure 5: Definition of $\mathcal{S}ubl$

Similarly, for a binding expression **let** $v = e_1$**in** e_2 **end**, $\mathcal{Inc}$ reduces e_2 with the assumption that v equals e_2 added into the information set. At the end, all subexpressions are reduced with necessary lifting performed.

Finally, we define the function $\mathcal{Inc}$ as in (12), where I^C denotes the set $I \cup \{e_1 \leftrightarrow e_2 \mid \langle e_1, e_2, I' \rangle \in C, I \Rightarrow I'\}$. We need I^C instead of I because $\mathcal{Inc}$ does applicative order reduction, during which some subexpressions may be replaced by retrievals, and thus the equations in I may involve the cache parameter. As a result, the underlying logic needs to know the equality relation involving the cache parameter to make inferences. For example, to reduce the expression e in (6), first $null(C)$ is reduced to $null(r)$ according to (9), and thus $I_{[e_1]} = \{null(r) \leftrightarrow F\}$ for the expression e_1 in the false branch of e. Now to specialize $out(C, R)$ at e_1, we use the information set $\{null(r) \leftrightarrow F\} \cup \{null(x) \leftrightarrow null(r)\}$, and we obtain the same expression as in (7).

$$\mathcal{Inc}[\![e]\!]ICD = \langle e''', C'', D'' \rangle, \quad \text{where} \quad \begin{array}{ll} \langle e''', C'' \rangle & = \mathcal{Repl}[\![\mathcal{Simp}[\![e'']\!]I^{C'}]\!]I^{C'}C' \\ \langle e'', D'' \rangle & = \begin{cases} \mathcal{IncApply}[\![e']\!]I^{C'}C'D' & \text{if } e' \text{ is } f(e_1, ..., e_n) \\ \langle e', D' \rangle & \text{otherwise} \end{cases} \\ \langle e', C', D' \rangle & = \begin{cases} \mathcal{Subl}[\![e]\!]ICD & \text{if } e \text{ is not } v \\ \langle e, C, D \rangle & \text{otherwise} \end{cases} \end{array} \tag{12}$$

The function $\mathcal{Inc}$ proceeds as follows. First, if an expression e has subexpressions, then $\mathcal{Inc}$ calls $\mathcal{Subl}$ to recursively reduce the subexpressions in turn. The cache set and definition set may be changed while reducing subexpressions. Then if the resulting expression is a function application, $\mathcal{Inc}$ calls $\mathcal{IncApply}$ and aims to replace the application with an application of an introduced function that computes incrementally. The definition set may be changed by $\mathcal{IncApply}$. Finally, $\mathcal{Inc}$ uses $\mathcal{Simp}$ to simplify the top-level expression, and then calls $\mathcal{Repl}$ to replace the resulting expression by a retrieval from a cached result, if possible. The cache set may be changed by $\mathcal{Repl}$.

Auxiliary Specializer. The auxiliary specializer $\mathcal{G}$ is defined in a way similar to $\mathcal{Inc}$, but is much simpler. It simplifies subexpressions in applicative order and lifts conditions and bindings as $\mathcal{Inc}$ does, but there are no cache sets or definitions sets involved. If simplification of a function application unfolds the application, then $\mathcal{G}$ is applied to the unfolded application. Let $\mathcal{Subl}'$ be $\mathcal{Subl}$ except that $\mathcal{Subl}'$ takes only an expression and an information set as arguments, returns only an expression, and calls $\mathcal{G}$ instead of $\mathcal{Inc}$. Then $\mathcal{G}$ is defined as follows:

$$\mathcal{G}[\![e]\!]I = \begin{cases} \mathcal{G}[\![e'']\!]I & \text{if } e' \text{ is } f(e_1, ..., e_n) \text{ and } e'' \neq e' \\ e'' & \text{otherwise} \end{cases}, \quad \text{where} \quad \begin{array}{ll} e'' & = \mathcal{Simp}[\![e']\!]I \\ e' & = \begin{cases} \mathcal{Subl}'[\![e]\!]I & \text{if } e \text{ is not } v \\ e & \text{otherwise} \end{cases} \end{array} \tag{13}$$

5 Manipulating Recursive Function Applications

We define the *definition set*, which is a global set of functions introduced during the derivation procedure to compute function applications incrementally. We describe how to maintain the definition set when introducing functions and how to use the introduced functions to replace appropriate function applications.

5.1 Definition Set

The definition set D is a set of tuples $\langle f(e_1, ..., e_n),\ f'(v_1, ..., v_k),\ C \rangle$, where C is a single-element cache set $\{\langle e_c, e_r, I \rangle\}$, such that

1) f is a function in the original set F, expressions $e_1, ..., e_n$ depend on x and possibly on y, f' is a new function introduced in the set D, and $v_1, ..., v_k$ are variables in $e_1, ..., e_n, e_c$, and e_r,

2) if the cache set C is valid, i.e., the equations in the information set I hold, and $e_c = e_r$, then whenever $f(e_1, ..., e_n)$ terminates with a value, $f'(v_1, ..., v_k)$ terminates with the same value, and

3) a definition of f' is obtained by incrementalizing the unfolded $f(e_1, ..., e_n)$ using I and C, and some of the parameters of f' may be dead and eliminated after the incrementalization.

For example, given $out(C, R) = r$ with empty information set at the initial application $out(C, insert(i, a, R))$, we introduce a new function out', and we get the initial definition set

$$\{\langle out(C, insert(i, a, R)),\ out'(C, i, a, R, r),\ \{\langle out(C, R), r, \emptyset\rangle\}\rangle\} \tag{14}$$

where a definition of out' is to be obtained by incrementalizing the unfolded $out(C, insert(i, a, R))$ using $out(C, R) = r$.

Intuitively, an element in the definition set D says that a new function f' is introduced such that, if the equations in the information set I hold, and the value of e_c can be retrieved from a cached result by computing e_r, then $f'(v_1, ..., v_k)$ computes $f(e_1, ..., e_n)$ incrementally. To obtain a definition of f', we unfold $f(e_1, ..., e_n)$, incrementalize the unfolded application using the sets I and C, and then eliminate redundant parameters. While we incrementalize the unfolded $f(e_1, ..., e_n)$, we may encounter other function applications before we obtain a final definition of f'. We say f' is *fully defined* if, for every introduced function g' in D that f' (transitively) depends on, a final definition of g' has been obtained.

Note the restriction that the cache set C contains only *one* element, which reflects our main heuristic for introducing new functions. In general, a function application has its context information set and a current cache set. Any element in these sets might be used in incrementalizing the unfolded application. But we do not know a *priori*, before examining the unfolded application, what elements are used and how. Therefore, any dynamic decision must be an approximation. Our one-cache-element heuristic is based on the observation that, in a well-structured program, a function application is expected to be computed incrementally based on the cached result of a corresponding previous computation. As a consequence of our way of choosing the single cache element, as described below, there is only one variable in the expression e_r. This variable depends on r and is introduced as a parameter of f'. We call it the *current cache parameter* during the process of incrementalizing the unfolded $f(e_1, ..., e_n)$.

A function f may correspond to multiple introduced functions, since there may be multiple occurrences of applications of f during the derivation, and different applications may correspond to different introduced functions.

5.2 Generalization for Function Introduction

Given a function application $f(e_1, ..., e_n)$, let I be the information set at $f(e_1, ..., e_n)$, and C the current cache set C for the unfolded application that contains $f(e_1, ..., e_n)$. To introduce a function f' to compute $f(e_1, ..., e_n)$ incrementally, the main task is to decide, based on I and C, a valid and relevant cache element that is to be used to incrementalize the computation of $f(e_1, ..., e_n)$. An interaction with this comes from using a version of *generalization* that enables f' to be used in more general settings and, at the same time, does not impede the discovery of incrementality.

Considerations. Our use of generalization ignores substructures of expressions to introduce functions for more general uses. For example, consider the function application

$$row(car(C), insert(i, a, R)) \quad \text{with} \quad \langle row(car(C), R),\ car(r),\ \{null(C) \leftrightarrow F\}\rangle \in C_{out} \tag{15}$$

and $I = \{null(C) \leftrightarrow F\}$ in the false branch of (6). Instead of introducing

$$\langle row(car(C), insert(i, a, R)),\ row'(C, i, a, R, r),\ \{\langle row(car(C), R),\ car(r),\ \{null(C) \leftrightarrow F\}\rangle\}\rangle \tag{16}$$

and replacing the application by $row'(C, i, a, R, r)$, we introduce

$$\langle row(c, insert(i, a, R)),\ row'(c, i, a, R, r_1),\ \{\langle row(c, R),\ r_1,\ I'\rangle\}\rangle \tag{17}$$

where $I' = \{null(C) \leftrightarrow F, car(C) \leftrightarrow c\}$, and replace the application by $row'(car(C), i, a, R, car(r))$. We say that c generalizes $car(C)$, and r_1 generalizes $car(r)$. Obviously, the latter row' is more general than the former and can be used in more general settings.

Basically, the largest common super-expression of all occurrences of a variable is generalized by a single (new) variable. However, there are two considerations. First, generalization should not impede the discovery of incrementality. For example, if we consider $row(car(C), insert(i, a, R))$ in (15), then $insert(i, a, R)$ is not generalized by a variable, since we want to separate subcomputations depending only on x from the rest so that the former can possibly be replaced by retrievals. Therefore, one guideline is to generalize as much as possible, but not cross the boundary between subexpressions depending only on x and the rest.

The second consideration is associated with the main task of deciding a valid and most relevant cache element to be used to incrementalize the computation of $f(e_1, ..., e_n)$. For example, among the valid cache elements in (8), the element in (15) is used to incrementalize $row(car(C), insert(i, a, R))$. To arrive at this choice, consider $row(car(C), insert(i, a, R))$ together with the two expressions in a cache element. With the element in (15), we can generalize more than with any other element in (8). Also, the information set becomes I', as in (17), since it relates I with the new variable c. Therefore, the guideline is to generalize the function application together with the two expressions in each valid cache element, choose the element that allows most generalization, and relates the information set with the new variables.

To summarize, our use of generalization does not impede the discovery of incrementality and helps obtain the most relevant cache element. We should note that these are online techniques for the generalization.

Generalization. We present the above ideas formally as follows. Given expressions $e_1, ..., e_m$, let $u_1, ..., u_k$ be all the variables in them. Let $\{u_l, u_{j_1}, ..., u_{j_h}\} \subseteq \{u_1, ..., u_k\}$ but $u_l \notin \{u_{j_1}, ..., u_{j_h}\}$. An expression e is the *largest common* $u_l \backslash \{u_{j_1}, ..., u_{j_h}\}$*-cover expression* of $e_1, ..., e_m$ if e is the largest common super-expression of all occurrences of u_l in the e_i's, such that $u_{j_1}, ..., u_{j_h}$ do not appear in e.

Given $f(e_1, ..., e_n)$ with I and C. Suppose u^r is the current cache parameter. Let $u_1, ..., u_m$ be all the variables other than u^r in $e_1, ..., e_n$. Let $u_1^x, ..., u_p^x$ be those u_i's that depend only on x, and $u_1^y, ..., u_q^y$ be the rest of u_i's. Let $\langle e_c, e_r, I_1\rangle$ be any element in C such that $I \Rightarrow I_1$ and all the variables in e_c are in $\{u_1^x, ..., u_p^x\}$, and thus the element is valid and relevant.

Let E be the set of expressions e such that e is the largest common $u^r \backslash \{u_1, ..., u_m\}$-cover or $u_j^x \backslash \{u^r, u_1^y, ..., u_q^y\}$- or $u_k^y \backslash \{u^r, u_1^x, ..., u_p^x\}$-cover expression of $e_1, ..., e_n, e_c$, and e_r for some u_j^x or u_k^y. Let

$$\theta = \{e/v \mid e \in E\} \tag{18}$$

where v's are distinct new variable names,[2] then θ is a substitution corresponding to these largest common cover expressions of $e_1, ..., e_n, e_c$, and e_r. Using the inverse substitution $\theta^{-1} = \{v/e \mid e/v \in \theta\}$, we obtain $e_1', ..., e_n', e_c'$, and e_r' such that $e_i' = e_i\theta^{-1}$ for $i = 1, ..., n, c, r$, and we obtain an information set I' such that I' is $I\theta^{-1}$ extended with equations induced by θ that are relevant to $I\theta^{-1}$, i.e.,

$$I' = I\theta^{-1} \cup \{e \leftrightarrow v \mid e/v \in \theta, \text{ a variable in } e \text{ occurs in } I\theta^{-1}\}.$$

We say that $\langle e_1', ..., e_n', e_c', e_r', I'\rangle$ is a *generalization* for $\langle f(e_1, ..., e_n), I, C\rangle$ with *substitution* θ. For any e/v in θ, we say variable v *generalizes* expression e. It is clear that such a generalization obeys the first consideration.

A tuple $\langle f(e_1, ..., e_n), I, C\rangle$ may have more than one generalization if there are more than one element in C. Suppose A_1 and A_2 are two generalizations with substitutions θ_1 and θ_2, respectively. We say A_1 is *more general* than A_2 if any expression in $\{e \mid e/v \in \theta_2\}$ is a subexpression of some expression in $\{e \mid e/v \in \theta_1\}$. We say A_1 is *most general* if no other generalization is more general. This incorporates the second consideration.

[2] If an expression e in the set E is a variable u, then the corresponding variable v can be u, i.e., v does not have to be a new variable in this case.

5.3 Function Introduction and Replacement

Given a function application $f(e_1,...,e_n)$, let I be the information set at $f(e_1,...,e_n)$, C the current cache set for the unfolded application that contains $f(e_1,...,e_n)$, and D the current definition set. If we can use a previously introduced function f' in D to compute $f(e_1,...,e_n)$ incrementally, then $f(e_1,...,e_n)$ is replaced by an application of f'. Otherwise, we introduce a new function f' into D to compute $f(e_1,...,e_n)$ incrementally and, if f' computes fast, replace $f(e_1,...,e_n)$ by an application of this f', otherwise, leave $f(e_1,...,e_n)$ unchanged; as a result, the definition set is changed as a side effect. This process is achieved by $\mathcal{I}nc\mathcal{A}pply$, first introduced in Section 3. It is defined in Figure 6 and explained below.

$\mathcal{I}nc\mathcal{A}pply[\![f(e_1,...,e_n)]\!]\,I\,C\,D$

$= \langle f'(v_{i_1}\theta,...,v_{i_j}\theta),\ D\rangle$ if $\exists\,\langle f(e_1',...,e_n'), f'(v_{i_1},...,v_{i_j}), \{\langle e_c', e_r', I'\rangle\}\rangle \in D$ with a substitution θ s.t.
$f(e_1,....,e_n) \leftrightarrow_I^* f(e_1'\theta,...,e_n'\theta),\ \ I \Rightarrow I'\theta,\ \ e_c'\theta \rightarrow_{IC}^* e_r'\theta$, and
if f' is fully defined, $t(f'(v_{i_1}\theta,...,v_{i_j}\theta)) \le t(f(e_1,...,e_n))$;

$= \langle f'(v_{i_1}\theta,...,v_{i_j}\theta), D'''\rangle$ else if $f(e_1,...,e_n)$ depends on x but can not be replaced by a retrieval, and,
after obtaining a definition of f' as follows,
if f' is fully defined, $t(f'(v_{i_1}\theta,...,v_{i_j}\theta)) \le t(f(e_1,...,e_n))$;

introduce $d = \langle f(e_1',...,e_n'), f'(v_1,...,v_k), C'\rangle$ with θ, where $C' = \{\langle e_c', e_r', I'\rangle\}$,
and obtain $f'(v_{i_1},...,v_{i_j})$, to be defined as some e''', by the following steps:
1) $e' = e[e_1'/v_1,...,e_n'/v_n]$, where f is defined by $f(v_1,...,v_n) = e$
2) $\langle e'', C'', D''\rangle = \mathcal{I}nc[\![e']\!]\,I'\,C'\,D'$, where $D' = D \cup \{d\}$
3) $\langle f'(v_{i_1},...,v_{i_j}), D'''\rangle = \mathcal{E}lim[\![f'(v_1,...,v_k)]\!]\,D''$, where $f'(v_1,..,v_k)$ is defined as e''

$= \langle f(e_1,...,e_n),\ D''''\rangle$ otherwise, where D'''' is D''' as above if it is computed and D otherwise

Figure 6: Definition of $\mathcal{I}nc\mathcal{A}pply$

Function Replacement. Since an introduced function f' is associated with a cache element as an invariance, to use f', we need to justify the corresponding invariance. We say that a function application $f(e_1,...,e_n)$ with I, C, and D can be replaced by an application of a previously introduced function f' if there is $\langle f(e_1',...,e_n'), f'(v_{i_1},...,v_{i_j}), \{\langle e_c', e_r', I'\rangle\}\rangle$ in D and there is a substitution θ over the variables in $e_1',...,e_n', e_c'$, e_r', and I' such that

1) $f(e_1,...,e_n)$ equals $f(e_1'\theta,...,e_n'\theta)$, the invariance $I'\theta$ holds, $e_c'\theta$ can be replaced by $e_r'\theta$, and
2) if f' is fully defined, $f'(v_{i_1}\theta,...,v_{i_j}\theta)$ is asymptotically at least as fast as $f(e_1,...,e_n)$.

In this case, $f(e_1,...,e_n)$ can be replaced by $f'(v_{i_1}\theta,...,v_{i_j}\theta)$, and the definition set D remains unchanged.

For example, given the definition set (14), the application

$$out(cdr(C), insert(i,a,R)) \quad \text{with} \quad \langle out(cdr(C),R),\ cdr(r),\ \{null(C) \leftrightarrow F\}\rangle \in C_{out} \tag{19}$$

in the false branch of (6) can be replaced by $out'(cdr(C), i, a, R, cdr(r))$.

Function Introduction. If $f(e_1,...,e_n)$ with I, C, and D can not be replaced by an application of a previously introduced function in D, then we introduce a new function f' into D to compute $f(e_1,...,e_n)$ incrementally. Following the basic derivation idea, we introduce f' only if $f(e_1,...,e_n)$ depends on x but can not be replaced by a retrieval from a cached result.

Given $f(e_1,...,e_n)$ with I and C, let $\langle e_1',...,e_n', e_c', e_r', I'\rangle$ be a most general generalization with substitution θ. Let $v_1,...,v_k$ be all the variables in $e_1',...,e_n', e_c'$, and e_r'. We introduce $\langle f(e_1',...,e_n'), f'(v_1,...,v_k), C'\rangle$, where $C' = \{\langle e_c', e_r', I'\rangle\}$, into D to get D', and we obtain a definition of f' by the following three steps:

1) unfold the application $f(e'_1, ..., e'_n)$ to get e';
2) incrementalize e' with information set I', cache set C', and definition set D' to get e'';
3) eliminate dead parameters of f', defined by $f'(v_1, ..., v_k) = e''$, in computing $f'(v_1, ..., v_k)$.

Note that the second step uses the function $\mathcal{I}nc$, which may use $\mathcal{I}nc\mathcal{A}pply$ recursively for function applications. After the third step, if we obtain $f'(v_{i_1}, ..., v_{i_j})$ and, if f' is fully defined, $t(f'(v_{i_1}\theta, ..., v_{i_j}\theta)) \leq t(f(e_1, ..., e_n))$, then we replace $f(e_1, ..., e_n)$ by $f'(v_{i_1}\theta, ..., v_{i_j}\theta)$. The set D is changed as a side effect.

Dead Parameter Elimination. After the second step above, $f'(v_1, ..., v_k)$ computes $f(e'_1, ..., e'_n)$ and is defined as e''. Since e'' is obtained by replacing some subcomputations of $f(e'_1, ..., e'_n)$ depending on x by computations depending on the current cache parameter, those parameters of f' on which the replaced computations depend may become dead.

Dead code elimination is a traditional optimization [1, 25]. We assume a subroutine $\mathcal{E}lim$ is given so that $\mathcal{E}lim[\![f'(v_1, ..., v_k)]\!]D''$, where $f'(v_1, .., v_k)$ is defined as e'' in D'', returns the pair $\langle f'(v_{i_1}, ..., v_{i_j}), D'''\rangle$, where $1 \leq i_1 < ... < i_j \leq k$ and $f'(v_{i_1}, ..., v_{i_j})$ is defined as some e''' in D''' after dead parameter elimination, and $f'(v_{i_1}, ..., v_{i_j})$ returns a value if and only if $f'(v_1, ..., v_k)$ returns the same value.

Example. Consider our running example. For the application $row(car(C), insert(i, a, R))$ in the false branch of (6), we introduce a new function row' as in (17). To obtain a definition of row', we first unfold $row(c, insert(i, a, R))$ to get

$$\begin{array}{l}\textbf{if } null(insert(i, a, R)) \textbf{ then } nil \\ \textbf{else } cons(c * car(insert(i, a, R)),\ row(c, cdr(insert(i, a, R))))\end{array} \tag{20}$$

Then, we incrementalize (20) using $row(c, R) = r_1$ as given by the cache set in (17). The incrementalization is sketched as follows. It is easy to see that $insert(i, a, R)$ in the condition can be unfolded and the condition simplified to true, and thus (20) is reduced to

$$cons(c * car(insert(i, a, R)),\ row(c, cdr(insert(i, a, R)))) \tag{21}$$

The first occurrence of $insert(i, a, R)$ in (21) can be unfolded, conditions in the unfolded application lifted, and car of $cons$ applications simplified. Thus, (21) becomes

$$\begin{array}{l}\textbf{if } i \leq 1 \textbf{ then } cons(c * a,\ row(c, cdr(insert(i, a, R)))) \\ \textbf{else if } null(R) \textbf{ then } cons(c * a,\ row(c, cdr(insert(i, a, R)))) \\ \textbf{else } cons(c * car(R),\ row(c, cdr(insert(i, a, R))))\end{array} \tag{22}$$

The three occurrences of $insert(i, a, R)$ in (22) can be specialized under their corresponding contexts, unfolded, and then cdr of $cons$ applications simplified. Thus, (22) becomes

$$\begin{array}{l}\textbf{if } i \leq 1 \textbf{ then } cons(c * a,\ row(c, R)) \\ \textbf{else if } null(R) \textbf{ then } cons(c * a,\ row(c, nil)) \\ \textbf{else } cons(c * car(R),\ row(c, insert(i-1, a, cdr(R))))\end{array} \tag{23}$$

In the first branch of (23), $row(c, R)$ can be directly replaced by r_1. In the second branch, $row(c, nil)$ can be specialized and unfolded to nil. For the third branch, we have $null(R) \leftrightarrow F$; thus $row(c, R)$ is specialized to $cons(c * car(R),\ row(c, cdr(R)))$ and the cache set is extended so that

$$c * car(R) = car(r_1) \quad \text{and} \quad row(c, cdr(R)) = cdr(r_1)$$

Thus, $c*car(R)$ can be replaced by $car(r_1)$, and the application $row(c, insert(i–1, a, cdr(R)))$ can be replaced by $row'(c, i-1, a, cdr(R), cdr(r_1))$. Additionally, in a situation similar to (9), $null(R)$ can be replaced by

$null(r_1)$. Thus, (23) is reduced to

$$\begin{array}{l}\textbf{if } i \leq 1 \textbf{ then } cons(c*a,\ r_1)\\ \textbf{else if } null(r_1) \textbf{ then } cons(c*a,\ nil)\\ \textbf{else } cons(car(r_1),\ row'(c, i-1, a, cdr(R), cdr(r_1)))\end{array} \tag{24}$$

Finally, for $row'(c, i, a, R, r_1)$ defined as (24), it is clear that the parameter R is dead and can be eliminated. We obtain the final definition of row' as given in Figure 2. The application $row(car(C), insert(i, a, R))$ can be replaced by $row'(car(C), i, a, car(r))$, since the latter is asymptotically at least as fast as the former.

To complete our example, for the initial application $out(C, insert(i, a, R))$, we introduce a new function out' as in (14). In incrementalizing the unfolded application of out as in (6), the Boolean expression $null(C)$ can be replaced by $null(r)$ due to (9), the application of row can be replaced by $row'(car(C), i, a, car(r))$ as just given above, and the recursive application of out can be replaced by $out'(cdr(C), i, a, R, cdr(r))$ as followed from (19). Therefore, the unfolded application (6) is reduced to

$$\begin{array}{l}\textbf{if } null(r) \textbf{ then } nil\\ \textbf{else } cons(row'(car(C), i, a, car(r)),\ out'(cdr(C), i, a, R, cdr(r)))\end{array} \tag{25}$$

For $out'(C, i, a, R, r)$ defined as (25), it is clear that the parameter R is dead and can be eliminated. We obtain the final definition of out' as given in Figure 2. Finally, the application $out(C, insert(i, a, R))$ can be replaced by $out'(C, i, a, r)$, i.e., given $out(C, R) = r$, $out'(C, i, a, r)$ computes $out(C, insert(i, a, R))$ and is at least as fast.

6 Summarizing the Derivation Procedure

The derivation procedure can be summarized as follows. The function $\mathcal{I}nc\mathcal{A}pply$ maintains the global set D, introduces new functions to compute function applications incrementally, and replaces these applications by appropriate applications of introduced functions. $\mathcal{I}nc\mathcal{A}pply$ calls the function $\mathcal{I}nc$, which maintains a cache set C, discovers subcomputations whose values can be retrieved from cached results, and incrementalizes the computation of an unfolded function by simplification and replacement using retrievals. $\mathcal{I}nc$ recursively calls $\mathcal{I}nc\mathcal{A}pply$ if a subcomputation is a function application. The derivation procedure starts with

$$\mathcal{I}nc\mathcal{A}pply[\![f_0(x \oplus y)]\!]\ \emptyset\ \{\langle f_0(x), r, \emptyset\rangle\}\ \emptyset \tag{26}$$

and, if it terminates, returns $\langle f_0'(x, y, r),\ D\rangle$, where D is the set of functions introduced during the derivation. We can eliminate dead functions in F, the set of functions in the original program, and D that are not reachable from f_0' in the call-graph.

The derivation procedure preserves the semantics of programs and achieves at least as fast computations, i.e., if $f_0(x) = r$, then (a) whenever $f_0(x \oplus y)$ returns a value, $f_0'(x, y, r)$ returns the same value; and (b) $f_0'(x, y, r)$ is asymptotically at least as fast as $f_0(x \oplus y)$. To see this, notice that semantics are preserved and fast computations are achieved by all of the transformations in the derivation procedure — simplification by $\mathcal{S}imp$, computation of cache sets and replacement by $\mathcal{R}epl$, lifting of conditions and bindings by $\mathcal{S}ubl$, and function replacement and introduction with generalization by $\mathcal{I}nc\mathcal{A}pply$. Note that unfolding may result in computations that terminate more often than the original computations.

6.1 Transformation and Analysis Techniques Used

The derivation procedure combines a number of program analysis and transformation techniques to achieve the ambitious goal of deriving incremental programs. It is a deterministic transformational procedure.

The transformation starts with $f_0(x \oplus y)$, so that f_0' is computable, and aims to improve the efficiency by replacing subcomputations whose values can be retrieved from cached result r of $f_0(x)$ by corresponding retrievals. This starting point is similar to that of partial evaluation, which starts with a trivial specialized program given by Kleene's *s-m-n* theorem and attempts improvements by symbolic reductions or similar techniques.

Transformation Techniques. We summarize the major transformation techniques used and emphasize how they are combined to achieve our goal.

First, context information is collected for each subcomputation and used to simplify the computation, which mimics the main techniques of generalized partial evaluation [14], where program states are represented symbolically and programs are specialized with the help of a theorem prover. In addition to simplification, context information has another important role in our work, i.e., it serves as keys to cached results and introduced functions for valid replacement to happen.

Second, a cache set is maintained for each unfolded application and used to incrementalize it, i.e., to replace certain subcomputations, under certain context information, by retrievals from a cache result of a previous computation. A cache set is augmented, finitely and in a disciplined way, with the help of an auxiliary specializer so that the cached result is utilized effectively under valid context information. The use of a cached result often suggests memoization [23, 4]. However, the real power of our approach comes from the effective exploitation of a memoized value under valid context information. The approach to be proposed in Section 6.3 for increasing incrementality by caching auxiliary information can be regarded as a form of smart memoization.

Third, in consistence with the strict semantics of our language, we apply simplification and replacement on subcomputations in applicative order, and, moreover, we lift conditions and bindings out of subcomputations. This lifting technique is similar in spirit to the driving transformation by supercompilation [44]. It causes relatively drastic reorganization of program structures that helps expose incrementality that is otherwise hidden.

Fourth, a global definition set is maintained and used to replace function applications, with corresponding relevant cache elements and valid context information, by applications of introduced functions. Function introduction with generalization and function replacement use the unfold/define/fold scheme [6] in a regulated manner so that the transformations are deterministic and the derived programs do not lose termination. Moreover, relevant cache elements with valid context information are chosen to be passed into introduced functions, so that they can be effectively used to incrementalize the computation of corresponding function applications.

Last, after the replacements described above, we apply dead code elimination, a traditional optimization technique [1, 25]. It is particularly useful here, since replacement changes dependencies between computations, and computations on which no other computations depend are then dead and can be eliminated.

Analysis Techniques. To implement the above transformations, several program analysis techniques are needed and summarized here.

First, time analysis [47, 37] is used when replacing subcomputations by retrievals or replacing function applications by applications of introduced functions.[3] It is a must if we want to guarantee the efficiency of the derived programs.

Then, a number of analysis [19] are used to assist transforming function applications. *Dependence analysis* enables us to recognize subcomputations that are possibly computed incrementally, i.e., subcomputations depending on x, and thus avoid introducing functions for function applications that depend only on y, which then helps the derivation procedure terminate. *Call-graph analysis* tells us whether a function is recursively defined and also whether an introduced function is fully defined. *Occurrence counting analysis* helps us decide whether an unfolding duplicates computations.

Finally, dead code analysis recognizes dead code to be eliminated. In particular, dead parameters of functions can be recognized with the help of dependence analysis, and dead functions can be identified with

[3] The minor use of time analysis in assisting some simplifications can be easily avoided.

the help of call-graph analysis.

Additionally, other analysis techniques, although not mentioned in our transformations so far, would also benefit the derivation procedure. For example, type analysis would be helpful for simplifying overloaded functions. Also, static analysis could provide annotations that guide the derivation and help it terminate, mimicking binding time analysis in partial evaluation, as discussed below.

Last but not least, we should note that the quality of a derived incremental program depends on the corresponding non-incremental program. We should not expect "genuine creativity" without discoveries and proofs of some "substantial" theorems. On another hand, with the power of our combined techniques, a very simple theorem prover can already help us derive efficient incremental programs. Illustrative examples can be found in Section 7.

6.2 Improving the Derivation Procedure

A number of optimizations can be made to the derivation procedure. An implementer would naturally realize most of them. As an example, assume our replacement guarantees

$$\text{if } \mathcal{R}epl[\![e]\!]IC = \langle e', C'\rangle, \text{ then } \mathcal{R}epl[\![e']\!]IC' = \langle e', C'\rangle,$$

then we can make the optimization:

$$\mathcal{I}nc[\![\textbf{if}\, e_1 \,\textbf{then}\, e_2 \,\textbf{else}\, e_3]\!]ICD = \mathcal{I}nc[\![e_3]\!]IC'D', \quad \text{if } \mathcal{I}nc[\![e_1]\!]ICD = \langle F, C', D'\rangle. \tag{27}$$

A relatively important improvement is with the function introduction for a function application $f(e_1, ..., e_n)$, as in the second case in Figure 6. While we incrementalize the unfolded application, its cache set is extended from C' to C'', but C'' is discarded after this, even if $C''\theta$ might be used in incrementalizing the rest of the unfolded application that contains $f(e_1, ..., e_n)$. To make use of C'' for this purpose, we can let $\mathcal{I}nc\mathcal{A}pply$ also return the set $C''\theta$ and merge it with the cache set of the unfolded application that contains $f(e_1, ..., e_n)$.

Termination. The derivation procedure follows function applications and introduces new functions to compute these applications incrementally. Therefore, if functions are recursively defined, the derivation procedure may not terminate due to introducing infinitely many functions following infinite unfolding. Non-termination is a traditional problem in a transformational approach, and it is well-know that there is a trade-off between termination of the transformation and efficiency of the transformed programs.

In our derivation, we only introduce new functions for function applications that depend on x, which may affect the efficiency of other function applications, but makes the derivation terminate more often without impeding the discovery of incrementality. It is also clear that function replacement and the notion of generalization for function introduction help the derivation terminate in a natural way. However, our heuristic of one cache element per introduced function might impede achieving incrementality, since this element may not be *sufficient*, i.e., it may not enable all of the simplifications and replacements that are possible when using more cache elements. We could overcome this by using as many cache elements as possible when introducing a function and eliminating useless ones later. But this may cause a too complicated treatment of recursive functions and may make the derivation terminate less often. On the other hand, this is a place where separate passes of static analysis could help, imitating binding time analysis in partial evaluation. This suggests a direction for future work.

Although in general, any attempt to limit function introductions could affect achieving incrementality for certain programs, it does not hurt to try a few good heuristics with more reasonable termination behavior. For example, we may introduce a new function at a function application only if we can effectively decide that, in incrementalizing the unfolded application, some subcomputations can be simplified. Thus assuming we have a complete equality reasoning mechanism and a sufficient cache element when introducing a function, if the derivation procedure does not terminate, there must be simplification possible along an infinite path, and thus there must an execution of the original program that does not terminate. In other words, if the original program terminates on all inputs, then the derivation procedure terminates, and the derived program terminates on all inputs at least as fast with the right values. Note, however, that the complexity of the derivation procedure may not be bounded by the size of a given program, since it may loop on ground values. The rationale is that computations done at transformation time need not be done in the transformed programs.

Other Concerns. Two other weaknesses result from unfolding as done by the derivation procedure. First, only partial correctness is preserved, i.e., a derived program may terminate more often than the original program. Second, subcomputations may be duplicated in a derived program.

Both drawbacks can be overcome by inserting **let** bindings to compute the arguments when unfolding function applications, i.e., instead of unfolding a function application to $e_f[e_1/v_1, ..., e_n/v_n]$, we unfold it to

$$\mathbf{let}\ v_1{=}e_1\ \mathbf{in}\ ...\ \mathbf{let}\ v_n{=}e_n\ \mathbf{in}\ e_f\ \mathbf{end}\ ...\ \mathbf{end}$$

Then we modify the condition of unfolding **let** expressions in $\mathcal{S}imp$, namely, **let** $v = e_1$ **in** e_2 **end** can be unfolded only if $e_2[e_1/v]$ neither duplicates non-trivial computations nor discards non-terminate computations, where the latter means either e_1 can be effectively decided to terminate or v occurs at least once on every (syntactic) execution path in e_2. As occurrence counting analysis helps decide whether an unfolding duplicates computations, it can also help decide whether an unfolding discards computations.

Similar solutions are proposed in partial evaluation [24, 5]. Note that, even without this technique, the efficiency of our derived programs are guaranteed with the help of time analysis. But in partial evaluation where no time analysis is employed, a transformed program could take exponential time while the original program takes only polynomial time [24]. As a matter of fact, even with this technique, time analysis is still needed in our derivation, since we replace subcomputations by retrievals from a cache result only when we can save time by doing so. This is inherent in incremental computation and is a complication over partial evaluation.

6.3 Increasing Incrementality by Caching Auxiliary Information

In the derivation approach presented above, the derived function f' only uses the cached result r of $f(x)$ to compute $f(x \oplus y)$ incrementally.[4] Adding auxiliary information about x for f' to use could lead to greater incrementality, i.e., f' might be able to compute $f(x \oplus y)$ even faster by making use of the auxiliary information. We must augment the values returned by f to include this auxiliary information.

Let $\hat{f}$ denote the function obtained by extending f to return the augmented values, and assume that when $f(x)$ is needed, it is projected out of $\hat{f}(x)$. Suppose $\hat{f}(x)$ returns the augmented value $\hat{r}$. Then using our derivation procedure presented above, we can obtain an incremental version $\hat{f}'$ of $\hat{f}$ such that $\hat{f}'(x, y, \hat{r}) = \hat{f}(x \oplus y)$. Note that the domain of $\hat{f}'$ is augmented from that of f' to include the auxiliary information as input. At the same time, the range of $\hat{f}'$ is also augmented from that of f' to include the corresponding auxiliary information about $x \oplus y$. Therefore, we are prepared for incremental computation after further input changes, which is a natural requirement for normal applications.

We can regard the extension of f to $\hat{f}$ as a separate step before the derivation procedure. In principle, given a function f and an input change operation $\oplus$, there is no general way of obtaining $\hat{f}$ to enable greater incrementality in computing the value of $f(x \oplus y)$. We propose to approach the problem in two stages.

First, there may be subcomputations performed in $f(x)$ that are not embedded in its return value r but are crucial for achieving greater incrementality in computing $f(x \oplus y)$. We can expand $f(x)$ and $f(x \oplus y)$ to identify such computations in $f(x)$ and then extend $f(x)$ to embed the results of these computations in the final return value.

[4] From this section on, for notational convenience, we use f to denote f_0.

Second, there may be information about x that is not computed by the original function $f(x)$ at all but is crucial for obtaining greater incrementality in computing $f(x \oplus y)$. We expect to discover such information in the computations in $f(x \oplus y)$ that only depend on x but are not in $f(x)$. We can expand $f(x \oplus y)$ and $f(x)$ to identify such computations in $f(x \oplus y)$ and then extend $f(x)$ to compute them efficiently as well.

To illustrate the two points above, we present a simple example. Let $x = \langle x_1, x_2 \rangle$, where x_1 and x_2 are two lists, and function $f(x_1, x_2)$ return the product of the lengths of the two lists:

$$f(x_1, x_2) = \quad len(x_1) * len(x_2), \qquad len(x) = \quad \textbf{if } null(x) \textbf{ then } 0 \textbf{ else } 1 + len(cdr(x))$$

Let $y = \langle y_1, y_2 \rangle$ and let the new input to f, $x \oplus y$, be $\langle cons(y_1, x_1), cons(y_2, x_2) \rangle$. Suppose r is the cached result of $f(x_1, x_2)$ and we use it in computing $f(x \oplus y)$ incrementally. Following the derivation procedure, we introduce $f'(y_1, x_1, y_2, x_2, r)$ to compute $f(cons(y_1, x_1), cons(y_2, x_2))$ incrementally. After unfolding the application of f and reducing subexpressions, we get $(len(x_1)+1) * (len(x_2)+1)$. Then, using properties of the primitive function '$*$', we get $len(x_1) * len(x_2) + len(x_1) + len(x_2) + 1$, where $len(x_1) * len(x_2)$ can be replaced by r. Thus, we obtain an incremental version $f'(x, r)$ that computes $f(x \oplus y)$:

$$f'(x_1, x_2, r) = \quad r + len(x_1) + len(x_2) + 1$$

Although f' saves computing a '$*$' operation, it is of dubious value if we must recompute $len(x_1)$ and $len(x_2)$.

Using the idea of the first point above, we see that $len(x_1)$ and $len(x_2)$ are subcomputations performed in $f(x_1, x_2)$ whose values could be used to compute $f(x \oplus y)$ even faster than $f'(x, r)$ but can not be retrieved from the cached result r. Thus, we extend f to $\hat{f}_1$ such that $\hat{f}_1$ embeds the auxiliary information $len(x_1)$ and $len(x_2)$ in the return value $\hat{r}_1$:

$$\hat{f}_1(x_1, x_2) = \quad \textbf{let } l_1 = len(x_1) \textbf{ in let } l_2 = len(x_2) \textbf{ in } triple(l_1 * l_2,\ l_1,\ l_2) \textbf{ end end}$$

where *triple* is a constructor with corresponding selectors $1st$, $2nd$, and $3rd$. It is easy to see that using our derivation procedure, we can obtain an incremental version $\hat{f}'_1(\hat{r}_1)$ that computes $\hat{f}_1(x \oplus y)$:

$$\hat{f}'_1(\hat{r}_1) = \quad triple(1st(\hat{r}_1) + 2nd(\hat{r}_1) + 3rd(\hat{r}_1) + 1,\ 2nd(\hat{r}_1) + 1,\ 3rd(\hat{r}_1) + 1)$$

Compared to $f'(x_1, x_2, r)$, $\hat{f}'_1(\hat{r}_1)$ saves computing $len(x_1)$ and $len(x_2)$. Note that a cache of size three is required, whereas f' requires only a cache of size one.

Using the idea of the second point above, we see that the computation $len(x_1) + len(x_2)$ depends completely on x and its value would enable even faster computation of $f(x \oplus y)$ than $\hat{f}'_1(\hat{r}_1)$ but the '$+$' operation is not contained in the computation $f(x)$ at all. Moreover, caching this value would obviate the need to to cache the values of $len(x_1)$ and $len(x_2)$ separately. Thus, we extend f to $\hat{f}_2$ such that $\hat{f}_2$ returns $\hat{r}_2$ containing both the product and the sum of the two lengths:

$$\hat{f}_2(x_1, x_2) = \quad \textbf{let } l_1 = len(x_1) \textbf{ in let } l_2 = len(x_2) \textbf{ in } pair(l_1 * l_2,\ l_1 + l_2) \textbf{ end end}$$

where *pair* is a constructor with corresponding selectors fst and snd. It is easy to see that using our derivation procedure, we can obtain an incremental version $\hat{f}'_2(\hat{r}_2)$ that computes $\hat{f}_2(x \oplus y)$:

$$\hat{f}'_2(\hat{r}_2) = \quad pair(fst(\hat{r}_2) + snd(\hat{r}_2) + 1,\ snd(\hat{r}_2) + 2)$$

Compared to $\hat{f}'_1(\hat{r}_1)$, $\hat{f}'_2(\hat{r}_2)$ does slightly less arithmetic and uses a smaller cache.

These two techniques above are currently being studied. The degree to which it is possible to generate the auxiliary information automatically is an open question.

Corresponding to finding auxiliary information, many dynamic algorithms use specially designed data structures to answer queries quickly. Although there is no universally applicable data structure, some apply to a broad class of problems [11]. Accommodating such general data structures in our model for deriving incremental programs might help in deciding whether a data structure is applicable to a certain problem. How this might be done is another question open for study.

6.4 Mechanization

With the oracle of a theorem prover, time analysis techniques, and heuristics for function introductions, the derivation can be fully automated. In practice, the derivation can be made semi-automatic when some of these oracles are only semi-automatically provided.

Although we see the derivation as certainly no more automatable than partial evaluation, it is desirable to at least use the computer as a sophisticated editor, suggesting and carrying out detailed transformations. It is also nice that the derived programs are in the same language as the original programs, and therefore they are executable and one can check solutions and try out alternatives.

We have implemented a prototype system called CACHET for deriving incremental programs based on our approach. The implementation uses the Synthesizer Generator [35], a system for generating language-based editors, and consists of about 14,000 lines of code written in SSL, the Synthesizer Generator language for specifying editors. Source-to-source transformations are operations built in to our editor.

Currently, the transformation rules are invoked manually mainly for two reasons. First, the Synthesizer Generator does not currently have a rewrite engine for us to do an applicative order reduction easily as is required by the derivation procedure. Second, we want an interactive environment to study the applicability of various transformations, thus manual invocation is suitable most of the time. Also, at present, we are only using a very simple equality reasoning engine, not a full-blown theorem prover.

CACHET has been used to derive numerous incremental programs. It is also helpful in studying transformations for caching auxiliary information. We plan to add rewrite mechanism into the Synthesizer Generator to further automate derivations by CACHET. We also plan to interface CACHET to a substantial theorem prover.

7 Examples

To see the power and some interesting behavior of the derivation procedure, we consider incrementalizing several different sorting programs. Let *sort* be a function that takes a list of numbers x and returns the sorted list $sort(x)$. Let the change to the input of *sort* be that an extra number is added at the beginning of the list, i.e., $x' = cons(y, x)$.

7.1 Insertion Sort

Suppose the program is an insertion sort that inserts the first element of the list into recursively sorted list of the rest.

$$\begin{array}{rl} sort(x) = & \textbf{if } null(x) \textbf{ then } nil \\ & \textbf{else } insert(car(x), sort(cdr(x))) \\ \\ insert(i, x) = & \textbf{if } null(x) \textbf{ then } cons(i, nil) \\ & \textbf{else if } i \leq car(x) \textbf{ then } cons(i, x) \\ & \qquad \textbf{else } cons(car(x), insert(i, cdr(x))) \end{array} \tag{28}$$

To compute $sort(cons(y, x))$ incrementally using $sort(x) = r$, all we need to do is a function introduction, followed by an unfolding, a few simplifications, a replacement, and a dead parameter elimination, and finally a use of the introduced function, as sketched below:

$$\begin{array}{lll} sort'(y_1, x_1, r_1) & = sort(cons(y_1, x_1)), \quad \text{with } sort(x_1) = r_1 & \text{function introduction} \\ & = \textbf{if } null(cons(y_1, x_1)) \textbf{ then } nil & \text{unfolding} \\ & \quad \textbf{else } insert(car(cons(y_1, x_1)), sort(cdr(cons(y_1, x_1)))) & \\ & = insert(y_1, sort(x_1)) & \text{simplifications} \\ & = insert(y_1, r_1) & \text{replacement} \\ sort'(y_1, r_1) & = insert(y_1, r_1) & \text{dead parameter elimination} \\ sort(cons(y, x)) & = sort'(y, r), \quad \text{for } sort(x) = r & \text{use of the introduced function} \end{array}$$

The derived incremental program simply uses *insert* to insert the newly added number into the previously sorted list.

A more formal derivation following the derivation procedure is given below. We start with

$$\mathcal{I}nc\mathcal{A}pply[\![sort(cons(y,x))]\!]\ \emptyset\ \{\langle sort(x), r, \emptyset\rangle\}\ \emptyset \tag{29}$$

where we introduce $sort'$ as in the tuple d_1:

$$\langle sort(cons(y_1,x_1)), sort'(y_1,x_1,r_1), C_1\rangle, \quad \text{where } C_1 = \{\langle sort(x_1), r_1, \emptyset\rangle\}$$

and we obtain a definition of $sort'$ as follows:

1. We unfold $sort(cons(y_1,x_1))$ and get an expression e_1:

$$\begin{array}{l}\textbf{if } null(cons(y_1,x_1)) \textbf{ then } nil \\ \textbf{else } insert(car(cons(y_1,x_1)), sort(cdr(cons(y_1,x_1))))\end{array}$$

2. We incrementalize e_1:

$$\mathcal{I}nc[\![e_1]\!]\ \emptyset\ C_1 D_1, \quad \text{where } D_1 = \{d_1\}$$

   which first calls $\mathcal{S}ubl$ to reduce subexpressions:

$$\begin{array}{l}\mathcal{S}ubl[\![e_1]\!]\ \emptyset\ C_1 D_1 \\ < \text{by } (if_1),\ \mathcal{I}nc[\![null(cons(y_1,x_1))]\!]\ \emptyset\ C_1 D_1 = \langle F, C_1, D_1\rangle > \\ = \mathcal{S}ubl[\![\textbf{if } F \textbf{ then } nil \textbf{ else } e_{13}]\!]\ \emptyset\ C_1 D_1, \quad \text{where } e_{13} \text{ denotes the false branch of } e_1 \\ < \text{by } (if_n),\ \mathcal{I}nc[\![e_{13}]\!]\ \{F \leftrightarrow F\}\ C_1 D_1 = \langle insert(y_1,r_1), C_1, D_1\rangle \\ \qquad\qquad \text{since } car(cons(y_1,x_1)) = y_1,\ cdr(cons(y_1,x_1)) = x_1, \text{ and } sort(x_1) = r_1, \\ \qquad\qquad \mathcal{I}nc\mathcal{A}pply \text{ does not transform } insert(y_1,r_1) \text{ since it does not depend on } x > \\ = \langle \textbf{if } F \textbf{ then } nil \textbf{ else } insert(y_1,r_1), C_1, D_1\rangle\end{array}$$

   and then applies $\mathcal{S}imp$ and $\mathcal{R}epl$ to the resulting expression:

$$\begin{array}{l}\mathcal{R}epl[\![\mathcal{S}imp[\![\textbf{if } F \textbf{ then } nil \textbf{ else } insert(y_1,r_1)]\!]\ \emptyset^{C_1}]\!]\ \emptyset^{C_1}\ C_1 \\ = \mathcal{R}epl[\![insert(y_1,r_1)]\!]\ \emptyset^{C_1}\ C_1 \\ = insert(y_1,r_1)\end{array}$$

3. We eliminate dead parameters of $sort'$, defined by $sort'(y_1,x_1,r_1) = insert(y_1,r_1)$. Clearly, x_1 is dead. We obtain a final definition of $sort'$:

$$sort'(y_1,r_1) = insert(y_1,r_1) \tag{30}$$

It is clear that $sort'(y_1,r_1)$ computes asymptotically at least as fast as $sort(cons(y_1,x_1))$ because each transformation step above guarantees this relation. Therefore, $t(sort'(y,r)) \le t(sort(cons(y,x)))$. Thus, (29) returns

$$\langle sort'(y,r),\ \ \{\langle sort(cons(y_1,x_1)),\ sort'(y_1,r_1),\ \{\langle sort(x_1), r_1, \emptyset\rangle\}\rangle\}\rangle$$

where $sort'$ is defined as in (30) and *insert* is defined as in the original program (28).

7.2 Selection Sort

Suppose the program is a selection sort that selects the least number in the list as the first number in the sorted list and sorts the rest recursively.

$$
\begin{array}{ll}
sort(x) = & \textbf{if } null(x) \textbf{ then } nil \\
& \textbf{else let } k = least(x) \\
& \quad\quad \textbf{in } cons(k, sort(rest(x,k))) \textbf{ end} \\
\\
least(x) = & \textbf{if } null(cdr(x)) \textbf{ then } car(x) \\
& \textbf{else let } s = least(cdr(x)) \\
& \quad\quad \textbf{in if } car(x) \le s \textbf{ then } car(x) \textbf{ else } s \textbf{ end} \\
\\
rest(x,k) = & \textbf{if } k = car(x) \textbf{ then } cdr(x) \\
& \textbf{else } cons(car(x), rest(cdr(x), k))
\end{array}
\tag{31}
$$

Again, we start by introducing $sort'(y_1, x_1, r_1)$ to compute $sort(cons(y_1, x_1))$ incrementally. But while we incrementalize the unfolded $sort(cons(y_1, x_1))$ to get a definition of $sort'$, the application $least(cons(y_1, x_1))$ is transformed recursively, which results in lifting of some conditions and bindings, and then applications of $rest$ are transformed under these conditions and bindings. As the result of these transformations, $sort'$ compares y_1 with the first number in r_1 to decide whether y_1 should stay before r_1, and, if not, recursively considers y_1 with the rest of r_1. But this is exactly the process of inserting y_1 into r_1 at the right place. Thus, to a certain degree, the derivation procedure *discovered* the insertion process from the selection sort via a series of transformations.

As the derivation procedure is more complicated, an informal but complete derivation is given below. As just mentioned, to compute $sort(cons(y, x))$ incrementally using $sort(x) = r$, we start by introducing $sort'(y_1, x_1, r_1)$ for $sort(cons(y_1, x_1))$ with $sort(x_1) = r_1$.

1. Unfold $sort(cons(y_1, x_1))$:

$$
\begin{array}{l}
\textbf{if } null(cons(y_1,x_1)) \textbf{ then } nil \\
\textbf{else let } k = least(cons(y_1,x_1)) \textbf{ in } cons(k, sort(rest(cons(y_1,x_1),k))) \textbf{ end}
\end{array}
\tag{32}
$$

2. Simplify the condition $null(cons(y_1, x_1))$ to false, and thus (32) is to be simplified to the false branch:

$$
\textbf{let } k = least(cons(y_1,x_1)) \textbf{ in } cons(k, sort(rest(cons(y_1,x_1),k))) \textbf{ end} \tag{33}
$$

3. Consider $least(cons(y_1, x_1))$, and introduce $least'(y_2, x_2, r_2)$ for $least(cons(y_2, x_2))$ with $sort(x_2) = r_2$.

   3.1. Unfold $least(cons(y_2, x_2))$:

$$
\begin{array}{l}
\textbf{if } null(cdr(cons(y_2,x_2))) \textbf{ then } car(cons(y_2,x_2)) \\
\textbf{else let } s = least(cdr(cons(y_2,x_2))) \\
\quad\quad \textbf{in if } car(cons(y_2,x_2)) \le s \textbf{ then } car(cons(y_2,x_2)) \textbf{ else } s \textbf{ end}
\end{array}
\tag{34}
$$

   3.2. Simplify the condition $null(cdr(cons(y_2, x_2)))$ to $null(x_2)$, and replace $null(x_2)$ by $null(r_2)$ since

$$
null(sort(x_2)) \text{ is specialized to true (false)} \quad \text{when } null(x_2) \text{ is true (false).}
$$

   In the true branch, simplify $car(cons(y_2, x_2))$ to y_2. In the false branch, simplify $least(cdr(cons(y_2, x_2)))$ to $least(x_2)$, and replace $least(x_2)$ by $car(r_2)$ since when $null(r_2)$ is false

$$
sort(x_2) \text{ is specialized to } \quad \textbf{let } k_2 = least(x_2) \textbf{ in } cons(k_2, sort(rest(x_2,k_2))) \textbf{ end},
$$

   and then in the body of the **let** expression simplify $car(cons(y_2, x_2))$ to y_2. Thus, (34) becomes

$$
\textbf{if } null(r_2) \textbf{ then } y_2 \textbf{ else let } s = car(r_2) \textbf{ in if } y_2 \le s \textbf{ then } y_2 \textbf{ else } s \textbf{ end} \tag{35}
$$

3.3. For $least'$ defined by $least'(y_2, x_2, r_2) = (35)$, eliminate dead parameter x_2.

Replace $least(cons(y_1, x_1))$ by $least'(y_1, r_1)$, and unfold $least'(y_1, r_1)$ since $least'$ is not recursively defined and unfolding does not duplicate non-trivial computations. Thus, (33) becomes

$$\begin{array}{l}\textbf{let } k = (\textbf{if } null(r_1) \textbf{ then } y_1 \textbf{ else let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } y_1 \textbf{ else } s \textbf{ end}) \\ \textbf{in } cons(k, sort(rest(cons(y_1, x_1), k))) \textbf{ end}\end{array} \tag{36}$$

4. Lift the first condition $null(r_1)$ out of the top-level **let**. Thus, (36) becomes

$$\begin{array}{l}\textbf{if } null(r_1) \textbf{ then let } k = y_1 \textbf{ in } cons(k, sort(rest(cons(y_1, x_1), k))) \textbf{ end} \\ \textbf{else let } k = (\textbf{let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } y_1 \textbf{ else } s \textbf{ end}) \\ \qquad \textbf{in } cons(k, sort(rest(cons(y_1, x_1), k))) \textbf{ end}\end{array} \tag{37}$$

5. In the true branch of (37), simplify x_1 to nil. No functions are introduced for the applications of $rest$ and $sort$ since they do not depend on x. Then unfold the **let**:

$$cons(y_1, sort(rest(cons(y_1, nil), y_1))) \tag{38}$$

In the false branch of (37), lift the binding $s = car(r_1)$ out of the first **let**:

$$\begin{array}{l}\textbf{let } s = car(r_1) \textbf{ in let } k = (\textbf{if } y_1 \le s \textbf{ then } y_1 \textbf{ else } s) \\ \qquad\qquad\qquad \textbf{in } cons(k, sort(rest(cons(y_1, x_1), k))) \textbf{ end end}\end{array}$$

and then lift the condition $y_1 \le s$ out of the second **let**:

$$\begin{array}{l}\textbf{let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then let } k = y_1 \textbf{ in } cons(k, sort(rest(cons(y_1, x_1), k))) \textbf{ end} \\ \qquad\qquad\qquad \textbf{else let } k = s \textbf{ in } cons(k, sort(rest(cons(y_1, x_1), k))) \textbf{ end end}\end{array} \tag{39}$$

6. In the true branch of (39), first consider $rest(cons(y_1, x_1), k)$, and introduce $rest'(y_3, x_3, k_3, r_3)$ for $rest(cons(y_3, x_3), k_3)$ with $sort(x_3) = r_3$ and also $k_3 \leftrightarrow y_3$, *etc.*

6.1 Unfold $rest(cons(y_3, x_3), k_3)$:

$$\begin{array}{l}\textbf{if } k_3 = car(cons(y_3, x_3)) \textbf{ then } cdr(cons(y_3, x_3)) \\ \textbf{else } cons(car(cons(y_3, x_3)), rest(cdr(cons(y_3, x_3)), k_3))\end{array} \tag{40}$$

6.2 Simplify the condition $k_3 = car(cons(y_3, x_3))$ to $k_3 = y_3$, and further simplify it to true, and thus (40) is to be simplified to the true branch $cdr(cons(y_3, x_3))$, which is then simplified to x_3.

6.3 For $rest'$ defined by $rest'(y_3, x_3, k_3, r_3) = x_3$, eliminate dead parameters y_3, k_3, and r_3.

Replace $rest(cons(y_1, x_1), k)$ by $rest'(x_1)$, and unfold $rest'(x_1)$ to x_1. Then, replace $sort(x_1)$ by r_1. Finally, unfold the **let**. Thus, the true branch of (39) becomes

$$cons(y_1, r_1) \tag{41}$$

7. In the false branch of (39), first consider $rest(cons(y_1, x_1), k)$, and introduce $rest'_1(y_4, x_4, k_4, r_4)$ for $rest(cons(y_4, x_4), k_4)$ with $sort(x_4) = r_4$ and also $y_4 \le s \leftrightarrow F, k_4 \leftrightarrow s$, *etc.*

7.1 Unfold $rest(cons(y_4, x_4), k_4)$:

$$\begin{array}{l}\textbf{if } k_4 = car(cons(y_4, x_4)) \textbf{ then } cdr(cons(y_4, x_4)) \\ \textbf{else } cons(car(cons(y_4, x_4)), rest(cdr(cons(y_4, x_4)), k_4))\end{array} \tag{42}$$

7.2 Simplify the condition $k_4 = car(cons(y_4, x_4))$ to $k_4 = y_4$, and further simplify it to false, and thus (42) is to be simplified to the false branch, which is then simplified to $cons(y_4, rest(x_4, k_4))$.

7.3 For $rest'_1$ defined by $rest'_1(y_4, x_4, k_4, r_4) = cons(y_4, rest(x_4, k_4))$, eliminate dead parameter r_4.

Replace $rest(cons(y_1, x_1), k)$ by $rest'_1(y_1, x_1, k)$, and unfold $rest'_1(y_1, x_1, k)$ to $cons(y_1, rest(x_1, k))$. Then, replace $sort(cons(y_1, rest(x_1, k)))$ by $sort'(y_1, rest(x_1, k), cdr(r_1))$, since when $null(r_1)$ is false

$$sort(x_1) \text{ is specialized to } \quad \textbf{let } k_1 = least(x_1) \textbf{ in } cons(k_1, sort(rest(x_1, k_1))) \textbf{ end},$$

which implies $sort(rest(x_1, k)) = cdr(r_1)$ for $k = s = car(r_1) = k_1$. Finally, unfold the **let**. Thus, the false branch of (39) becomes

$$cons(s, sort'(y_1, rest(x_1, s), cdr(r_1))) \tag{43}$$

8. Putting (37) (38) (39) (41) (43) together, $sort'$ is defined by

$$\begin{array}{rl} sort'(y_1, x_1, r_1) = & \textbf{if } null(r_1) \textbf{ then } cons(y_1, sort(rest(cons(y_1, nil), y_1))) \\ & \textbf{else let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } cons(y, r_1) \\ & \qquad\qquad\qquad\qquad\qquad \textbf{else } cons(s, sort'(y_1, rest(x_1, s), cdr(r_1))) \textbf{ end} \end{array}$$

Eliminating dead parameter x_1, we obtain a final definition of $sort'$:

$$\begin{array}{rl} sort'(y_1, r_1) = & \textbf{if } null(r_1) \textbf{ then } cons(y_1, sort(rest(cons(y_1, nil), y_1))) \\ & \textbf{else let } s = car(r_1) \textbf{ in if } y_1 \le s \textbf{ then } cons(y_1, r_1) \\ & \qquad\qquad\qquad\qquad\qquad \textbf{else } cons(s, sort'(y_1, cdr(r_1))) \textbf{ end} \end{array} \tag{44}$$

It is easy to see that, in the true branch, $sort(rest(cons(y_1, nil), y_1))$ returns nil in constant time given any number y_1; in the false branch, the **let** expression could be unfolded. Thus, $sort'$ does exactly an insertion as the $insert$ in (28).

At the end, we have $sort(cons(y, x)) = sort'(y, r)$ for $sort(x) = r$, where $sort'$ is defined as in (44).

7.3 Merge Sort

Suppose the program is a merge sort that divides the list into two sublists, recursively sorts the two sublists, and then merges the two sorted sublists.

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

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

$$\begin{array}{rl} even(x) = & \textbf{if } null(x) \textbf{ then } nil \\ & \textbf{else } odd(cdr(x)) \end{array}$$

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

An insertion can be easily obtained if we are given the property that sorting the new list equals merging the new number into the previously sorted list.

$$
\begin{array}{lll}
sort'(y_1,x_1,r_1) & = sort(cons(y_1,x_1)), \quad \text{with } sort(x_1)=r_1 & \text{function introduction} \\
 & = merge(cons(y_1,nil), sort(x_1)) & \text{property relating } merge \text{ and } sort \\
 & = merge(cons(y_1,nil), r_1) & \text{replacement} \\
sort'(y_1,r_1) & = merge(cons(y_1,nil), r_1) & \text{dead parameter elimination} \\
sort(cons(y,x)) & = sort'(y,r), \quad \text{for } sort(x)=r & \text{use of the introduced function}
\end{array}
$$

The derived program $sort'$ basically performs an insertion with a constant factor overhead over the $insert$ in (28). The required property that relates $merge$ and $sort$ can be proved by a straightforward induction based on the associativity and commutativity of $merge$. However, if the above property is not given, then no incremental program can be derived using the derivation procedure. But what is interesting is the following.

Suppose we cache intermediate results during merge sort, i.e., we recursively cache sorted sublists. Then following the derivation procedure, we can easily obtain an *incremental merge sort* that incrementally sorts the new list by recursively merging the new number with the appropriate intermediate results.

An informal derivation with a few major transformation steps is given below. Let $\widehat{sort}$ be a function that extends $sort$ with cached intermediate results as auxiliary information.

$$
\begin{array}{ll}
\widehat{sort}(x) = & \textbf{if } null(x) \textbf{ then } con(nil, aux_0) \\
 & \textbf{else if } null(cdr(x)) \textbf{ then } con(cons(car(x), nil), aux_1) \\
 & \textbf{else let } so = \widehat{sort}(odd(x)) \textbf{ in} \\
 & \quad\quad \textbf{let } se = \widehat{sort}(even(x)) \textbf{ in} \\
 & \quad\quad con(merge(con_r(so), con_r(se)),\ aux(so,se)) \textbf{ end end}
\end{array}
$$

$$
sort(x) = con_r(\widehat{sort}(x))
$$

$\widehat{sort}(x)$ returns a con constructor application, where the first element is the value of $sort(x)$ and can be selected using con_r, and the second element is the auxiliary information corresponding to x and can be selected using con_a. The auxiliary information corresponding to x is a constant aux_0 if x is nil, a constant aux_1 if x is a single element list, and an aux construction of $\widehat{sort}(odd(x))$ and $\widehat{sort}(even(x))$ otherwise, where the first element can be selected using aux_o and second aux_e.

We start by introducing $\widehat{sort}'(y_1, x_1, \hat{r_1})$ to compute $\widehat{sort}(cons(y_1, x_1))$ incrementally using $\widehat{sort}(x_1) = \hat{r_1}$.

1. Unfold $\widehat{sort}(cons(y_1, x_1))$:

$$
\begin{array}{l}
\textbf{if } null(cons(y_1,x_1)) \textbf{ then } con(nil, aux_0) \\
\textbf{else if } null(cdr(cons(y_1,x_1))) \textbf{ then } con(cons(car(cons(y_1,x_1)), nil), aux_1) \\
\textbf{else let } so = \widehat{sort}(odd(cons(y_1,x_1))) \textbf{ in} \\
\quad\quad \textbf{let } se = \widehat{sort}(even(cons(y_1,x_1))) \textbf{ in} \\
\quad\quad con(merge(con_r(so), con_r(se)),\ aux(so,se)) \textbf{ end end}
\end{array}
\tag{45}
$$

2. Simplify the first condition $null(cons(y_1, x_1))$ to false, and thus the first branch is to be eliminated. Simplify the second condition $null(cdr(cons(y_1, x_1)))$ to $null(x_1)$, and in the second branch simplify $car(cons(y_1, x_1))$ to y_1. In the third branch, simplify $odd(cons(y_1, x_1))$ to $cons(y_1, even(x_1))$, and $even(cons(y_1, x_1))$ to $odd(x_1)$. We get

$$
\begin{array}{l}
\textbf{if } null(x_1) \textbf{ then } con(cons(y_1,nil), aux_1) \\
\textbf{else let } so = \widehat{sort}(cons(y_1, even(x_1))) \textbf{ in} \\
\quad\quad \textbf{let } se = \widehat{sort}(odd(x_1)) \textbf{ in} \\
\quad\quad con(merge(con_r(so), con_r(se)),\ aux(so,se)) \textbf{ end end}
\end{array}
\tag{46}
$$

3. When $null(x_1)$ is false, $\widehat{sort}(x_1)$ is specialized to a conditional expression with condition $null(cdr(x_1))$. Thus the false branch of (46) is to be separated into two corresponding cases.

4. $null(x_1)$ is replaced by $null(con_r(\hat{r_1}))$ since

$$null(con_r(\widehat{sort}(x_1))) \text{ is specialized to true (false)} \quad \text{when } null(x_1) \text{ is true (false).}$$

$null(cdr(x_1))$ is replaced by $null(cdr(con_r(\hat{r_1})))$ since

$$null(cdr(con_r(\widehat{sort}(x_1)))) \text{ is specialized to true (false)} \quad \text{when } null(cdr(x_1)) \text{ is true (false).}$$

In the branch where $null(cdr(x_1))$ is true, $even(x_1)$ becomes nil, then $\widehat{sort}(cons(y_1, nil))$ becomes $con(cons(y_1, nil), aux_1)$; $odd(x_1)$ becomes $cons(car(x_1), nil)$, then $\widehat{sort}(cons(car(x_1), nil))$ becomes $con(cons(car(x_1), nil), aux_1)$, which is then replaced by $\hat{r_1}$ since

$$\widehat{sort}(x_1) \text{ is specialized to } con(cons(car(x_1), nil), aux_1) \quad \text{when } null(cdr(x_1)) \text{ is true.}$$

In the other branch where $null(cdr(x_1))$ is false,

$$\widehat{sort}(x_1) \text{ is specialized to } con(..., aux(\widehat{sort}(odd(x_1)), \widehat{sort}(even(x_1)))),$$

which implies $\widehat{sort}(odd(x_1)) = aux_o(con_a(\hat{r_1}))$ and $\widehat{sort}(even(x_1)) = aux_e(con_a(\hat{r_1}))$; thus $\widehat{sort}(cons(y_1, even(x_1)))$ is replaced by $\widehat{sort}'(y_1,\ even(x_1),\ aux_e(con_a(\hat{r_1})))$, and $\widehat{sort}(odd(x_1))$ is replaced by $aux_o(con_a(\hat{r_1}))$. We get

$$\begin{array}{l}
\textbf{if } null(con_r(\hat{r_1})) \textbf{ then } con(cons(y_1, nil), aux_1) \\
\textbf{else if } null(cdr(con_r(\hat{r_1}))) \textbf{ then} \\
\quad \textbf{let } so = con(cons(y_1, nil), aux_1) \textbf{ in} \\
\quad con(merge(con_r(so), con_r(\hat{r_1})),\ aux(so, \hat{r_1})) \textbf{ end} \\
\textbf{else let } so = \widehat{sort}'(y_1, even(x_1), aux_e(con_a(\hat{r_1}))) \textbf{ in} \\
\quad \textbf{let } se = aux_o(con_a(\hat{r_1})) \textbf{ in} \\
\quad con(merge(con_r(so), con_r(se)),\ aux(so, se)) \textbf{ end end}
\end{array} \tag{47}$$

5. Eliminate dead parameter of $\widehat{sort}'$, defined by $\widehat{sort}'(y_1, x_1, \hat{r_1}) = (47)$. Obviously x_1 is dead. We obtain a final definition of $\widehat{sort}'$:

$$\begin{array}{rl}
\widehat{sort}'(y_1, \hat{r_1}) = & \textbf{if } null(con_r(\hat{r_1})) \textbf{ then } con(cons(y_1, nil), aux_1) \\
& \textbf{else if } null(cdr(con_r(\hat{r_1}))) \textbf{ then} \\
& \quad \textbf{let } so = con(cons(y_1, nil), aux_1) \textbf{ in} \\
& \quad con(merge(con_r(so), con_r(\hat{r_1})),\ aux(so, \hat{r_1})) \textbf{ end} \\
& \textbf{else let } so = \widehat{sort}'(y_1, aux_e(con_a(\hat{r_1}))) \textbf{ in} \\
& \quad \textbf{let } se = aux_o(con_a(\hat{r_1})) \textbf{ in} \\
& \quad con(merge(con_r(so), con_r(se)),\ aux(so, se)) \textbf{ end end}
\end{array} \tag{48}$$

At the end, we have $\widehat{sort}(cons(y, x)) = \widehat{sort}'(y, \hat{r})$ for $\widehat{sort}(x) = \hat{r}$, where $\widehat{sort}'$ is defined as in (48). $\widehat{sort}'(y, \hat{r})$ incrementally sorts the new list and maintains the corresponding auxiliary information.

Both insertion sort and selection sort take $O(n^2)$ time, where n is the length of the input list, and merge sort takes $O(n \log n)$ time. Insertion takes only $O(n)$ time; but it uses $O(n)$ space to store the previously sorted list. Incremental merge sort also takes $O(n)$ time; but it uses $O(n \log n)$ space to store intermediate results.

The derivation of the incremental merge sort suggests that our approach of exploiting cached values for incrementality is powerful: the power of caching obviates the reliance on a theorem prover for proving certain properties. We can also view it as trading space for theorem proving abilities.

8 Related Work and Conclusion

Our approach to deriving incremental programs combines a number of program analysis and transformation techniques, which have been summarized in Section 6.1. Here, we take a closer look at related work in incremental computation, which is introduced in Section 1 and partitioned into three classes.

First of all, given particular problems with certain input changes, can our approach be used to derive as efficient incremental programs as those in the first class? The general answer is positive, but with three caveats. First, the particular problem needs to be coded naturally in the language for which our approach is presented. Second, the quality of a derived incremental program depends on the way the non-incremental program is coded, as seen in the sort examples in Section 7. Third, auxiliary information is needed for many incremental problems but may be difficult to discover. In this case, we can use the ideas in Section 6.3, at least use the derivation procedure on programs that are extended to compute manually discovered auxiliary information, and derive programs that incrementally maintain the auxiliary information.

Since our transformational approach is related to partial evaluation in some aspects, it is worthwhile to specially compare it with the work by Sundaresh and Hudak [43, 42] in the second class. The common aspect is that both works aim at obtaining incremental computation by transforming non-incremental programs. However, two approaches follows different lines. Their work mostly *uses* partial evaluation, with extra efforts on partitioning program inputs and combining residual programs. Our method combines a series of analysis and transformation techniques that "parallel" those used in (generalized) partial evaluation, but with the goal of incrementalization in addition to specialization, and therefore employs overall more extensive and more complicated techniques. We believe a major limitation of the Sundaresh-Hudak framework is that it can only handle input changes according to a pre-given input partition, which is partly implied as a work in the second class.

Our work is closest in spirit to the finite differencing techniques of the third class. The name "finite differencing" was originally given by Paige and Koenig [31]. Their work generalizes Cocke's strength reduction [9] and provides a convenient framework for implementing a host of transformations including Earley's "iterator inversion" [10]. They develop a set of rules for differentiating set-theoretic expressions and combine these rules using a chain rule to derive inexpensive programs with incremental loop bodies. Such techniques are indispensable as part of an optimizing compiler for languages like SETL or APL [30, 7]. The APTS program transformation system [29] has been developed for such purposes. Our technique differs from theirs in that it applies to programs written in a standard language like Lisp. In general, such programs are written at a lower abstraction level so that a fixed set of rules for differentiating expressions involving complex objects like sets is not sufficient. The technique we propose can be regarded as a principle and a systematic approach, through which incrementalities can be *discovered* in existing programs written in standard languages.

Smith's work in KIDS [39, 40] is closely related to ours. KIDS is a semi-automatic program development system that aims to derive efficient programs from high-level specifications [41], as is APTS. Its version of finite differencing was developed for the optimization of its derived functional programs and has two basic operations: abstraction and simplification. Abstraction of a function f adds an extra cache parameter to f. Simplification simplifies the definition of f given the added cache parameter. However, as to *how* the cache parameter should be used in the simplification to provide incrementality, KIDS provides only the observation that distributive laws can often be applied. The Munich CIP project [32] has a strategy for finite differencing that captures similar ideas. It first "defines by a suitable embedding a function f'", and then "derives a recursive version of f' using generalized unfold/fold strategy", but provides no special techniques for discovering incrementality. Both works provide only general strategies with no precise procedure to follow, and therefore are much less automatable than ours.

A simpler version of our approach for deriving incremental programs was proposed earlier [21]. The derivation there consists of four steps: (1) expanding the computation of $f_0(x \oplus y)$, (2) introducing $f_0'(x, y, r)$ with the cache parameter $r = f_0(x)$, (3) replacing subcomputations depending on x by retrievals from r, and (4) eliminating dead computations. It has the major limitation that only *one* incremental function, namely

f_0', is introduced to compute $f_0(x \oplus y)$ incrementally. Therefore, it can not derive, for example, the functions in Figure 2 from those in Figure 1. The present paper implements an improved version that overcomes this limitation following an idea proposed in [21], where it was called "focusing and switching".

The simpler version can be regarded as a special case of the improved version. To see this, let $\mathcal{I}nc\mathcal{A}pply$ introduce only f_0' for $f_0(x \oplus y)$ and replace only appropriate applications of f_0 by applications of f_0'. For any other application $f(e_1, ..., e_n)$, let $\mathcal{I}nc\mathcal{A}pply$ apply $\mathcal{I}nc$ in-place to the unfolded application under some condition that *approximates* the condition of function unfolding in the first step of the simpler version. Then, the following equation *approximately* holds:

$$\mathcal{I}nc\mathcal{A}pply[\![f_0(x \oplus y)]\!]\,\emptyset\,\{\langle f_0(x), r, \emptyset\rangle\}\,\emptyset = \mathcal{E}lim[\![f_0'(x, y, r)]\!]\ \{\langle f_0(x \oplus y), f_0'(x, y, r), \{\langle f_0(x), r, \emptyset\rangle\}\rangle\}$$

$$\text{for } f_0' \text{ defined by} \qquad f_0'(x, y, r) = \mathcal{S}_t[\![\mathcal{D}_t[\![\mathcal{G}_o[\![f_0(x \oplus y)]\!]\,\emptyset]\!]\,\emptyset]\!]\,\emptyset\,\{\langle f_0(x), r\rangle\}$$

where $\mathcal{G}_o$, $\mathcal{D}_t$, and $\mathcal{S}_t$ implement the first three steps of the simpler version, respectively. The words *approximate* and *approximately* are used since the improved version may have replaced subcomputations of the arguments $e_1, ..., e_n$ by retrievals from cached results before considering the application of f, but the simpler version unfolds an application in the first step based on some condition of the arguments that does not involve cached results.

We conclude with the contribution of our work to a general model of incremental computation, namely, a model $\mathcal{M}$ that takes a (non-incremental) program f written in some language $\mathcal{L}$ and an input change $\oplus$, which is also describable in $\mathcal{L}$, and derives f', an incremental version of f under $\oplus$. Such a model addresses all three classes of work in incremental computation, for the following reasons. The development of particular incremental algorithms in the first class is a special case of $\mathcal{M}$, where f and $\oplus$ are fixed according to particular problems, and f' is derived manually. An incremental execution framework in the second class is a kind of $\mathcal{M}$ that is general in that it automatically incrementalizes any application program f, but has poor specializability in that any change $\oplus$ to program f is handled in the way prescribed by the framework (and often no explicit f' is derived). Work in the third class is not only general, but also specialized to any program f and change $\oplus$; however, so far effective methods focus on the class of $\mathcal{M}$ where the language $\mathcal{L}$ is limited to very high-level languages. What is needed is an effective approach for deriving incremental programs from non-incremental ones written in a standard language.

We have presented such a systematic approach for deriving incremental programs from non-incremental programs written in a standard functional programming language. It begins the study of a general model for incremental computation along unique lines distinct from all other approaches. Although this problem is, in general, very hard, we have shown that an effective approach can be developed to derive incremental programs by effectively combining particular program transformation and analysis techniques.

Although we presented our approach in terms of a first-order functional language with strict semantics, we have reason to believe that our basic principle applies to other standard languages as well, e.g., higher-order functional languages, functional languages with lazy semantics, and imperative languages. Of course, special program analysis and transformation techniques related to these language features must be exploited, and they may complicate the derivation issues in one way or another, just as when partial evaluation techniques are developed to cope with such language features. On the other hand, these other language features allow some algorithms to be coded more naturally and incremental versions derived to be more efficient, making a general model for incremental computation more complete.

By studying these general techniques, we aim to better understand the essence of incremental computation. We also aim to establish a general framework in which different ideas on incremental computation can be integrated. By specializing the general techniques to different applications, we will be able to obtain particular incremental algorithms, particular incremental computation techniques, and particular incremental computation languages. Their applications could include most problems discussed in the literature [34].

References

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

[2] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, January 1990.

[3] R. A. Ballance, S. L. Graham, and M. L. V. D. Vanter. The *Pan* language-based editing system. *ACM Transactions on Programming Languages and Systems*, 1(1):95–127, January 1992.

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

[5] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

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

[7] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, September 1988/89.

[8] S. Ceri, M. A. W. Houtsma, A. M. Keller, and P. Samarati. Achieving incremental consistency among autonomous replicated databases. *IFIP Transactions A [Computer Science and Technology]*, A-25:223–237, 1993.

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

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

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

[12] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LISP and Functional Programming*, pages 307–322, 1990.

[13] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[14] Y. Futamura and K. Nogi. Generalized partial evaluation. In *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[15] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, July 1979.

[16] 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.

[17] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[18] 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.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[20] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, December 1992.

[21] Y. Liu and T. Teitelbaum. Deriving incremental programs. Technical Report TR 93-1384, Department of Computer Science, Cornell University, Ithaca, New York, September (Revised October) 1993.

[22] R. Medina-Mora and P. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, SE-7(5):472–482, September 1981.

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

[24] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[25] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[26] R. Paige. *Formal Differentiation: A Program Synthesis Technique*, volume 7 of *Computer Science. Artificial Intelligence*. UMI Research Press, Ann Arbor, Michigan, 1981. Revision of Ph.D. Thesis - New York University, 1979.

[27] 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.

[28] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory, Vol. 2*, pages 171–210. Plenum Press, New York, March 1984.

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

[30] R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient ram code - a case study. *Journal of Symbolic Computation*, 4(2):207–232, 1987.

[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] 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.

[34] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 502–510, Charleston, South Carolina, January 1993.

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

[36] 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.

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

[38] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

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

[40] D. R. Smith. KIDS - a knowledge-based software development system. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, chapter 19, pages 483–514. AAAI Press/The MIT Press, 1991. Proceedings of the Workshop on Automating Software Design, AAAI '88.

[41] D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305–321, 1990.

[42] 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).

[43] 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.

[44] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[45] B. T. Vander Zanden. Incremental constraint satisfaction and its application to graphical interfaces. Ph.D. Thesis TR 88-941, Department of Computer Science, Cornell University, October 1988.

[46] A. Varma and S. Chalasani. An incremental algorithm for TDM switching assignments in satellite and terrestrial networks. *IEEE Journal on Selected Areas in Communications*, 10(2):364–377, February 1992.

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

[48] B. Wegbreit. Goal-directed program transformation. *IEEE Transactions on Software Engineering*, SE-2(2):69–80, June 1976.

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

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

[51] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991.