# Deriving Incremental Programs

Yanhong A. Liu* Tim Teitelbaum*
Department of Computer Science, Cornell University, Ithaca, NY 14853
{yanhong, tt}@cs.cornell.edu

October 1993

**Abstract**

A systematic stepwise transformational approach is given for deriving incremental programs from non-incremental programs. We exploit partial evaluation, other static analysis and transformation techniques, and domain-specific knowledge in order to provide a degree of incrementality not otherwise achievable by a generic incremental evaluator. The generality of our model of incremental computation is discussed and compared with related work. Illustrative examples using the transformational approach are given.

## 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 computation of $f(x)$ with result $r$ and a new input $x \oplus \delta x$ to the program $f$, an incremental program $f'$ computes the result of $f(x \oplus \delta x)$ by making use of $r$, avoiding computations previously done by $f(x)$. Methods of incremental computation have widespread application throughout software, e.g., optimizing compilers[1, 7, 8], transformational programming[21, 37], interactive editing systems[30, 3], *etc.* Based on the observation that explicitly incremental programs are hard to write, we aim to provide incremental computation automatically from non-incremental programs.

In this paper, we present a technique for formally deriving an incremental program $f'$ from a non-incremental program $f$ and input change $\oplus$. We exploit partial evaluation, other static analysis and transformation techniques, and domain-specific knowledge in an attempt to provide a degree of incrementality not otherwise achievable by a generic incremental evaluator. The technique is presented as a four-step transformation: (1) expanding the computation $f(x \oplus \delta x)$ to separate

*The authors gratefully acknowledge the support of the Office of Naval Research under contract No. N00014-92-J-1973

computations on $x$ from the rest, (2) introducing the cache parameter $r = f(x)$, (3) replacing equivalent computations on $x$ by retrieving results from $r$, and (4) eliminating redundant code. In the third step, a subroutine called auxiliary generalized partial evaluation is employed to help discover incrementalities. We also cast our derivation approach in terms of a generalized partial evaluation point of view.

Paige's work on finite differencing is the pioneering work on deriving incremental programs[24, 20, 36, 6]. Whereas Paige's methods are for very high-level set-theoretic languages, our techniques apply to general functional programs, as does Smith's approach in KIDS[37, 38]. For further discussions and comparisons between these methods, see Section 8.

The rest of the paper is organized as follows. In Section 2, we give the definition of incremental programs. Our basic approach for the derivation of incremental programs is outlined in Section 3. The formalization of the problem with the language we are using is presented in Section 4. In Section 5, we formalize the transformations for the derivation and argue that they preserve the semantics of the programs. An example using the transformations is given in Section 6, in which an insert program is derived from a sort program. Several related issues, including a limitation of the approach and directions for remedying them, are discussed in Section 7. Finally, we discuss related work in incremental computation in Section 8. The generality of our model of incremental computation is discussed in the last two sections as well.

## 2 Incremental Programs

Let $f$ be a program, and let $x$ be an input for $f$. Let $\delta x$ be a *change* in the value of $x$, and let $\oplus$ be a *change operation* that combines value $x$ and change $\delta x$ to produce a new input value $x \oplus \delta x$. In general, the domains of $x$ and $\delta x$ need not be the same. For example, consider a list of integers $x$, an integer $\delta x$, and $x \oplus \delta x = cons(\delta x, x)$; i.e., $x$ is changed by prepending integer $\delta x$ at the head of the list.

Let $r = f(x)$; i.e., $r$ is the result of executing program $f$ on input $x$. We call $r$ the *cached result* and aim to make use of it in computing $f(x \oplus \delta x)$. Let $f'_\oplus$ be a program such that whenever $r$ is the cached result $f(x)$, $f'_\oplus(x, \delta x, r)$ computes $f(x \oplus \delta x)$ for all possible $x$ and $\delta x$, i.e.,

$$(\forall x)(\forall \delta x)(\forall r)\ [f(x) = r \implies f'_\oplus(x, \delta x, r) \approx f(x \oplus \delta x)], \tag{1}$$

where '$\approx$' means that either both sides terminate with the same value or both diverge. Then $f'_\oplus$ is called an *incremental version* of $f$ under $\oplus$ if, in addition to satisfying (1), $f'_\oplus(x, \delta x, r)$ computes faster than $f(x \oplus \delta x)$ for almost all $x$ and $\delta x$ under some time model and makes *non-trivial* use of

$r$, i.e., it is not that $f'(x, \delta x, c) \approx f(x \oplus \delta x)$ for almost all $x$, $\delta x$, and $c$.

For example, assume the RAM model and let $sort(x)$ be a selection sort on a list $x$ of numbers that uses a naive select (which finds a least element in a list by traversing the list from the beginning to the end). Then, $sort(x)$ takes time $O(n^2)$ for $x$ of length $n$. Consider the change operation $x \oplus \delta x = cons(\delta x, x)$, where $\delta x$ is a number. There exists a $sort'(x, \delta x, r)$ that, when $r$ is $sort(x)$, computes $sort(cons(\delta x, x))$ by inserting $\delta x$ in $r$ in $O(n)$ time. In particular, $sort'$ makes non-trivial use of $r$. Thus, $sort'$ is an incremental version of $sort$. In contrast, consider a $sort''(x, \delta x, r)$ that computes $sort(cons(\delta x, x))$ by doing a merge sort on $cons(\delta x, x)$ and takes only $O(n \log n)$ time. $sort''$ is not an incremental version of $sort$ since it gains its speedup without the use of $r$.

Suppose $f'_\oplus$ is an incremental version of $f$ under $\oplus$. We say that $f'_\oplus$ is a *complete incremental version* if $f'_\oplus$ does not depend on the parameter $x$, i.e., $f'_\oplus(c, \delta x, r) \approx f(x \oplus \delta x)$ for all $c$, $x$, and $\delta x$. In other words, $f'_\oplus(x, \delta x, r)$ computes $f(x \oplus \delta x)$ but uses only the change parameter $\delta x$ and the cached result $r$. In the sort example, $sort'$ is a complete incremental version of $sort$ under $\oplus$.

For a formal definition of incremental programs, see [19]. When no confusion arises, we simply write $f'$ for $f'_\oplus$. For typographical convenience, we shall typically use $y$ for the change parameter rather than $\delta x$. Also, throughout the paper, $r$ always refers to the cached result of a previous computation $f(x)$.

In [19], it is proved that in general, given a program $f$ and an input change operation $\oplus$, whether an incremental version exists is undecidable. Accordingly, we compromise by seeking an approximation to an incremental version, which we call a *differentiated version*. A differentiated version $f'$ computes the same value as the original program $f$ only if both of them terminate, i.e.,

$$(\forall x)(\forall \delta x)(\forall r)\ [\ f(x) = r \Longrightarrow (\forall d_1)(\forall d_2)\ [\ f'_\oplus(x, \delta x, r) = d_1 \wedge f(x \oplus \delta x) = d_2 \Longrightarrow d_1 = d_2\ ]\ ]. \qquad (2)$$

Also, $f'$ computes at least as fast as the original program for almost all inputs, while an incremental version computes strictly faster than the original program for almost all inputs and makes non-trivial use of $r$. If a differentiated version only uses the change parameter $\delta x$ and the cached result $r$, then it is called a *complete differentiated version*.

Obviously, a differentiated version $f'(x, \delta x, r)$ can be trivially defined to be $f(x \oplus \delta x)$. But then no efficiency is gained by $f'$, which is of no interest. The goal is to make $f'$ as efficient as possible by having it use the cached result $r$ as much as possible.

## 3 Derivation Approach

**Basic Ideas.** We aim to identify in the computation of $f(x \oplus y)$ those computations that are also performed in the computation of $f(x)$ and avoid recomputing them if possible.

More precisely, by expanding the computation of $f(x \oplus y)$, those computations on $x$ in $f(x \oplus y)$ can be separated out from the rest of the computation. Given the cached result $r$ of a previous computation of $f(x)$, the results of some of the computations on $x$ in $f(x \oplus y)$ may be retrieved more efficiently from $r$ than computing them from scratch. This more efficient way to compute $f(x \oplus y)$ is captured in the definition of the differentiated version $f^*(x, y, r)$.

In order to explain the basic ideas further, we break the derivation approach into four steps.

Step 1. Expanding the computation of $f(x \oplus y)$ to separate out the computations on $x$ from the rest of computation. This is achieved by unfolding [5] and simplifying function applications whose arguments involve both $x$ and $y$.

Step 2. Introducing the cached result $r$ of the computation $f(x)$. In this step, we define $f^*(x, y, r)$ to be the expanded computation of $f(x \oplus y)$ obtained from Step 1. In particular, if an application of $f$ whose arguments involve both $x$ and $y$ occurs recursively in the expanded computation of $f(x \oplus y)$, it is replaced by an instance of $f^*(x, y, r)$, if possible. Thus, the computation of $f(x \oplus y)$ is performed by $f^*(x, y, r)$.

Step 3. Using the cached result $r$ in the computation of $f^*(x, y, r)$. In the definition of $f^*(x, y, r)$ obtained from Step 2, we find computations that are also performed by $f(x)$, and replace them by retrievals from the cached result $r$ where possible. As a straightforward example, an occurrence of $f(x)$ in the definition of $f^*(x, y, r)$ can be replaced by $r$. Moreover, within the context of a subexpression in the definition of $f^*(x, y, r)$, the computation $f(x)$ may have a specialized form. If the subexpression has that specialized form, then it can also be replaced by $r$.

Step 4. Eliminating redundant computations in the resulting definition obtained from Step 3. This can be done using standard optimizations. In particular, since some computations depending on $x$ have been replaced by computations on $r$, part or all of parameter $x$ may have become redundant. Eliminating them results in a more efficient differentiated version $f'(\bar{x}, y, r)$, where $\bar{x}$ is $x$ or the part of $x$ that remains as a parameter.

**Example.** To illustrate the four-step derivation approach, we present a simple case of matrix multiplication as an example. Let $x$ be $\langle C, R \rangle$, where $C$ and $R$ are two lists of numbers. The function $mtxMul(C, R)$ treats $C$ as a column vector, $R$ as a row vector, and returns their outer product[15]. The resulting matrix is represented as a list of rows, where each row is a list of numbers. The auxiliary function $rowMul(e, R)$ multiplies an element $e$ with each element in a row $R$ and returns the row of products. The function definitions are:

$$\begin{array}{ll} mtxMul(C,R) = & \textbf{if } null(C) \textbf{ then } nil \\ & \textbf{else } cons(rowMul(car(C),R),\ mtxMul(cdr(C),R)) \\ rowMul(e,R) = & \textbf{if } null(R) \textbf{ then } nil \\ & \textbf{else } cons(e * car(R),\ rowMul(e, cdr(R))) \end{array} \tag{3}$$

Suppose $r$ is the cached result $mtxMul(C, R)$ and the change operation is defined as $\langle C, R\rangle \oplus y = \langle C, cons(y, R)\rangle$. We aim to compute $mtxMul(C, cons(y, R))$ more efficiently by making use of $r$.

Step 1 considers

$$mtxMul(C, cons(y, R)).$$

First, we unfold $mtxMul(C, cons(y, R))$ since its arguments involve both $x$ and $y$. We get

$$\begin{array}{l} \textbf{if } null(C) \textbf{ then } nil \\ \textbf{else } cons(rowMul(car(C), cons(y,R)),\ mtxMul(cdr(C), cons(y,R))). \end{array} \tag{4}$$

Then, we unfold $rowMul(car(C), cons(y, R))$ in (4) since its arguments also involve both $x$ and $y$. We get

$$\begin{array}{l} \textbf{if } null(C) \textbf{ then } nil \\ \textbf{else } cons(\textbf{if } null(cons(y,R)) \textbf{ then } nil \\ \qquad\qquad \textbf{else } cons(car(C) * car(cons(y,R)),\ rowMul(car(C), cdr(cons(y,R)))), \\ \qquad\quad mtxMul(cdr(C), cons(y,R))). \end{array} \tag{5}$$

Using the axioms of data types $null(cons(h, t)) = false$, $car(cons(h, t)) = h$, and $cdr(cons(h, t)) = t$ and specializing the false conditional expression in (5) to its false branch, we get

$$\begin{array}{l} \textbf{if } null(C) \textbf{ then } nil \\ \textbf{else } cons(cons(car(C) * y,\ rowMul(car(C), R)),\ mtxMul(cdr(C), cons(y,R))). \end{array} \tag{6}$$

The function application $mtxMul(cdr(C), cons(y, R))$ in (6) is not further unfolded since it is an instance of $mtxMul(C, cons(y, R))$, which has already been unfolded.

Step 2 defines $mtxMul^*(\langle C, R\rangle, y, r)$ based on (6). The expression $mtxMul(cdr(C), cons(y, R))$ in (6) is equivalent to $mtxMul(\langle cdr(C), R\rangle \oplus y)$. Replacing this application of $mtxMul$ with an application of $mtxMul^*$, we get

$$\begin{array}{l} \textbf{if } null(C) \textbf{ then } nil \\ \textbf{else } cons(cons(car(C) * y,\ rowMul(car(C), R)), \\ \qquad\qquad mtxMul^*(\langle cdr(C), R\rangle, y, mtxMul(cdr(C), R))), \end{array} \tag{7}$$

which is then used as the initial definition of $mtxMul^*(\langle C, R\rangle, y, r)$. Note the invariance that the third parameter of $mtxMul^*$ is the application of $mtxMul$ to the first parameter of $mtxMul^*$.

Step 3 aims to use the cached result $r$ to compute (7) more efficiently. In particular, in the false branch of the conditional expression, $null(C)$ is false. In this context, we specialize (3) and find that

$$mtxMul(C, R) = cons(rowMul(car(C), R),\ mtxMul(cdr(C), R)). \tag{8}$$

From the invariance $mtxMul(C, R) = r$, we have $cons(rowMul(car(C), R),\ mtxMul(cdr(C), R))$ $= r$, which implies,

$$rowMul(car(C), R) = car(r) \ \text{ and } \ mtxMul(cdr(C), R)) = cdr(r). \tag{9}$$

After substitutions in (7) using the last two equations, we get

$$\begin{array}{l} \textbf{if } null(C) \textbf{ then } nil \\ \textbf{else } cons(cons(car(C) * y,\ car(r)),\ mtxMul^*(\langle cdr(C), R\rangle, y, cdr(r))), \end{array} \tag{10}$$

which is a new definition of $mtxMul^*(\langle C, R\rangle, y, r)$.

Step 4 eliminates redundant code in the definition (10) of $mtxMul^*$. In particular, the parameter $R$ of $mtxMul^*(\langle C, R\rangle, y, r)$ is not used as data or for any control, except in the recursive call to $mtxMul^*$ itself as the corresponding argument. After eliminating it, we get a final definition

$$mtxMul'(C, y, r) = \begin{array}{l} \textbf{if } null(C) \textbf{ then } nil \\ \textbf{else } cons(cons(car(C) * y, car(r)),\ mtxMul'(cdr(C), y, cdr(r))). \end{array} \tag{11}$$

Therefore, given that $mtxMul(C, R) = r$, we have that $mtxMul'(C, y, r) = mtxMul^*(\langle C, R\rangle, y, r)$ $= mtxMul(C, cons(y, R))$. The function $mtxMul'$ is a differentiated version of $mtxMul$ for the given notion of input change. Note that it is not a complete differentiated version since $C$ can not be eliminated from (11).

**Generalized Partial Evaluation.** It is convenient and fruitful to view our derivation approach as an instance of *generalized partial evaluation* (GPE). GPE is a semantic-based program transformation: given an expression $e$ and an information set $I$ that describes partial knowledge about $e$, GPE produces a *residual expression* $e'$ by specializing $e$ using $I$.

The notion of GPE has appeared in [13] [42], where it is called generalized partial computation(GPC). It was originally proposed to overcome the drawbacks of conventional partial evaluation, which only specializes programs given static values for certain arguments. The key idea of GPE is to employ a theorem prover during the specialization to utilize information such as the logical structure of programs, algebraic properties of primitive functions, axioms for abstract data types, *etc.*

Expanding the computation of $f(x \oplus y)$ in Step 1 can be regarded as a special case of GPE, in which the expression $e$ is $f(v)$, and the information set states that $v$ is $x \oplus y$ and the cached result $f(x)$ is available. We call this use of GPE the *original generalized partial evaluation* (oGPE); the residual expression of the oGPE is called the *original residual expression.* In the matrix multiplication example above, (6) is the original residual expression.

The specialization of $f(x)$ in a specific context in Step 3 can also be regarded as a special case of GPE. We call this the *auxiliary generalized partial evaluation* (aGPE); a residual expression of the aGPE is called an *auxiliary residual expression.* For the example above, the specialization of $mtxMul(C, R)$ in the context where $null(C)$ is false is an aGPE that yields the right hand side of (8) as an auxiliary residual expression.

In [13] and [42], GPE transformation methods for a restricted form of language (u-form) and a lazy functional language are discussed, and small examples are given. In this paper, we see that combining the idea of GPE transformations with other special transformation techniques results in an effective transformational approach for deriving incremental programs.

# 4 Formalization of the Problem

First, we present the syntax of the simple language we use and give it a simple semantics. Then, the derivation problem is described in terms of the language.

## 4.1 Language

We consider a pure first-order functional language with the following syntax.

**Syntactic Domains**

| | |
|---|---|
| $Var$ | variables, ranged over by $v, v_i, x_i, y_i, r$ |
| $Con$ | constructors, ranged over by $c$ |
| $PrimFun$ | primitive functions, ranged over by $p$ |
| $Fun$ | functions, ranged over by $f, f_i, g, g_i$ |
| $Exp$ | expressions, ranged over by $e, e_i$ |

**Abstract Syntax**

For $e \in Exp$,

| | | |
|---|---|---|
| $e ::= v$ | $v \in Var$ | variable |
| $\mid \quad c(e_1, ..., e_n)$ | $c \in Con$ | constructor application |

$$
\begin{array}{llll}
| & p(e_1, ..., e_n) & p \in PrimFun & \text{primitive function application} \\
| & f(e_1, ..., e_n) & f \in Fun & \text{function application} \\
| & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & & \text{conditional expression} \\
| & \textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end} & & \text{binding expression}
\end{array}
$$

A program is a set $F$ of mutually recursive function definitions

$$\{f_i(v_1, ..., v_{n_i}) = e_i \mid i \in I_F\} \tag{12}$$

where $I_F$ is a finite index set satisfying $0 \in I_F$. The application of $f_0$ to input $x \equiv \langle x_1, ..., x_n \rangle$[1,2] is the expression to be evaluated in the context of these definitions. Note that, from this section on, $f_0$ refers to the function for which we aim to derive a differentiated version, whereas $f$ refers to an arbitrary function defined in $F$.

A simple strict semantics for expressions can be given by semantic function $\mathcal{E} : Exp \to Env \to Val$, where $Env$ and $Val$ are semantic domains. In particular, constructor applications and primitive function applications are strict. $Val$ is the set of values, including constants and constructions. $Con_i$ is the set of constructors of arity $i$. Integers and Boolean values ($T$ for true and $F$ for false) are included among the constants. $Env$ is a type-respecting mapping from variables to values. Formally, we give the semantic domains and the semantic equations below, where $D_\perp$ is the domain $D$ extended with a value $\perp$ that is below all elements in $D$ and $v \mapsto d$ means that the variable $v$ is bound to the value $d$. Sometimes, we use $[v \rightarrowtail d]$ as an abbreviation for $[v_1 \mapsto d_1, ..., v_n \mapsto d_n]$.

**Semantic Domains**

$$
\begin{array}{l}
Val = Constants + Constructions, \quad \text{ranged over by } d, d_i \\
\qquad \text{where } Constants = (Con_0)_\perp, \\
\qquad\qquad Constructions = (Con_1 \times Val)_\perp + (Con_2 \times Val \times Val)_\perp + ... + (Con_n \times Val^n)_\perp + ... \\
Env = Var \to Val, \quad \text{ranged over by } \rho
\end{array}
$$

**Semantic Equations**

$$
\begin{array}{ll}
\mathcal{E}[\![v]\!]\rho & = \rho(v) \\
\mathcal{E}[\![c(e_1, ..., e_n)]\!]\ \rho & = c(\mathcal{E}[\![e_1]\!]\rho, ..., \mathcal{E}[\![e_n]\!]\rho) \\
\mathcal{E}[\![p(e_1, ..., e_n)]\!]\ \rho & = p(\mathcal{E}[\![e_1]\!]\rho, ..., \mathcal{E}[\![e_n]\!]\rho) \\
\mathcal{E}[\![f(e_1, ..., e_n)]\!]\ \rho & = \mathcal{E}[\![e]\!]\ \rho[v_1 \mapsto d_1, ..., v_n \mapsto d_n], \text{ if } d_i \text{ is of the right type for } v_i \text{ of } f \\
 & = \perp, \text{ otherwise,} \\
 & \text{where } d_i = \mathcal{E}[\![e_i]\!]\rho, \text{ for } i = 1..n, \text{ and } f \text{ is defined by } f(v_1, ..., v_n) = e \\
\mathcal{E}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ \rho & = \mathcal{E}[\![e_2]\!]\rho, \text{ if } d = T \\
 & = \mathcal{E}[\![e_3]\!]\rho, \text{ if } d = F \\
 & = \perp, \text{ otherwise,} \\
 & \text{where } d = \mathcal{E}[\![e_1]\!]\rho \\
\mathcal{E}[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!]\ \rho & = \mathcal{E}[\![e_2]\!]\ \rho[v \mapsto d], \text{ if } d \text{ of the right type for } v,
\end{array}
$$

[1] In the following presentation, we use '$\equiv$' for identity.

[2] For simplicity, the arity $n_0$ of function $f_0$ is denoted by $n$.

$= \perp$, otherwise,
where $d = \mathcal{E}[\![e_1]\!]\rho$

### 4.2 Description of the Problem

We are given a set $F$ of function definitions, as in (12), and an *input change operation* $\oplus$. The operation $\oplus$ describes the change to the arguments of the function $f_0$. It combines a tuple of old arguments $x \equiv \langle x_1, ..., x_n \rangle$ and a tuple of values $y \equiv \langle y_1, ..., y_m \rangle$ to form a tuple of new arguments $x' \equiv \langle x'_1, ..., x'_n \rangle$, i.e., $x' = x \oplus y$. The operation $\oplus$ can be regarded as a tuple of $n$ separate *single change operations* $\oplus_1, ..., \oplus_n$ for $x'_1, ..., x'_n$ respectively, such that $x'_i = x \oplus_i y$ for each $i = 1, ..., n$. Each of the single change operations $\oplus_i$ is defined in such a way that $x'_i$ is computed as some function $g_i$ of parameters $x_j$'s and $y_k$'s. Usually, $g_i$ is a constructor application or a primitive function application. The most typical $y$ and $\oplus$ are the tuple $y \equiv \langle y_1, ..., y_n \rangle$ and the operation $\oplus$ such that there exists a binary function $g_i$ of $x_i$ and $y_i$, such that $x'_i = g_i(x_i, y_i)$, for each $i = 1, ..., n$.

Note that in order to keep our language simple, we do not include tuples as expressions. However, tuples are used in the presentation as a notational convenience. In particular, if $x \equiv \langle x_1, ..., x_n \rangle$, $y \equiv \langle y_1, ..., y_m \rangle$, $\oplus \equiv \langle \oplus_1, ..., \oplus_n \rangle$, and $x \oplus_i y$ is defined using a function $g_i$ of $x_j$'s and $y_k$'s, then $f_0(x)$ is an abbreviation for $f_0(x_1, ..., x_n)$, $f_0^*(x, y, f_0(x))$ is an abbreviation for $f_0^*(x_1, ..., x_n, y_1, ..., y_m, f_0(x_1, ..., x_n))$, and $x \oplus y$ is an abbreviation for $\langle x \oplus_1 y, ..., x \oplus_n y \rangle$, where $x \oplus_i y$ is an abbreviation for $g_i(x_1, ..., x_n, y_1, ..., y_m)$. These abbreviations are used not only for tuples of variables, but also for tuples of expressions.

**Time Model.** We are only interested in using the cached result if we can save time by doing so. Accordingly, we must have a model that describes the time needed to compute expressions. We use $\mathcal{T}(e)$ to denote the time needed to compute expression $e$. If $r$ is the only free variable in an expression $e$, then $\mathcal{T}(e)$ is described as "the time needed to retrieve the value of $e$ from cached result $r$". If $e \equiv \langle e_1, ..., e_n \rangle$, we use $\mathcal{T}(e)$ to denote the sum of the $\mathcal{T}(e_i)$'s. The function $\mathcal{T}$ can be obtained from standard constructions[44, 32, 35]. Note that different primitive functions may take different amounts of time to evaluate.

Let $e(v_1, ..., v_n)$ denote an expression $e$ in which $v_1, ..., v_n$ are all the possible free variables. In general, given two expressions $e_1(v_{11}, ..., v_{1n_1})$ and $e_2(v_{21}, ..., v_{2n_2})$, it is not decidable whether $e_2(v_{21}, ..., v_{2n_2})$ computes faster than $e_1(v_{11}, ..., v_{1n_1})$ for given values of $v_{1i}$'s and $v_{2j}$'s. Therefore, we say that $\mathcal{T}(e_2(v_{21}, ..., v_{2n_2})) \leq \mathcal{T}(e_1(v_{11}, ..., v_{1n_1}))$ for given values of $v_{1i}$'s and $v_{2j}$'s if we can *effectively* confirm the inequality using heuristics for function $\mathcal{T}$. We write

$$t(e_2(v_{21},...,v_{2n_2})) \leq_{P(v_{11},...,v_{1n_1},v_{21},...,v_{2n_2})} t(e_1(v_{11},...,v_{1n_1})) \tag{13}$$

for some predicate $P$ to denote that we can effectively decide that there exists a constant $k$ such that

$$\begin{aligned}\forall d_{v_{11}},...,d_{v_{1n_1}},d_{v_{21}},...,d_{v_{2n_2}}(\ &\mathcal{E}[\![P(v_{11},...,v_{1n_1},v_{21},...,v_{2n_2})]\!]\ [v_1 \rightarrowtail d_{v_1}, v_2 \rightarrowtail d_{v_2}] = T \\ &\Longrightarrow \mathcal{T}(e_2(d_{v_{21}},...,d_{v_{2n_2}})) \leq k \cdot \mathcal{T}(e_1(d_{v_{11}},...,d_{v_{1n_1}}))\ )\end{aligned} \tag{14}$$

and abbreviate (13) as $t(e_2) \leq_P t(e_1)$ when no confusion arises. In particular, we use $t(e_2) \leqq_P t(e_1)$ to denote that (14) holds for $k = 1$. The subscript $P$ is omitted if the inequality in (14) holds unconditionally. Note that $t(e_1) \leq_P t(e_2)$ or $t(e_1) \leqq_P t(e_2)$ is just notation as $t$ itself is not defined.

As a matter of fact, we do not need to know the exact time needed to evaluate an expression. Instead, we only care about comparing the times needed to evaluate two expressions. Therefore, we can use heuristics to compare times. For example, if $\mathcal{T}(e) \leq \mathcal{T}(e_1) + min\{\mathcal{T}(e_2), \mathcal{T}(e_3)\}$, then $\mathcal{T}(e) \leq \mathcal{T}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)$. If $\mathcal{T}(e_1) \leq \mathcal{T}(e_2)$, then $\mathcal{T}(p(e_1)) \leq \mathcal{T}(p(e_2))$ for any applicable primitive function $p$. In particular, if $e_1$ is a subexpression of $e_2$, then $\mathcal{T}(e_1) \leq \mathcal{T}(e_2)$. For example, $\mathcal{T}(e_1) \leq \mathcal{T}(car(cons(e_1, e_2)))$. We can also make use of the transitivities of various comparison relations.

Given $F$, $\oplus$, and the time model $\mathcal{T}$, we aim to apply a series of transformations to obtain a finite set $F'$ of function definitions $\{f'_i(v_1,...,v_{n'_i}) = e'_i \mid i \in I_{F'}\}$, where $0 \in I_{F'}$ and $f'_0$ is a differentiated version of $f_0$ under $\oplus$.

## 5 Formalization of the Transformations for the Derivation

In order make the formalization of the transformational approach tractable, we break it into four steps:

| step | description | transformation |
|---|---|---|
| 1 | expanding $f_0(x \oplus y)$ to separate computations on $x$ and $y$ | $\mathcal{G}_o$ |
| 2 | introducing the cache parameter $r$ | $\mathcal{D}_t$ |
| 3 | using the cache parameter | $\mathcal{S}_t$ |
| 4 | eliminating redundant code | standard optimizations |

The definition of a differentiated version $f_0^*$ of $f_0$ is obtained by transforming $f_0(x \oplus y)$ with $\mathcal{G}_o$, $\mathcal{D}_t$, and $\mathcal{S}_t$ in turn. The final differentiated version $f'_0$ is obtained by applying standard optimizations, especially redundant code elimination, on $f_0^*$ and the other function definitions in the set $F$. In the following, we discuss each step separately and argue (a) that together they preserve the semantics of the program, and (b) that computations using the derived programs are asymptotically at least as fast as using the original programs. An example using the transformations is given in Section 6.

Since this formalization breaks the transformation into separate steps, it has some obvious limitations and inefficiencies. For a characterization of this simple formalization and ideas for a more general approach, see Section 7.

### 5.1 Information Sets and Underlying Logic

Section 3 introduced the GPE view of our derivation approach. GPE specializes an expression $e$ with respect to an information set $I$. We use *Info* to denote the domain of information sets. For any $I \in$ *Info*, $I$ is a set of equations whose conjunction represents the information with which an expression is to be specialized. An equation is a pair of expressions and has type $Exp \times Exp$, where the two expressions are related by the Boolean primitive equality function. In this paper, we use $e \leftrightarrow e'$ to denote that $e$ equals $e'$ for two expressions $e$ and $e'$. Usually, an information set is collected from the context of an expression. For example, let $e$ be the expression

$$\textbf{let } v = p(v_1) \textbf{ in if } v < v_2 \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ end}.$$

The information set for subexpression $e_2$ in $e$ is $\{v \leftrightarrow p(v_1),\ v < v_2 \leftrightarrow F\}$.

The key idea of GPE is to use a theorem prover to make inferences based on its logic and the information collected about a program. We are only interested in logics that contain axioms, inference rules, and basic formulas that are true for the language we are using. For any information set $I$, let $e_I$ be a Boolean-valued expression in our language that represents the information of the set $I$. A logic is called an *underlying logic* [13] for our language if it is *compatible* with the semantics of our language, where compatibility is defined as follows:

**Definition 5.1** *A logic $L$ is* compatible *with the semantics $\mathcal{E}$ of our language if and only if, for any two information sets $I$ and $I'$ such that the equations in $I'$ are provable from those in $I$ based on the logic $L$, we have $\forall d_v,\ \mathcal{E}[\![e_I]\!]\ [v \rightarrowtail d_v] = T \implies \mathcal{E}[\![e_{I'}]\!]\ [v \rightarrowtail d_v] = T$, where $v_1, ..., v_k$ are all the free variables in $e_I$ and $e_{I'}$.*

In the transformations below, we assume that $L_0$ is some fixed underlying logic of our language. In principle, $L_0$ can be any underlying logic, depending on the power we desire for making inferences about our programs. In this paper, we assume that a theorem prover based on the logic $L_0$ is available, and we write $e \leftrightarrow_I^* e'$ to denote that a finite proof that $e$ equals $e'$ can be found by the theorem prover using equations in the set $I$.

### 5.2 Step 1: Separating $x$ and $y$

Step 1 specializes the function $f_0$ for arguments of the form $x \oplus y$. The goal of this step is to expand computations of $f_0(x \oplus y)$ into computations on $x$ and $y$ separately in the hope that computations on $x$ alone can be avoided by making use of the cached result $f_0(x)$.

Step 1 unfolds function applications whose arguments involve computations on both $x$ and $y$. Conversely, function applications whose arguments depend purely on $x$ or $y$ are not unfolded. The intuition behind this strategy is as follows: First, by unfolding and simplifying function applications on arguments involving both $x$ and $y$, we can hope to expose computations on $x$ and on $y$ separately. Secondly, computations purely related to $y$ are new and need to be carried out anyway. The only purpose in unfolding such function applications would be to optimize them, but incrementalization, not optimization, is the subject of this paper. Thirdly, the intention in unfolding function applications that depend only on $x$ would be either to enable optimizations of such computations or to help equate them with computations performed by $f_0(x)$. The former is not the subject of this paper, as just mentioned. The latter is actually a theorem proving problem, which is not the subject of this paper either. There is a fourth category of function applications, namely those that depend on neither $x$ nor $y$. Such computations allow a straightforward optimization, i.e., evaluation by unfolding and simplification. However, this is a potential source of nontermination. We may wish to omit such optimizations, as will be discussed later for the unfolding rule. In short, our main concern is incrementality of $f_0$ with respect to the new input $x \oplus y$.

Step 1 is implemented by the transformation $\mathcal{G}_o$ defined in Figure 1, where the type of $\mathcal{G}_o$ is $Exp \rightarrow Info \rightarrow Exp$. Given an expression $e \in Exp$ and an information set $I \in Info$, $\mathcal{G}_o$ specializes $e$ with respect to $I$. By applying $\mathcal{G}_o$ to the expression $f_0(x \oplus y)$ and the null information set $\emptyset$, we obtain expression $e^o$:

$$e^o \equiv \mathcal{G}_o[\![f_0(x \oplus y)]\!]\ \emptyset. \tag{15}$$

#### 5.2.1 Definition of $\mathcal{G}_o$

The body of function $\mathcal{G}_o$ consists of a set of cases conditioned on the structure of its argument expression. The first column in Figure 1 gives a name for each case, where a description of the case is encoded in the name:

| name | top-level construct |
|---|---|
| $v$ | variable |
| $c$ | constructor application |
| $p$ | primitive function application |
| $f$ | function application |
| $if$ | **if**-expression |
| $let$ | **let**-expression |

| Name | Transformation | Condition |
| --- | --- | --- |
| $v$ | $\mathcal{G}_o[\![v]\!]\ I$ | |
| $v_c$ | $= c$ | if $v \leftrightarrow_I^* c$ |
| $v_n$ | $= v$ | otherwise |
| $c$ | $\mathcal{G}_o[\![c(e_1,...,e_n)]\!]\ I$ | |
| $c_i$ | $= \mathcal{G}_o[\![c(e_1,...,e_{i-1},\mathcal{G}_o[\![e_i]\!]I,e_{i+1},...,e_n)]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, but $e_i$ is reducible |
| $c_{i-if}$ | $= \mathcal{G}_o[\![$**if** $e_1'$ **then** $c(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)$ **else** $c(e_1,...,e_{i-1},e_3',e_{i+1},...,e_n)]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, $e_i$ is irreducible, and $e_i$ is **if** $e_1'$ **then** $e_2'$ **else** $e_3'$ |
| $c_{i-let}$ | $= \mathcal{G}_o[\![$**let** $v = e_1'$ **in** $c(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)$ **end**$]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, $e_i$ is irreducible, and $e_i$ is **let** $v = e_1'$ **in** $e_2'$ **end** |
| $c_n$ | $= c(e_1,...,e_n)$ | otherwise |
| $p$ | $\mathcal{G}_o[\![p(e_1,...,e_n)]\!]\ I$ | |
| $p_i$ | $= \mathcal{G}_o[\![p(e_1,...,e_{i-1},\mathcal{G}_o[\![e_i]\!]I,e_{i+1},...,e_n)]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, but $e_i$ is reducible |
| $p_{i-if}$ | $= \mathcal{G}_o[\![$**if** $e_1'$ **then** $p(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)$ **else** $p(e_1,...,e_{i-1},e_3',e_{i+1},...,e_n)]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, $e_i$ is irreducible, and $e_i$ is **if** $e_1'$ **then** $e_2'$ **else** $e_3'$ |
| $p_{i-let}$ | $= \mathcal{G}_o[\![$**let** $v = e_1'$ **in** $p(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)$ **end**$]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, $e_i$ is irreducible, and $e_i$ is **let** $v = e_1'$ **in** $e_2'$ **end** |
| $p_e$ | $= \mathcal{G}_o[\![Simp[\![p(e_1,...,e_n)]\!]I]\!]\ I$ | if $e_1,...,e_n$ are irreducible, not **if** or **let**, but $p(e_1,...,e_n)$ can be simplified under $I$ |
| $p_n$ | $= p(e_1,...,e_n)$ | otherwise |
| $f$ | $\mathcal{G}_o[\![f(e_1,...,e_n)]\!]\ I$ | |
| $f_i$ | $= \mathcal{G}_o[\![f(e_1,...,e_{i-1},\mathcal{G}_o[\![e_i]\!]I,e_{i+1},...,e_n)]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, but $e_i$ is reducible |
| $f_{i-if}$ | $= \mathcal{G}_o[\![$**if** $e_1'$ **then** $f(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)$ **else** $f(e_1,...,e_{i-1},e_3',e_{i+1},...,e_n)]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, $e_i$ is irreducible, and $e_i$ is **if** $e_1'$ **then** $e_2'$ **else** $e_3'$ |
| $f_{i-let}$ | $= \mathcal{G}_o[\![$**let** $v = e_1'$ **in** $f(e_1,...,e_{i-1},e_2',e_{i+1},...,e_n)$ **end**$]\!]\ I$ | if $e_1,...,e_{i-1}$ are irreducible, not **if** or **let**, $e_i$ is irreducible, and $e_i$ is **let** $v = e_1'$ **in** $e_2'$ **end** |
| $f_e$ | $= \mathcal{G}_o[\![e[v_1 := e_1,...,v_n := e_n]]\!]\ I$ where $f$ is defined as $f(v_1,...,v_n) = e$ | if $e_1,...,e_n$ are irreducible, not **if** or **let**, but the argument pattern is favorable and is not in the list $l_f$ Also, put the argument pattern in the list $l_f$. |
| $f_n$ | $= f(e_1,...,e_n)$ | otherwise |
| $if$ | $\mathcal{G}_o[\![$**if** $e_1$ **then** $e_2$ **else** $e_3]\!]\ I$ | |
| $if_1$ | $= \mathcal{G}_o[\![$**if** $(\mathcal{G}_o[\![e_1]\!]I)$ **then** $e_2$ **else** $e_3]\!]\ I$ | if $e_1$ is reducible |
| $if_{1-if}$ | $= \mathcal{G}_o[\![$**if** $e_1'$ **then** (**if** $e_2'$ **then** $e_2$ **else** $e_3$) **else** (**if** $e_3'$ **then** $e_2$ **else** $e_3$)$]\!]\ I$ | if $e_1$ is irreducible, and $e_1$ is **if** $e_1'$ **then** $e_2'$ **else** $e_3'$ |
| $if_{1-let}$ | $= \mathcal{G}_o[\![$**let** $v = e_1'$ **in** (**if** $e_2'$ **then** $e_2$ **else** $e_3$) **end**$]\!]\ I$ | if $e_1$ is irreducible, and $e_1$ is **let** $v = e_1'$ **in** $e_2'$ **end** |
| $if_2$ | $= \mathcal{G}_o[\![e_2]\!]\ I$ | if $e_1$ is irreducible, not **if** or **let**, and $e_1 \leftrightarrow_I^* T$ |
| $if_3$ | $= \mathcal{G}_o[\![e_3]\!]\ I$ | if $e_1$ is irreducible, not **if** or **let**, and $e_1 \leftrightarrow_I^* F$ |
| $if_{23}$ | $=$ **if** $e_1$ **then** $\mathcal{G}_o[\![e_2]\!]\ (I \cup \{e_1 \leftrightarrow T\})$ **else** $\mathcal{G}_o[\![e_3]\!]\ (I \cup \{e_1 \leftrightarrow F\})$ | otherwise |
| $let$ | $\mathcal{G}_o[\![$**let** $v = e_1$ **in** $e_2$ **end**$]\!]\ I$ | |
| $let_1$ | $= \mathcal{G}_o[\![$**let** $v = (\mathcal{G}_o[\![e_1]\!]I)$ **in** $e_2$ **end**$]\!]\ I$ | if $e_1$ is reducible |
| $let_{1-if}$ | $= \mathcal{G}_o[\![$**if** $e_1'$ **then** (**let** $v = e_2'$ **in** $e_2$ **end**) **else** (**let** $v = e_3'$ **in** $e_2$ **end**)$]\!]\ I$ | if $e_1$ is irreducible, and $e_1$ is **if** $e_1'$ **then** $e_2'$ **else** $e_3'$ |
| $let_{1-let}$ | $= \mathcal{G}_o[\![$**let** $v' = e_1'$ **in** (**let** $v = e_2'$ **in** $e_2$ **end**) **end**$]\!]\ I$ | if $e_1$ is irreducible, and $e_1$ is **let** $v' = e_1'$ **in** $e_2'$ **end** |
| $let_2$ | $= \mathcal{G}_o[\![e_2[v := v_1]]\!]\ I$ | if $e_1$ is irreducible, not **if** or **let**, and $v_1 \leftrightarrow_I^* e_1$ |
| $let_2'$ | $= \mathcal{G}_o[\![$**let** $v = e_1$ **in** $\mathcal{G}_o[\![e_2]\!]\ (I \cup \{v \leftrightarrow e_1\})$ **end**$]\!]\ I$ | if $e_1$ is irreducible, not **if** or **let**, not $v_1 \leftrightarrow_I^* e_1$, but $e_2$ is reducible |
| $let_e$ | $= \mathcal{G}_o[\![e_2[v := e_1]]\!]\ I$ | if $e_1$ is irreducible, not **if** or **let**, not $v_1 \leftrightarrow_I^* e_1$, $e_2$ is irreducible, but the **let** expression is unfoldable |
| $let_n$ | $=$ **let** $v = e_1$ **in** $e_2$ **end** | otherwise |

Figure 1: Definition of $\mathcal{G}_o$

Subscript $i$ in a name indicates that subexpression $e_i$ within the named construct is being reduced. Subscript *i-if* or *i-let* indicates that the subexpression being reduced is an **if** or a **let** expression. Subscript $e$ indicates that an expression itself is reduced after all of its subexpressions have been reduced. Subscript $n$ indicates that an expression is not further reducible. The last column gives side conditions for the reduction rules.

The presentation of $\mathcal{G}_o$ is simplified by omitting detailed control structures that sequence $\mathcal{G}_o$ through its argument subexpressions. Thus, we just present the case of $\mathcal{G}_o$ working on subexpression $e_i$ of the top-level construct and condition it on the *irreducibility* of subexpressions $e_1$ through $e_{i-1}$. Operationally, when we say that subexpressions $e_1$ through $e_{i-1}$ are irreducible, we mean that they are the result of having already applied $\mathcal{G}_o$ for subexpressions at those positions. We say that an expression $e$ is *reducible* if $e$ is not irreducible.

The meaning of $v$ rules are straightforward. A variable that can be shown to be equal to a constant $c$ under the assumptions of the given information set is replaced by $c$.

For a constructor application, $\mathcal{G}_o$ recursively reduces each argument in turn. If a reduced argument expression is an **if**-expression or a **let**-expression, then the **if** or **let** is lifted out of the constructor.

On a primitive function application, $\mathcal{G}_o$ first reduces each argument and, as in the case of a constructor application, lifts any **if** or **let** expressions. Then it uses the properties of the primitive function to simplify the application. We say that a primitive function application $p(e_1, ..., e_n)$ can be *simplified (to $e'$) under $I$* if we can effectively find an expression $e'$ such that $p(e_1, ..., e_n) \leftrightarrow_I^* e'$ and $t(e') \leqq_{e_I} t(p(e_1, ..., e_n))$. We define $Simp[\![p(e_1, ..., e_n)]\!]I$ to be an expression $e'$ if $p(e_1, ..., e_n)$ can be simplified to $e'$ under $I$ but $e'$ can not be further simplified under $I$.

On a function application, $\mathcal{G}_o$ also first reduces each argument and, if necessary, lifts an **if** or a **let** as in the case of a constructor application. Then, it considers whether the function application needs to be unfolded using rule $f_e$. As this is the key point for $\mathcal{G}_o$, it will be discussed in detail below. Note that since rule $f_e$ involves function unfolding, in order to guarantee the consistency of information sets, we assume that each variable in a **let** binding is renamed by the unfolding so that it is distinct from all other variables currently used.

For an **if**-expression, $\mathcal{G}_o$ first reduces the Boolean expression. If the reduced Boolean expression is another **if** or **let**-expression, then this **if** or **let** is lifted out of the original top-level **if**-expression. Suppose the Boolean expression is reduced already and is not an **if** or a **let**-expression. Then if the Boolean expression can be proved to be true (false) under the assumptions of the given information set, the **if** structure simplifies to the true (false) branch alone, which is then reduced. Otherwise, if the Boolean expression can not be proved to be true or false, the Boolean expression is left as a residual and $\mathcal{G}_o$ reduces both the true and false branches. In this case, the assumption that the Boolean expression equals true (false) is added into the information set for the true (false) branch of the **if**-expression.

For a **let**-expression, $\mathcal{G}_o$ is similar to the case of an **if**-expression. It first reduces the binding expression and, if necessary, lifts **if** and **let**-expressions. Suppose the binding expression is reduced already and is not an **if** or a **let**-expression. Then if a variable $v_1$ can be proved to be equal to the binding expression, then the **let** structure simplifies to its body alone (with variable $v$ replaced by $v_1$), which is then reduced. Otherwise, $\mathcal{G}_o$ reduces the body of the **let**-expression with the information set enlarged by the binding; afterwards $\mathcal{G}_o$ unfolds the **let**-expression if the **let**-expression is unfoldable. We say that a **let**-expression '**let** $v = e_1$ **in** $e_2$ **end**' is unfoldable if there is no duplicated computation of $e_1$ in $e_2[v := e_1]$, i.e., $e_1$ is a variable or $v$ appears at most once on every execution path where $v$ is needed in $e_2$. Recall that we assume that variables in **let** bindings are renamed by function unfoldings, which guarantees the consistency of information sets.

### 5.2.2 Unfolding of Function Applications

We now return to the question of which function applications to unfold. Recall that we are interested in unfolding function applications that depend on both the old input $x$ and the change parameter $y$.

Given an expression $e$, a single change operation $\oplus_i$, and an information set $I$, we say that $e$ is *favorably decomposable (under I)* into $e_x \oplus_i e_y$ if $e$ depends on both $x$ and $y$ and we can find two tuples of expressions, $e_x$ and $e_y$, such that

1) $e_x$ depends on $x$ but not on $y$,
2) $e_y$ depends on $y$ and possibly on $x$, and
3) $e \leftrightarrow_I^* e_x \oplus_i e_y$.

As a trivial but important example, if $e \equiv e_1 \oplus_i e_2$, where $e_1$ depends on $x$ but not on $y$ and $e_2$ depends on $y$, then $e$ is favorably decomposable into $e_1 \oplus_i e_2$ under any information set $I$.

Given the input change operation $\oplus \equiv \langle \oplus_1, ..., \oplus_n \rangle$, we say that an expression $e$ is *most favorably decomposable* into $e_x \oplus_i e_y$ if $e$ is favorably decomposable into $e_x \oplus_i e_y$ for some $i$ and $t(e_x \oplus_i e_y) \leq t(e'_x \oplus_j e'_y)$ for all favorable decompositions $e'_x \oplus_j e'_y$ of $e$ for $j = 1, ..., n$.

In order to avoid unlimited unfoldings, we associate an *argument pattern* with each function application and unfold applications of a function at most once for each of its *favorable* argument patterns. Consider a function application $f(e_1, ..., e_n)$. Suppose that $e_1, ..., e_n$ have been reduced and are not **if** or **let**-expressions. We say that $f(e_1, ..., e_n)$ *has a favorable argument pattern* if at least one of the $e_i$'s either has a most favorable decomposition or depends on neither $x$ nor $y$. Formally, we associate an *argument pattern* $\langle pat_1, ..., pat_n\rangle$ with the function application $f(e_1, ..., e_n)$, where

$$pat_i = \begin{cases} [j] & \text{if } e_i \text{ is most favorably decomposable into } e_{ix} \oplus_j e_{iy} \text{ for some } j \\ e_i & \text{if } e_i \text{ depends on neither } x \text{ nor } y \\ [0] & \text{otherwise} \end{cases}$$

We say that the argument pattern of a function application is *favorable* if not all of its components are [0].

For each function $f$, we keep a list $l_f$ of the argument patterns for which $f$ has already been unfolded. The unfolding rule $f_e$ of $\mathcal{G}_o$ unfolds an application of $f$ if and only if its argument pattern is favorable and does not occur in the list $l_f$. Note that this is merely one of many possible unfolding strategies; a discussion of unfolding strategies appears in Section 7.

The unfolding rule is the only possible source of nontermination. We can guarantee that $\mathcal{G}_o$ always terminates by forbidding the unfolding of function applications caused by arguments that depend on neither $x$ nor $y$, i.e., we could define $pat_i$ to be $[j]$ if $e_i$ is most favorably decomposable into $e_{ix} \oplus_j e_{iy}$ and [0] otherwise. For this notion of favorable argument pattern, each function has only a fixed number of favorable argument patterns, so termination is assured. This choice omits the class of straightforward optimization — evaluation by unfolding and simplification. Of course, allowing such unfoldings coupled with some heuristics to prevent divergence could be an effective implementation strategy.

We can prove the following proposition for $\mathcal{G}_o$ by an induction on the structure of expressions, following the $\mathcal{G}_o$ rules, and making use of the fact that the underlying logic is compatible with the semantics of our language.

**Proposition 5.2** *For any expression $e$ and information set $I$ of $e$, let $v_1, ..., v_k$ be all the free variables in $e$ and $e_I$. If $\mathcal{G}_o[\![e]\!]I$ terminates, then, $\forall d_v$, if $\mathcal{E}[\![e_I]\!]\,[v \rightarrowtail d_v] = T$, then $\forall d$, $\mathcal{E}[\![e]\!]\,[v \rightarrowtail d_v] = d$ $\Longrightarrow \mathcal{E}[\![\mathcal{G}_o[\![e]\!]I]\!]\,[v \rightarrowtail d_v] = d$, and $t(\mathcal{G}_o[\![e]\!]I) \leq t(e)$.*

Therefore, if the application of $\mathcal{G}_o$ on $f_0(x \oplus y)$ terminates and we get $e^o \equiv \mathcal{G}_o[\![f_0(x \oplus y)]\!]\,\emptyset$, then we have the following corollary.

**Corollary 5.3** *If $\mathcal{G}_o[\![f_0(x \oplus y)]\!]\,\emptyset$ terminates with $e^o \equiv \mathcal{G}_o[\![f_0(x \oplus y)]\!]\,\emptyset$, then we have, $\forall d_x, d_y, d$,*

$$\mathcal{E}[\![f_0(x \oplus y)]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y] = d \implies \mathcal{E}[\![e^o]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y] = d,$$

*and $t(e^o) \leq t(f_0(x \oplus y))$.*

### 5.3 Step 2: Introducing the Cached-Result Parameter $r$

Step 2 defines the function $f_0^*$ such that $f_0^*(x, y, r)$ computes $f_0(x \oplus y)$ when $r$ is the cached result $f_0(x)$. The extra parameter $r$ is introduced as an explicit name for the cached result $f_0(x)$. Our motivation for introducing $r$ is so that, in a later step, some computations performed by $f_0(x \oplus y)$ may be replaced by expressions involving the cached result $r$, provided that such expressions compute the same values and are at least as fast as doing the original computations.

We define

$$f_0^*(x, y, r) = e^*, \tag{16}$$

where $e^*$ is a transformed version of $e^o$. The transformation of $e^o$ involves replacing some calls $f_0(e)$ with recursive calls $f_0^*(e_x, e_y, f_0(e_x))$, where $e_x$ and $e_y$ are a *definition candidate pair* for $e$, defined below, such that $e_x \oplus e_y$ is equivalent to $e$. Note that this transformation preserves the invariant that the third argument to $f_0^*$ is always $f_0$ applied to the first argument of $f_0^*$. Also note that, in the body of $f_0^*$ as defined in this step, parameter $r$ is not used in any substantive way; it will be used by Step 3 to replace, in the body of $f_0^*$, some computations on $x$ by expressions involving $r$.

The transformation of $e^o$ into $e^*$ is implemented by $\mathcal{D}_t$, which has type $Exp \to Info \to Exp$. Expression $e^*$, the body of function $f_0^*$, is obtained by applying $\mathcal{D}_t$ to $e^o$ and the null information set $\emptyset$:

$$e^* \equiv \mathcal{D}_t[\![e^o]\!]\ \emptyset. \tag{17}$$

Figure 2 defines $\mathcal{D}_t$. The names of rules have the same meaning as for the $\mathcal{G}_o$ transformation, with the addition that the subscript $s$ indicates substitution of a function application.

All rules other than those dealing with function applications are straightforward. They just collect and pass on information sets for the expressions being transformed. If there is no function call to $f_0$ in $e^o$, then $e^* \equiv e^o$.

Consider a (recursive) function application $f_0(e_1, ..., e_n)$ in the expression $e^o$ with the collected information $I$. Let $e \equiv \langle e_1, ..., e_n \rangle$. If there exist a pair of tuples $e_x$ and $e_y$ such that $e_i$ is favorably decomposable into $e_x \oplus_i e_y$ under $I$ for each $i = 1, ..., n$ and $t(e_x \oplus e_y) \leqq t(e)$, then we call $\langle e_x, e_y \rangle$ a *definition candidate pair* for $f_0(e)$, and replace the occurrence of $f_0(e)$ with $f_0^*(e_x, e_y, f_0(e_x))$.

Function $f_0^*$ is not necessarily a differentiated version of $f_0$ because, although transformation $\mathcal{D}_t$ preserves semantics, it may actually result in a slower program. To compare the time of $e^*$ and $e^o$, consider a function application $f_0^*(e_x, e_y, f_0(e_x))$ in $e^*$ that replaces function application $f_0(e)$ in $e^o$. The fact that $\langle e_x, e_y \rangle$ is a definition candidate pair for $f_0(e)$ guarantees that computing

| Name | Transformation | | Condition |
|---|---|---|---|
| $v$ | $\mathcal{D}_t[\![v]\!]\ I$ | $= v$ | |
| $c$ | $\mathcal{D}_t[\![c(e_1, ..., e_n)]\!]\ I$ | $= c(\mathcal{D}_t[\![e_1]\!]I, ..., \mathcal{D}_t[\![e_n]\!]I)$ | |
| $p$ | $\mathcal{D}_t[\![p(e_1, ..., e_n)]\!]\ I$ | $= p(\mathcal{D}_t[\![e_1]\!]I, ..., \mathcal{D}_t[\![e_n]\!]I)$ | |
| $f_s$ | $\mathcal{D}_t[\![f(e_1, ..., e_n)]\!]\ I$ | $= f_0^*(e_x, e_y, f(e_x))$ | if $f \equiv f_0$ and $\langle e_x, e_y \rangle$ is a definition candidate pair for $f(e_1, ..., e_n)$ |
| $f_n$ | | $= f(\mathcal{D}_t[\![e_1]\!]I, ..., \mathcal{D}_t[\![e_n]\!]I)$ | otherwise |
| $if$ | $\mathcal{D}_t[\![$**if** $e_1$ **then** $e_2$ **else** $e_3]\!]\ I$ | $=$ **if** $\mathcal{D}_t[\![e_1]\!]I$ **then** $\mathcal{D}_t[\![e_2]\!]\ (I \cup \{e_1 \leftrightarrow T\})$ **else** $\mathcal{D}_t[\![e_3]\!]\ (I \cup \{e_1 \leftrightarrow F\})$ | |
| $let$ | $\mathcal{D}_t[\![$**let** $v = e_1$ **in** $e_2$ **end**$]\!]\ I$ | $=$ **let** $v = \mathcal{D}_t[\![e_1]\!]I$ **in** $\mathcal{D}_t[\![e_2]\!]\ (I \cup \{v \leftrightarrow e_1\})$ **end** | |

Figure 2: Definition of $\mathcal{D}_t$

$e_x$ and $e_y$ is at least as fast as computing $e$. However, $f_0(e_x)$ may need a substantial amount of time to compute and hence transformation $\mathcal{D}_t$ produces a slower program. This apparent step backwards is remedied in Step 3, which restores $f_0^*(e_x, e_y, f_0(e_x))$ to $f_0(e)$ unless it can be replaced by $f_0^*(e_x, e_y, e_r)$, where $e_r$ computes $f_0(e_x)$ rapidly using the cached value $r$ and $f_0^*(e_x, e_y, f_0(e_x))$ can compute at least as fast as $f_0(e)$ by making use of $e_r$ .

Note that, among various heuristics that can be used to find a definition candidate pair for $f(e)$ under information $I$, there are two special cases that are most useful:

1. If there exists a right-reverse operation $\ominus$ of $\oplus$ such that $y = z \ominus x$ iff $x \oplus y = z$, and $e \ominus x$ can be simplified element-wise under $I$ to form $e_y$ such that $t(x \oplus e_y) \leqq t(e)$, then $\langle x, e_y \rangle$ is a definition candidate pair for $f_0(e)$. In this case, transformation $\mathcal{D}_t$ may replace $f_0(e)$ by $f_0^*(x, e_y, f_0(x))$, and Step 3 can subsequently replace $f_0(x)$ by the cached result $r$.

2. If there exists a left-reverse operation $\ominus$ of $\oplus$ such that $x = z \ominus y$ iff $x \oplus y = z$, and $e \ominus y$ can be simplified element-wise under $I$ to form $e_x$ such that $e_x$ does not depend on $y$ and $t(e_x \oplus y) \leqq t(e)$, then $\langle e_x, y \rangle$ is a definition candidate pair for $f_0(e)$. In this case, transformation $\mathcal{D}_t$ may replace $f_0(e)$ by $f_0^*(e_x, y, f_0(e_x))$.

For the transformation $\mathcal{D}_t$ described above, we have the following proposition:

**Proposition 5.4** *With the function definition* $f_0^*(x, y, r) = e^*$, *we have,* $\forall d_x, d_y, d_1, d_2$,

$$\mathcal{E}[\![e^o]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y] = d_1 \wedge \mathcal{E}[\![e^*]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y] = d_2 \implies d_1 = d_2.$$

## 5.4 Cache Sets

We would like to find, in the computation of $f_0(x \oplus y)$ performed by $f_0^*(x, y, r) \equiv e^*$, expressions whose values can be retrieved quickly from the cached result $r$. Of course, literal occurrences of the subexpression $f_0(x)$ in $e^*$ can be replaced by $r$. But we aim to locate many other opportunities to use the cached result $r$. For example,

- Expressions that are provably equal to but not identical to $f_0(x)$ can be replaced by $r$.
- If $f_0(x)$ is provably equal to $g(e_x)$ and $g$ has an inverse $g^{-1}$, then any expressions $e'_x$ provably equal to $e_x$ can be replaced by $g^{-1}(r)$ if $t(g^{-1}(r)) \leq_{f_0(x)=r} t(e'_x)$.

Such equalities need not hold globally throughout $e^*$. Rather, they only need to hold in the local context of the expression that is a candidate for replacement.

For $e$ being a subexpression of $e^*$, let $I_e$ be the information set at $e$, say as computed by Step 1 or 2. We associate a *cache set* $C_e$ with $e$. We use *Cache* to denote the domain of cache sets. For any $C_e \in Cache$, $C_e$ is a set of expression pairs $\langle e_x, e_r \rangle$ such that,

1) $e_x \leftrightarrow^*_{I_e} e_r$, and
2) $x_1, ..., x_n$ are the only possible free variables of $e_x$, and $r$ is the only free variable of $e_r$.

In general, the expression pairs in $C_e$ may have no relation to $e$ other than their validity with respect to $I_e$. Note that, although all pairs $\langle e_x, e_r \rangle$ in $C_e$ satisfy 1) and 2) above, $C_e$ is not necessarily a complete set of such pairs. The computation of $C_e$ will be given in Step 3, where a cache set is collected for the expression $e$ and is then used to determine whether $e$ can be replaced by a use of the cached result $r$.

For any subexpression $e$ of $e^*$, $e_r$ is called a *substitution candidate* for $e$ iff

1) $\exists e_x, e \leftrightarrow^*_I e_x$ and $\langle e_x, e_r \rangle \in C_e$, and
2) $t(e_r) \leq_{e_x = e_r} t(e)$.

We use $e \rightarrow^*_{IC} e_r$ to denote that $e_r$ is a substitution candidate for $e$ and $t(e_r) \leq t(e'_r)$ for all substitution candidates $e'_r$ of $e$ that we can find.

## 5.5 Step 3: Using the Cache

We are now ready to redefine function $f_0^*$ so that it makes use of the cached-result parameter $r$. Step 2 defined $f_0^*$ by $f_0^*(x, y, r) = e^*$. We now redefine

$$f_0^*(x, y, r) = e^t, \tag{18}$$

where $e^t$ is derived from $e^*$ by transformation $\mathcal{S}_t$:

$$e^t \equiv \mathcal{S}_t[\![e^*]\!]\ \emptyset\ \{\langle f_0(x), r\rangle\}, \tag{19}$$

and $\mathcal{S}_t$ has type $Exp \rightarrow Info \rightarrow Cache \rightarrow Exp$. For each subexpression $e$ of $e^*$, $\mathcal{S}_t$ collects the information set $I_e$ from the context of $e$, computes the cache set $C_e$, and, where possible, substitutes $e$ with its fastest substitution candidate.

### 5.5.1 Definition of $\mathcal{S}_t$

Figure 3 defines $\mathcal{S}_t$. The first column gives a name for each rule. The subscripts of rule names have the same significance as those for $\mathcal{G}_o$. The subscript $r$ identifies a rule that does a direct substitution using the cached result. The subscript $rp$ identifies a rule that does a substitution for a predicate application.

| Name | Transformation | Condition |
|---|---|---|
| $v$ | $\mathcal{S}_t[\![v]\!]\ I\ C$ | |
| $v_r$ | $= e_r$ | if $v \rightarrow^*_{IC} e_r$ |
| $v_n$ | $= v$ | otherwise |
| $c$ | $\mathcal{S}_t[\![c(e_1, ..., e_n)]\!]\ I\ C$ | |
| $c_r$ | $= e_r$ | if $c(e_1, ..., e_n) \rightarrow^*_{IC} e_r$ |
| $c_i$ | $= c(\mathcal{S}_t[\![e_1]\!]IC, ..., \mathcal{S}_t[\![e_n]\!]IC)$ | otherwise |
| $p$ | $\mathcal{S}_t[\![p(e_1, ..., e_n)]\!]\ I\ C$ | |
| $p_r$ | $= e_r$ | if $p(e_1, ..., e_n) \rightarrow^*_{IC} e_r$ |
| $p_{rp}$ | $= p(e_{r1}, ..., e_{rn})$ | else if $p(e_1, ..., e_n)$ is predicate substitutable by $p(e_{r1}, ..., e_{rn})$ under $I$ and $C$ |
| $p_i$ | $= Simp\mathcal{S}_t[\![p(\mathcal{S}_t[\![e_1]\!]IC, ..., \mathcal{S}_t[\![e_n]\!]IC)]\!]\ I\ C$ | otherwise |
| $f$ | $\mathcal{S}_t[\![f(e_1, ..., e_n)]\!]\ I\ C$ | |
| $f_r$ | $= e_r$ | if $f(e_1, ..., e_n) \rightarrow^*_{IC} e_r$ |
| $f_{rp}$ | $= f(e_{r1}, ..., e_{rn})$ | else if $f(e_1, ..., e_n)$ is predicate substitutable by $f(e_{r1}, ..., e_{rn})$ under $I$ and $C$ |
| $f_i$ | $= f(\mathcal{S}_t[\![e_1]\!]IC, ..., \mathcal{S}_t[\![e_n]\!]IC)$ | else if $f \not\equiv f_0^*$ or $f \equiv f_0^*$ and $f_0^*(e_1, ...e_n)$ is preferred over the original function application $f_0(e)$ |
| $f_i'$ | $= \mathcal{S}_t[\![f_0(e)]\!]\ I\ C$ | otherwise |
| $if$ | $\mathcal{S}_t[\![$**if** $e_1$ **then** $e_2$ **else** $e_3]\!]\ I\ C$ | |
| $if_r$ | $= e_r$ | if **if** $e_1$ **then** $e_2$ **else** $e_3 \rightarrow^*_{IC} e_r$ |
| $if_{23}$ | $=$ **if** $e_1'$ **then** $\mathcal{S}_t[\![e_2]\!]I_2C_2$ **else** $\mathcal{S}_t[\![e_3]\!]I_3C_3$ | otherwise |
| | where $e_1' = \mathcal{S}_t[\![e_1]\!]IC$, $I_2 = I \cup \{e_1 \leftrightarrow T\}$, $C_2 = \mathcal{C}(C, I_2)$, $I_3 = I \cup \{e_1 \leftrightarrow F\}$, $C_3 = \mathcal{C}(C, I_3)$ | |
| $let$ | $\mathcal{S}_t[\![$**let** $v = e_1$ **in** $e_2$ **end**$]\!]\ I\ C$ | |
| $let_r$ | $= e_r$ | if **let** $v = e_1$ **in** $e_2$ **end** $\rightarrow^*_{IC} e_r$ |
| $let_2'$ | $=$ **let** $v = e_1'$ **in** $\mathcal{S}_t[\![e_2]\!]I_2C_2$ **end** | otherwise |
| | where $e_1' = \mathcal{S}_t[\![e_1]\!]IC$, $I_2 = I \cup \{v \leftrightarrow e_1\}$, $C_2 = \mathcal{C}(C, I_2)$ | |
| $simp$ | $Simp\mathcal{S}_t[\![p(e_1, ..., e_n)]\!]\ I\ C$ | |
| $simp_s$ | $= Simp[\![p(e_1, ..., e_n)]\!]\ I^C$ | if $p(e_1, ..., e_n)$ can be simplified with substitution under $I$ and $C$ |
| $simp_n$ | $= e$ | otherwise |

Figure 3: Definition of $\mathcal{S}_t$

$\mathcal{S}_t$ works inductively on the structure of expressions. For each expression $e$, it replaces $e$ with $e_r$

if $e \rightarrow^*_{IC} e_r$. Otherwise, it examines $e$'s subexpressions. Rule $v_n$ is the base case. Rule $c_i$ involves a straightforward application of $\mathcal{S}_t$ on each argument.

For a primitive function application $p(e_1, ..., e_n)$, if $p$ is a predicate and $p(e_1, ..., e_n)$ is *predicate substitutable* by $p(e_{r1}, ..., e_{rn})$, then rule $p_{rp}$ replaces $p(e_1, ..., e_n)$ by $p(e_{r1}, ..., e_{rn})$. The formal notion of predicate substitution will be explained later, but the idea is that $p(e_1, ..., e_n)$ may be replaced by $p(e_{r1}, ..., e_{rn})$ even if each $e_i$ may not be replaced by $e_{ri}$. For example, if sorting a list $x$ returns a sorted list $r$, i.e., $sort(x) = r$, then although, in general, $x$ is not equivalent to $r$, $null(x)$ is equivalent to $null(r)$ and thus can be replaced by $null(r)$. If none of the rules $p_r$ or $p_{rp}$ applies, then $p_i$ applies $\mathcal{S}_t$ straightforwardly on each argument. $Simp\mathcal{S}_t$ simplifies the resulting primitive function application by making use of cache $C$ and simplifier *Simp* described in Step 1. It is defined in rule *simp*. Given information set $I$ and cache set $C$, let $I^C$ denote the set $I \cup \{e_x \leftrightarrow e_r \mid \langle e_x, e_r \rangle \in C\}$. We say that primitive function application $p(e_1, ..., e_n)$ can be *simplified by substitution under* $I$ *and* $C$ if $p(e_1, ..., e_n)$ can be simplified under $I^C$; we define $Simp\mathcal{S}_t[\![p(e_1, ..., e_n)]\!]IC = e'$ if $Simp[\![p(e_1, ..., e_n)]\!]\ I^C = e'$.

For a function application $f(e_1, ..., e_n)$, if $f$ is a Boolean function and $f \not\equiv f_0^*$, then $f_{rp}$ also tries a predicate substitution as rule $p_{rp}$ does. If $f \equiv f_0^*$, then, as a result of Step 2, $f(e_1, ..., e_n) \equiv f_0^*(e_x, e_y, f_0(e_x))$ for some $e_x$ and $e_y$. In this case, $\mathcal{S}_t$ makes sure that the function application of $f_0^*$ that eventually appears in $e^t$ can be computed at least as fast as the original function application of $f_0$. Formally, if $e_r = \mathcal{S}_t[\![f_0(e_x)]\!]IC$, we say that $f_0^*(e_x, e_y, f_0(e_x))$ is *preferred* over the original function application $f_0(e)$ if $t(f_0^*(e_x, e_y, e_r)) \leq_{f_0(x)=r} t(f_0(e))$. Testing for such preference requires us to compare the definitions of $f_0^*$ and $f_0$; Proposition 5.5 gives a heuristic that can always be applied for this purpose.

**Proposition 5.5** *For* $e^o$ *as obtained from (15), if* $t(e^o) \leqq_{f_0(x)=r} t(f_0(x \oplus y))$ *and* $t(e_r) \leq_{f_0(x)=r} t(e)$ *for each* $f_0(e)$ *in* $e^o$ *that is replaced by* $f_0^*(e_x, e_y, e_r)$ *in* $e^t$, *then with* $f_0^*(x, y, r) = e^t$, *we have* $t(e^t) \leq_{f_0(x)=r} t(e^o)$ *and* $t(f_0^*(e_x, e_y, e_r)) \leq_{f_0(x)=r} t(f_0(e))$.

This heuristic requires that the time to compute $e^o$ be no greater than the time to compute $f_0(x \oplus y)$ in an absolute sense, i.e., a slowdown by a constant factor is not allowed. Since the possible slowdown is caused by computing argument expressions more than once after function unfoldings, we need to examine whether these expressions can be simplified in context so that together they take at most the time of computing the argument expressions once on every execution path where they are needed. Similarly, to ensure that $f_0^*$ computes at least as fast as $f_0$ in general without using Proposition 5.5, we need to guarantee that all these argument expressions can be either simplified in context or replaced by uses of the cached result $r$ so that together they take at most the time of computing them once on every possible path. If $f_0^*(e_x, e_y, f_0(e_x))$ is *preferred* over the original function application $f_0(e)$, then the function application of $f_0^*$ is retained in the resulting expression $e^t$ and $\mathcal{S}_t$ is applied to each argument; otherwise, the original function application $f_0(e)$ is restored and $\mathcal{S}_t$ is applied to it. Note that function applications are not further unfolded by $\mathcal{S}_t$, as all necessary unfoldings have been done by $\mathcal{G}_o$ in Step 1.

For **if** and **let**-expressions, $\mathcal{S}_t$ applies similar ideas. It collects information from the logical structure of the expression in the set $I$, enlarges the cache $C$ using a closure-like function $\mathcal{C}$, defined below, and continues making substitutions in the two branches of an **if**-expression or the body of a **let**-expression, respectively.

### 5.5.2 Auxiliary GPE and Its Usage

In order to define the notion of predicate substitution and the function $\mathcal{C}$, we need the aGPE transformation $\mathcal{G}_a$ as a subroutine. Given an expression $e_x$ and an information set $I$, $\mathcal{G}_a[\![e_x]\!]I$ specializes $e_x$ with respect to $I$. The basic idea of using $\mathcal{G}_a$ is this: given an expression $e$ in the context of information $I$, if there is $\langle e_x, e_r\rangle \in C$ such that $\mathcal{G}_a[\![e_x]\!]I = e$, then $e$ can be replaced by $e_r$ provided that $t(e_r) \leq_{e_x=e_r} t(e)$.

**Predicate Substitution.** Given a primitive function application or a function application $P(e_1, ..., e_n)$ for some predicate $P$, let $I$ and $C$ be the information set and cache set, respectively, of $P(e_1, ..., e_n)$. We say that $P(e_1, ..., e_n)$ is *predicate substitutable* with $P(e_{r1}, ..., e_{rn})$ under $I$ and $C$ if there are $\langle e_{xi}, e_{ri}\rangle \in C$ such that

$$\begin{array}{l} 1)\ \mathcal{G}_a[\![P(e_{x1}, ..., e_{xn})]\!]\ (I \cup \{P(e_1, ..., e_n) \leftrightarrow T\}) = T, \\ \quad\ \mathcal{G}_a[\![P(e_{x1}, ..., e_{xn})]\!]\ (I \cup \{P(e_1, ..., e_n) \leftrightarrow F\}) = F, \\ 2)\ P(e_{x1}, ..., e_{xn}) \text{ is defined if and only if } P(e_1, ..., e_n) \text{ is defined, and} \\ 3)\ t(P(e_{r1}, ..., e_{rn})) \leq_{e_{x1}=e_{r1}\wedge...\wedge e_{xn}=e_{rn}} t(P(e_1, ..., e_n)). \end{array} \tag{20}$$

In the matrix multiplication example of Section 3, consider $null(C)$ in expression (10). We have $\langle mtxMul(C, R), r\rangle$ in the cache set, and

$$\mathcal{G}_a[\![null(mtxMul(C, R))]\!]\ \{null(C) \leftrightarrow T\} = T, \quad \mathcal{G}_a[\![null(mtxMul(C, R))]\!]\ \{null(C) \leftrightarrow F\} = F,$$

since when $null(C)$ equals $T$, $mtxMul(C, R)$ is specialized to $nil$ and thus $null(mtxMul(C, R)$ to $T$, and when $null(C)$ equals $F$, $mtxMul(C, R)$ is specialized to a *cons* structure and thus $null(mtxMul(C, R)$ to $F$. Therefore, the expression $null(C)$ can actually be replaced by $null(r)$.

**Closure Function.** The function $\mathcal{C}$ used in **if** and **let** rules has type $Cache \rightarrow Info \rightarrow Cache$. For any $C \in Cache, I \in Info$, we call $\mathcal{C}(C, I)$ the *closure of $C$ under $I$* and define $\mathcal{C}(C, I) = C \cup C' \cup C''$, where

$$\begin{array}{l} C' = \{\langle \mathcal{G}_a[\![e_x]\!]I, e_r\rangle \mid \langle e_x, e_r\rangle \in C\}, \\ C'' = \{\langle e'_x, g^{-1}(e_r)\rangle \mid \langle e_x, e_r\rangle \in C,\ \mathcal{G}_a[\![e_x]\!]I \leftrightarrow^*_I g(e'_x)\}. \end{array} \tag{21}$$

For any $\langle e_x, e_r\rangle \in C$, if $\mathcal{G}_a$ specializes $e_x$ to $e$ under $I$, then $\langle e, e_r\rangle$ is added into $C'$. If $e$ is provably equal to $g(e'_x)$ for some primitive function $g$ and $g$ has an inverse $g^{-1}$, then $\langle e'_x, g^{-1}(e_r)\rangle$ is added into $C''$. In particular, if $e \equiv c(e'_{x1}, ..., e'_{xn})$ for some constructor $c$ and $c$'s corresponding selectors are $c_i^{-1}$'s, then $\langle e'_{xi}, c_i^{-1}(e_r)\rangle$ is added into $C''$ for $i = 1, ..., n$.

In the example of Section 3, the expression in the false branch of (7) has $I = \{null(C) \leftrightarrow F\}$ and $C = \{\langle mtxMul(C, R), r\rangle\}$. Then $C' = \{\langle cons(rowMul(car(C), R), mtxMul(cdr(C), R)), r\rangle\}$ by (8), and $C'' = \{\langle rowMul(car(C), R), car(r)\rangle, \langle mtxMul(cdr(C), R)), cdr(r)\rangle\}$ by (9). Thus, the cache set associated with that expression would be $C \cup C' \cup C''$ by (21).

**Auxiliary GPE.** From the above definitions, we see how $\mathcal{G}_a$ transformations are used. The goal is to help find more expressions whose values can be retrieved from a cached result. Therefore, an unfolding by $\mathcal{G}_a$ is beneficial for this purpose only if the unfolded expression can be specialized to a new expression under the given information $I$. In general, the new expression can be obtained in one of the following two ways:

1) simplification of primitive function applications using *Simp* as rule $p_e$ in $\mathcal{G}_o$;
2) specialization of **if** or **let** expressions using rules like $if_2$, $if_3$, or $let_2$ in $\mathcal{G}_o$.

Therefore, we unfold a function application only if the unfolded expression or any of its subexpressions can be simplified using *Simp* or specialized to a strict subexpression of itself. Other than the unfolding rule, $\mathcal{G}_a$ does most of the other transformations and simplifications in a fashion similar to $\mathcal{G}_o$.

Figure 4 defines $\mathcal{G}_a$, which has type $Exp \rightarrow Info \rightarrow Exp$. The definition of $\mathcal{G}_a$ is mostly the same as for $\mathcal{G}_o$; it differs only in its treatment of function applications and **let**-expressions. The notion of *irreducibility* for $\mathcal{G}_a$ is similar to irreducibility for $\mathcal{G}_o$.

Consider a function application $f(e_1, ..., e_n)$ in the context of information set $I$, where the $e_i$'s are already reduced by $\mathcal{G}_a$ and are not **if** or **let**-expressions. Suppose $e$ is the expression obtained by unfolding $f$ on $\langle e_1, ..., e_n\rangle$. We say that $f(e_1, ..., e_n)$ is *expandable by $\mathcal{G}_a$* if $e$ or any of $e$'s subexpressions can be simplified by *Simp* under $I$ or specialized to be a strict subexpression of itself under $I$. $\mathcal{G}_a$ unfolds a function application $f(e_1, ..., e_n)$ if and only if it is expandable. Recall

| Name | Transformation | Condition |
|---|---|---|
| $f$ | $\mathcal{G}_a[\![f(e_1,...,e_n)]\!]\ I$ | |
| $f_e$ | $= \mathcal{G}_a[\![e[v_1 := e_1,...,v_n := e_n]]\!]\ I$ where $f$ is defined by $f(v_1,...,v_n) = e$ | if $e_1,...,e_n$ are irreducible, not **if** or **let**, but $f(e_1,...,e_n)$ is expandable. |
| $let$ | $\mathcal{G}_a[\![$**let** $v = e_1$ **in** $e_2$ **end**$]\!]\ I$ | |
| $let_e$ | $= \mathcal{G}_a[\![e_2[v := e_1]]\!]\ I$ | otherwise |
| $let_n$ | < this case is deleted > | |
| $*$ | all other rules are the same as those for $\mathcal{G}_o$. | |

Figure 4: Definition of $\mathcal{G}_a$

that $\mathcal{G}_o$ unfolds $f(e_1,...,e_n)$ if and only if its argument pattern is favorable and occurs the first time for $f$. Here again, the unfolding rule $f_e$ is a potential source of nontermination.

For a **let**-expression, $\mathcal{G}_a$ applies rules $let_1, let_{1-if}, let_{1-let}, let_2, let'_2$ in turn, if possible. At the end, instead of possibly leaving the binding in the residual expression as $\mathcal{G}_o$ does, $\mathcal{G}_a$ replaces occurrences of the defined variable $v$ in $e_2$ by its defining expression $e_1$. This is merely an implementation strategy so that, when using the cache set, we do not have to keep track of equations introduced by the bindings of **let**-expressions. When a program has many **let**-expressions, this may cause code blow-up. However, when a program has a lot of **let**-expressions, we expect that many of them can be eliminated by the $let_2$ rule in the context of the information collected for the $\mathcal{G}_a$ transformation.

We can prove the following proposition for $\mathcal{G}_a$ by induction on the structure of expressions:

**Proposition 5.6** *For any expression $e$ and information set $I$ of $e$, let $v_1,...,v_k$ be all the free variables in $e$ and $e_I$. If $\mathcal{G}_a[\![e]\!]I$ terminates, then we have $\forall d_v$, if $\mathcal{E}[\![e_I]\!]\ [v \rightarrowtail d_v] = T$, then $\forall d$, $\mathcal{E}[\![e]\!]\ [v \rightarrowtail d_v] = d \implies \mathcal{E}[\![\mathcal{G}_a[\![e]\!]I]\!]\ [v \rightarrowtail d_v] = d$.*

**Corollary 5.7** *For any cache set $C$ and information set $I$, let $v_1,...,v_k$ be all the free variables in $e_I$ and $e_x$ and $e_r$'s for all $\langle e_x, e_r\rangle \in C$. Then we have $\forall d_v$, if $\mathcal{E}[\![e_I]\!]\ [v \rightarrowtail d_v] = T$ and $\forall \langle e_x, e_r\rangle \in C$, $\mathcal{E}[\![e_x]\!]\ [v \rightarrowtail d_v] = \mathcal{E}[\![e_r]\!]\ [v \rightarrowtail d_v]$, then $\forall \langle e_x, e_r\rangle \in \mathcal{C}(C, I)$, $\mathcal{E}[\![e']\!]\ [v \rightarrowtail d_v] = \mathcal{E}[\![e_r]\!]\ [v \rightarrowtail d_v]$.*

We redefine $f_0^*$ to be $f_0^*(x, y, r) = e^t$. We have the following proposition for $\mathcal{S}_t$ transformation:

**Proposition 5.8** *For any expression $e$, information set $I$ and cache set $C$ of $e$, let $v_1,...,v_k$ be all the free variables in $e$, $e_I$, and $e_x$ and $e_r$'s for all $\langle e_x, e_r\rangle \in C$. Let $e_1 \equiv \mathcal{D}_t[\![e]\!]I$. If $\mathcal{S}_t[\![e_1]\!]IC$ terminates with $e_2 \equiv \mathcal{S}_t[\![e_1]\!]IC$, then, with definition $f_0^*(x, y, r) = e^t$, $\forall d_v$, if $\mathcal{E}[\![e_I]\!]\ [v \rightarrowtail d_v] = T$ and $\forall \langle e_x, e_r\rangle \in C$, $\mathcal{E}[\![e_x]\!]\ [v \rightarrowtail d_v] = \mathcal{E}[\![e_r]\!]\ [v \rightarrowtail d_v]$, then $\forall d_1, d_2$, $\mathcal{E}[\![e]\!]\ [v \rightarrowtail d_v] = d_1 \wedge \mathcal{E}[\![e_2]\!]\ [v \rightarrowtail d_v] = d_2$ $\implies d_1 = d_2$, and $t(e_2) \leq_{e_{IC}} t(e)$.*

Therefore, if the $\mathcal{G}_a$ transformations issued by $\mathcal{S}_t$ terminate and we get $e^t \equiv \mathcal{S}_t[\![e^*]\!]\ \emptyset\ \{\langle f_0(x), r\rangle\}$, then we have the following corollary:

**Corollary 5.9** *If $\mathcal{S}_t[\![e^*]\!]\ \emptyset\ \{\langle f_0(x), r\rangle\}$ terminates with $e^t \equiv \mathcal{S}_t[\![e^*]\!]\ \emptyset\ \{\langle f_0(x), r\rangle\}$, then with definition $f_0^*(x,y,r) = e^t$, we have $\forall d_x, d_y, d_r, d_1, d_2$, if $\mathcal{E}[\![f_0(x)]\!]\ [x \rightarrowtail d_x] = d_r$, then*

$$\mathcal{E}[\![e^o]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y] = d_1 \wedge \mathcal{E}[\![e^t]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y, r \mapsto d_r] = d_2 \implies d_1 = d_2,$$

*and $t(e^t) \leq_{f_0(x)=r} t(e^o)$.*

Finally, we have our main result, which follows directly from Corollary 5.3 and Corollary 5.9: the semantics of the programs are preserved by the series of transformations above and computations using the differentiated versions are asymptotically at least as fast as computations using the original programs.

**Proposition 5.10** *If the oGPE and aGPE transformations terminate, then with the function definition $f_0^*(x,y,r) = e^t$, we have,*

*(1) $\forall d_x, d_y, d_r$, if $\mathcal{E}[\![f_0(x)]\!]\ [x \rightarrowtail d_x] = d_r$, then $\forall d_1, d_2$,*
$\quad\mathcal{E}[\![f_0(x \oplus y)]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y] = d_1 \wedge \mathcal{E}[\![f_0^*(x,y,r)]\!]\,[x \rightarrowtail d_x, y \rightarrowtail d_y, r \mapsto d_r] = d_2 \implies d_1 = d_2$,
*(2) $t(f_0^*(x,y,r)) \leq_{f_0(x)=r} t(f_0(x \oplus y))$.*

### 5.6 Step 4: Redundant Code Elimination

The purpose of this last step is to optimize the function $f_0^*(x,y,r) = e^t$ in the context of the set of functions $F$. The optimizations mainly involve redundant code elimination. At least the following three kinds of code may become redundant as a result of the above transformations.

First, some function definitions in $F$ may have become irrelevant to the evaluation of $f_0^*(x,y,r)$. This is because: After $f_0(x \oplus y)$ is expanded into $e^o$ by Step 1, $f_0^*$ with $r$ is introduced in Step 2 to form $e^*$ and then used in Step 3 to replace some expressions in $e^*$ to form $e^t$. The replaced expressions may be function applications. If all calls to a function are replaced, then the definition of the function becomes redundant. In particular, some function calls to the function $f_0$ are possibly replaced in Step 2 by calls to $f_0^*$. As a result, $f_0$ may become redundant even without substitutions in Step 3.

Second, some computations inside remaining functions may also have become redundant. This is obvious. Since some expressions are replaced by calls to $f_0^*$ or expressions involving $r$, some other expressions whose results are used for the computations of these replaced expressions may simply become unnecessary.

Third, some parameters of remaining functions may also have become redundant. This occurs because some computations dependent on the parameter $x$ are altered by the replacements in Step 3 to be dependent on the cache parameter $r$. Hence all or part of the parameter $x$ may have become irrelevant to the final computation.

There has been considerable work on such optimizations [1]. Standard data-flow and control-flow analysis algorithms can be adapted to reason about the dependencies within the programs and recognize the redundant code to be eliminated.

After optimization, we get the resulting function definition $f_0'(\bar{x}, y, r) = e'$, where $\bar{x} = \langle x_{i_1}, ..., x_{i_k} \rangle$, $1 \leq i_1 < ... < i_k \leq n$, in the context of a set, say $F'$ (including $f_0'$), of function definitions. $f_0'$ is a differentiated version of $f_0$. If the correctness of these standard optimizations guarantees that $\forall d_x, d_y, d_r, d$,

$$\mathcal{E}[\![f_0^*(x, y, r)]\!]\ [x \rightarrowtail d_x, y \rightarrowtail d_y, r \mapsto d_r] = d \iff \mathcal{E}[\![f_0'(\bar{x}, y, r)]\!]\ [x \rightarrowtail d_x, y \rightarrowtail d_y, r \mapsto d_r] = d,$$

and $t(f_0'(\bar{x}, y, r)) \leq_{f_0(x)=r} t(f_0^*(x, y, r))$, then combining this with Proposition 5.10, we have that $\forall d_x, d_y, d_r$, if $\mathcal{E}[\![f_0(x)]\!]\ [x \rightarrowtail d_x] = d_r$, then $\forall d_1, d_2$,

$$\mathcal{E}[\![f_0(x \oplus y)]\!]\ [x \rightarrowtail d_x, y \rightarrowtail d_y] = d_1 \wedge \mathcal{E}[\![f_0'(\bar{x}, y, r)]\!]\ [x \rightarrowtail d_x, y \rightarrowtail d_y, r \mapsto d_r] = d_2 \implies d_1 = d_2,$$

and $t(f_0'(\bar{x}, y, r)) \leq_{f_0(x)=r} t(f_0(x \oplus y))$.

Note that the optimizations of this last step try to make the resulting function $f_0'$ depend as little on the input $x$ as possible by eliminating those $x_i$'s with $i \in \{1, ..., n\} - \{i_1, ..., i_k\}$. If $x$ can be entirely eliminated, then $f_0'(y, r)$ is a complete differentiated version of $f_0$.

# 6 Example: Derivation of Insert from Sort

We are given a program for select sort consisting of the function definitions for *sort*, *least*, and *rest*. *Sort* takes a list $x$ of numbers and returns a sorted list $r$ of these numbers. Let the change in the input of *sort* be that an extra element $i$ is added at the beginning of the list $x$. Formally, we have

function definition:

```
    sort(x) =     if null(x) then nil
                  else let k = least(x)
                       in cons(k, sort(rest(x, k))) end
    least(x) =    if null(cdr(x)) then car(x)
                  else let s = least(cdr(x))
                       in if car(x) < s then car(x) else s end
    rest(x, k) =  if k = car(x) then cdr(x)
                  else cons(car(x), rest(cdr(x), k))
```

cached result:

$$\begin{array}{l} \quad sort(x) = r \\ \text{change in input:} \\ \quad x' = cons(i,x) \end{array}$$

We apply the series of transformations discussed in Section 5 to derive a differentiated version of *sort*. Instead of sorting the new input $cons(i,x)$ from scratch, the derived program just inserts the element $i$ into the sorted list $r$. Therefore, it is a complete differentiated version of *sort*. In fact, the derived program is also a complete incremental version of *sort*.

**Step 1.** Applying $\mathcal{G}_o$ on $sort(cons(i,x))$ produces the following transformations. Only the most substantial steps are listed.

$$\begin{array}{ll}
\mathcal{G}_o[\![ sort(cons(i,x)) ]\!]\emptyset & \\
\quad = \mathcal{G}_o[\![ \textbf{if } null(cons(i,x)) \textbf{ then } nil & \\
\qquad\quad \textbf{else let } k = least(cons(i,x)) & \\
\qquad\qquad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\qquad \textbf{end} ]\!]\emptyset & (f_e) \\
\quad = \mathcal{G}_o[\![ \textbf{let } k = least(cons(i,x)) & \\
\qquad\quad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\quad \textbf{end} ]\!]\emptyset & (if_3) \\
\quad = \mathcal{G}_o[\![ \textbf{let } k = \mathcal{G}_o[\![ \textbf{if } null(cdr(cons(i,x))) \textbf{ then } car(cons(i,x)) & \\
\qquad\qquad\qquad\qquad \textbf{else let } s = least(cdr(cons(i,x))) & \\
\qquad\qquad\qquad\qquad\quad \textbf{in if } car(cons(i,x)) < s \textbf{ then } car(cons(i,x)) & \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } s & \\
\qquad\qquad\qquad\qquad\quad \textbf{end} ]\!]\emptyset & \\
\qquad\quad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\quad \textbf{end} ]\!]\emptyset & (let_1, f_e) \\
\quad = \mathcal{G}_o[\![ \textbf{let } k = \textbf{if } null(x) \textbf{ then } i & \\
\qquad\qquad\qquad \textbf{else let } s = least(x) & \\
\qquad\qquad\qquad\quad \textbf{in if } i < s \textbf{ then } i \textbf{ else } s & \\
\qquad\qquad\qquad\quad \textbf{end} & \\
\qquad\quad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\quad \textbf{end} ]\!]\emptyset & (p_e) \\
\quad = \mathcal{G}_o[\![ \textbf{if } null(x) \textbf{ then let } k = i & \\
\qquad\qquad\qquad\quad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\qquad\qquad\quad \textbf{end} & \\
\qquad\quad \textbf{else let } k = \textbf{let } s = least(x) & \\
\qquad\qquad\qquad\quad \textbf{in if } i < s \textbf{ then } i \textbf{ else } s & \\
\qquad\qquad\qquad\quad \textbf{end} & \\
\qquad\qquad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\qquad \textbf{end} ]\!]\emptyset & (let_{1-if}) \\
\quad = \textbf{if } null(x) \textbf{ then } \mathcal{G}_o[\![ \textbf{let } k = i & \\
\qquad\qquad\qquad\quad \textbf{in } cons(k, \mathcal{G}_o[\![ sort(\textbf{if } k = i \textbf{ then } nil & \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } cons(i, rest(nil,k))) & \\
\qquad\qquad\qquad\qquad\qquad ]\!]\{null(x) \leftrightarrow T, k=i \leftrightarrow T\}) & \\
\qquad\qquad\qquad\quad \textbf{end} ]\!]\{null(x) \leftrightarrow T\} & \\
\qquad \textbf{else } \mathcal{G}_o[\![ \textbf{let } s = least(x) & \\
\qquad\qquad\quad \textbf{in let } k = \textbf{if } i < s \textbf{ then } i \textbf{ else } s & \\
\qquad\qquad\qquad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\qquad\qquad \textbf{end} & \\
\qquad\qquad\quad \textbf{end} ]\!]\{null(x) \leftrightarrow F\} & (if_{23}, v_c, f_e, p_e, let_{1-let}) \\
\quad = \textbf{if } null(x) \textbf{ then } \mathcal{G}_o[\![ \textbf{let } k = i \textbf{ in } cons(k, sort(nil)) \textbf{ end} & \\
\qquad\qquad\qquad ]\!]\{null(x) \leftrightarrow T\} & \\
\qquad \textbf{else let } s = least(x) & \\
\qquad\qquad \textbf{in } \mathcal{G}_o[\![ \textbf{if } i < s \textbf{ then let } k = i & \\
\qquad\qquad\qquad\qquad\quad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\qquad\qquad\qquad\quad \textbf{end} & \\
\qquad\qquad\qquad \textbf{else let } k = s & \\
\qquad\qquad\qquad\quad \textbf{in } cons(k, sort(rest(cons(i,x),k))) & \\
\qquad\qquad\qquad\quad \textbf{end} ]\!]\{null(x) \leftrightarrow F, s \leftrightarrow least(x)\} & \\
\qquad\qquad \textbf{end} & (if_2, let_{1-if}) \\
\quad = \textbf{if } null(x) \textbf{ then } \mathcal{G}_o[\![ \textbf{let } k = i \textbf{ in } cons(k, nil) \textbf{ end} & \\
\qquad\qquad\qquad ]\!]\{null(x) \leftrightarrow T\} &
\end{array}$$

$$
\begin{array}{ll}
\quad\mathbf{else\ let}\ s = least(x) & \\
\qquad\quad \mathbf{in\ if}\ i < s & \\
\qquad\qquad \mathbf{then}\ \mathcal{G}_o\llbracket \mathbf{let}\ k = i & \\
\qquad\qquad\qquad\quad \mathbf{in}\ cons(k, \mathcal{G}_o\llbracket sort(\mathbf{if}\ k = i\ \mathbf{then}\ x & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ cons(i, rest(x,k))) & \\
\qquad\qquad\qquad\qquad\qquad \rrbracket(I_2 \cup \{k{=}i \leftrightarrow T\})\ ) & \\
\qquad\qquad\qquad\quad \mathbf{end}\rrbracket I_2 & \\
\qquad\qquad \mathbf{else}\ \mathcal{G}_o\llbracket \mathbf{let}\ k = s & \\
\qquad\qquad\qquad\quad \mathbf{in}\ cons(k, \mathcal{G}_o\llbracket sort(\mathbf{if}\ k = i\ \mathbf{then}\ x & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ cons(i, rest(x,k))) & \\
\qquad\qquad\qquad\qquad\qquad \rrbracket(I_3 \cup \{k{=}s \leftrightarrow T\})\ ) & \\
\qquad\qquad\qquad\quad \mathbf{end}\rrbracket I_3 & \\
\qquad\quad \mathbf{end} & (f_e, if_2, if_{23}, f_e, p_e, f_e, p_e) \\
\quad \text{where}\ I_2 = \{null(x) \leftrightarrow F, s \leftrightarrow least(x), i < s \leftrightarrow T\} & \\
\qquad\quad I_3 = \{null(x) \leftrightarrow F, s \leftrightarrow least(x), i < s \leftrightarrow F\} & \\
= \mathbf{if}\ null(x)\ \mathbf{then}\ cons(i, nil) & \\
\quad \mathbf{else\ let}\ s = least(x) & \\
\qquad\quad \mathbf{in\ if}\ i < s\ \mathbf{then}\ \mathcal{G}_o\llbracket \mathbf{let}\ k = i\ \mathbf{in}\ cons(k, sort(x))\ \mathbf{end}\rrbracket I_2 & \\
\qquad\qquad \mathbf{else}\ \mathcal{G}_o\llbracket \mathbf{let}\ k = s & \\
\qquad\qquad\qquad \mathbf{in\ if}\ k = i\ \mathbf{then}\ cons(k, sort(x)) & \\
\qquad\qquad\qquad\quad \mathbf{else}\ cons(k, sort(cons(i, rest(x,k)))) & \\
\qquad\qquad\qquad \mathbf{end}\rrbracket I_3 & \\
\qquad\quad \mathbf{end} & (let_e, if_2, f_{1-if}, p_{2-if}) \\
= \mathbf{if}\ null(x)\ \mathbf{then}\ cons(i, nil) & \\
\quad \mathbf{else\ let}\ s = least(x) & \\
\qquad\quad \mathbf{in\ if}\ i < s\ \mathbf{then}\ cons(i, sort(x)) & \\
\qquad\qquad \mathbf{else\ if}\ s = i\ \mathbf{then}\ cons(s, sort(x)) & \\
\qquad\qquad\quad \mathbf{else}\ cons(s, sort(cons(i, rest(x,s)))) & \\
\qquad\quad \mathbf{end} & (let_e, let_e) \\
\equiv e^o &
\end{array}
$$

**Step 2.** Define $sort^*(x,i,r)$, where $r = sort(x)$, to be $e^o$ with $sort(cons(i, rest(x,s)))$ in $e^o$ replaced by $sort^*(rest(x,s), i, sort(rest(x,s)))$. We get

$$
\begin{array}{l}
sort^*(x,i,r) \\
\quad = \mathcal{D}_t\llbracket e^o \rrbracket\ \emptyset \\
\quad = \mathbf{if}\ null(x)\ \mathbf{then}\ cons(i, nil) \\
\qquad \mathbf{else\ let}\ s = least(x) \\
\qquad\qquad \mathbf{in\ if}\ i < s\ \mathbf{then}\ cons(i, sort(x)) \\
\qquad\qquad\quad \mathbf{else\ if}\ s = i\ \mathbf{then}\ cons(s, sort(x)) \\
\qquad\qquad\qquad \mathbf{else}\ cons(s, sort^*(rest(x,s), i, sort(rest(x,s)))) \\
\qquad\qquad \mathbf{end} \\
\quad \equiv e^*
\end{array}
$$

**Step 3.** First, we itemize here those $\mathcal{G}_a$ transformations that are needed.

$$
\begin{array}{ll}
\mathcal{G}_a\llbracket sort(x) \rrbracket \{null(x) \leftrightarrow T\} & \\
\quad = nil & (f_e, if_2) \\
\mathcal{G}_a\llbracket sort(x) \rrbracket \{null(x) \leftrightarrow F\} & \\
\quad = \mathcal{G}_a\llbracket \mathbf{let}\ k = least(x)\ \mathbf{in}\ cons(k, sort(rest(x,k)))\ \mathbf{end} & \\
\qquad\quad \rrbracket \{null(x) \leftrightarrow F\} & (f_e, if_3) \\
\quad = \mathcal{G}_a\llbracket cons(least(x), sort(rest(x, least(x))))\ \mathbf{end} & \\
\qquad\quad \rrbracket \{null(x) \leftrightarrow F\} & (let_e) \\
\quad = cons(least(x), sort(rest(x, least(x)))) & (v_n, f_n, p_n)
\end{array}
$$

Then, apply the transformation $\mathcal{S}_t$ on $e^*$.

$$
\begin{array}{l}
\mathcal{S}_t\llbracket e^* \rrbracket\ \emptyset\ \{\langle sort(x), r\rangle\} \\
= \mathcal{S}_t\llbracket \mathbf{if}\ null(x)\ \mathbf{then}\ cons(i, nil) \\
\qquad \mathbf{else\ let}\ s = least(x) \\
\qquad\qquad \mathbf{in\ if}\ i < s\ \mathbf{then}\ cons(i, sort(x))
\end{array}
$$

$$
\begin{array}{ll}
\qquad\qquad\quad \mathbf{else\ if}\ s = i\ \mathbf{then}\ \mathit{cons}(s, \mathit{sort}(x)) & \\
\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{cons}(s, \mathit{sort}^*(\mathit{rest}(x,s), i, \mathit{sort}(\mathit{rest}(x,s)))) & \\
\qquad\qquad \mathbf{end}\rrbracket\ \emptyset\ \{\langle \mathit{sort}(x), r\rangle\} & \\
= \mathbf{if}\ \mathit{Cond}\ \mathbf{then}\ \mathcal{S}_t\llbracket \mathit{cons}(i,\mathit{nil})\rrbracket\ \{\mathit{null}(x) \leftrightarrow T\} & \\
\qquad\qquad\qquad\qquad\qquad\quad \{\langle \mathit{sort}(x), r\rangle, \langle R_2, r\rangle\} & \\
\quad \mathbf{else}\ \mathcal{S}_t\llbracket \mathbf{let}\ s = \mathit{least}(x) & \\
\qquad\qquad \mathbf{in\ if}\ i < s\ \mathbf{then}\ \mathit{cons}(i, \mathit{sort}(x)) & \\
\qquad\qquad\quad \mathbf{else\ if}\ s = i\ \mathbf{then}\ \mathit{cons}(s, \mathit{sort}(x)) & \\
\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{cons}(s, \mathit{sort}^*(\mathit{rest}(x,s), i, \mathit{sort}(\mathit{rest}(x,s)))) & \\
\qquad\qquad \mathbf{end}\rrbracket\ \{\mathit{null}(x) \leftrightarrow F\}\ C & (\mathit{if}_{23}) \\
\quad \text{where } \mathit{Cond} = \mathcal{S}_t\llbracket \mathit{null}(x)\rrbracket\ \emptyset\ \{\langle \mathit{sort}(x), r\rangle\} = \mathit{null}(r), & (p_{rp}) \\
\qquad\qquad C = \{\langle \mathit{sort}(x), r\rangle, \langle R_3, r\rangle, \langle R_{31}, \mathit{car}(r)\rangle, \langle R_{32}, \mathit{cdr}(r)\rangle\} & \\
\qquad\qquad R_2 = \mathcal{G}_a\llbracket \mathit{sort}(x)\rrbracket\ \{\mathit{null}(x) \leftrightarrow T\} = \mathit{nil}, & \\
\qquad\qquad R_3 = \mathcal{G}_a\llbracket \mathit{sort}(x)\rrbracket\ \{\mathit{null}(x) \leftrightarrow F\} & \\
\qquad\qquad\quad = \mathit{cons}(\mathit{least}(x), \mathit{sort}(\mathit{rest}(x, \mathit{least}(x)))) & \\
\qquad\qquad R_{31} = \mathit{least}(x), \text{ and} & \\
\qquad\qquad R_{32} = \mathit{sort}(\mathit{rest}(x, \mathit{least}(x))) & \\
= \mathbf{if}\ \mathit{null}(r)\ \mathbf{then}\ \mathit{cons}(i,\mathit{nil}) & \\
\quad \mathbf{else\ let}\ s = R_{St} & \\
\qquad\quad \mathbf{in}\ \mathcal{S}_t\llbracket \mathbf{if}\ i < s\ \mathbf{then}\ \mathit{cons}(i, \mathit{sort}(x)) & \\
\qquad\qquad\quad \mathbf{else\ if}\ s = i\ \mathbf{then}\ \mathit{cons}(s, \mathit{sort}(x)) & \\
\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{cons}(s, \mathit{sort}^*(\mathit{rest}(x,s), i, \mathit{sort}(\mathit{rest}(x,s)))) & \\
\qquad\quad \mathbf{end}\rrbracket\ I\ C & (p_n, \mathit{let}_2) \\
\quad \text{where } R_{St} = \mathcal{S}_t\llbracket \mathit{least}(x)\rrbracket\ \{\mathit{null}(x) \leftrightarrow F\}\ C & \\
\qquad\qquad\quad = \mathit{car}(r) & (f_r) \\
\qquad\qquad I = \{\mathit{null}(x) \leftrightarrow F, s \leftrightarrow \mathit{least}(x), s \leftrightarrow R_{St}\} & \\
= \mathbf{if}\ \mathit{null}(r)\ \mathbf{then}\ \mathit{cons}(i,\mathit{nil}) & \\
\quad \mathbf{else\ let}\ s = \mathit{car}(r) & \\
\qquad\quad \mathbf{in\ if}\ i < s\ \mathbf{then}\ \mathcal{S}_t\llbracket \mathit{cons}(i, \mathit{sort}(x))\rrbracket\ (I \cup \{i < s \leftrightarrow T\})\ C & \\
\qquad\qquad \mathbf{else}\ \mathcal{S}_t\llbracket \mathbf{if}\ s = i\ \mathbf{then}\ \mathit{cons}(s, \mathit{sort}(x)) & \\
\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{cons}(s, \mathit{sort}^*(\mathit{rest}(x,s), i, \mathit{sort}(\mathit{rest}(x,s)))) & \\
\qquad\qquad\qquad \rrbracket\ (I \cup \{i < s \leftrightarrow F\})\ C & \\
\qquad\quad \mathbf{end} & (p_n, \mathit{if}_{23}) \\
= \mathbf{if}\ \mathit{null}(r)\ \mathbf{then}\ \mathit{cons}(i,\mathit{nil}) & \\
\quad \mathbf{else\ let}\ s = \mathit{car}(r) & \\
\qquad\quad \mathbf{in\ if}\ i < s\ \mathbf{then}\ \mathit{cons}(i, r) & \\
\qquad\qquad \mathbf{else\ if}\ s = i\ \mathbf{then}\ \mathcal{S}_t\llbracket \mathit{cons}(s, \mathit{sort}(x))\rrbracket\ (I \cup \{s = i \leftrightarrow T\})\ C & \\
\qquad\qquad\quad \mathbf{else}\ \mathcal{S}_t\llbracket \mathit{cons}(s, \mathit{sort}^*(\mathit{rest}(x,s), i, \mathit{sort}(\mathit{rest}(x,s)))) & \\
\qquad\qquad\qquad\qquad \rrbracket\ (I \cup \{i < s \leftrightarrow F\} \cup \{i = s \leftrightarrow F\})\ C & \\
\qquad\quad \mathbf{end} & (p_i, f_r, \mathit{if}_{23}) \\
= \mathbf{if}\ \mathit{null}(r)\ \mathbf{then}\ \mathit{cons}(i,\mathit{nil}) & \\
\quad \mathbf{else\ let}\ s = \mathit{car}(r) & \\
\qquad\quad \mathbf{in\ if}\ i < s\ \mathbf{then}\ \mathit{cons}(i, r) & \\
\qquad\qquad \mathbf{else\ if}\ s = i\ \mathbf{then}\ \mathit{cons}(s, r) & \\
\qquad\qquad\quad \mathbf{else}\ \mathit{cons}(s, \mathit{sort}^*(\mathit{rest}(x,s), i, \mathit{cdr}(r))) & \\
\qquad\quad \mathbf{end} & (p_i, f_r, p_i, f_i, f_r) \\
\equiv e^t &
\end{array}
$$

**Step 4.** Apply standard optimizations on $\mathit{sort}^*(x,i,r) = e^t$. The function definitions *sort*, *least*, *rest*, and the argument $x$ to the function $\mathit{sort}^*$ are eliminated eventually.

(merge conditional branches)

$$
\begin{array}{l}
\mathit{sort}^*(x,i,r) \\
\quad = \mathbf{if}\ \mathit{null}(r)\ \mathbf{then}\ \mathit{cons}(i,\mathit{nil}) \\
\qquad \mathbf{else\ let}\ s = \mathit{car}(r) \\
\qquad\qquad \mathbf{in\ if}\ i \le s\ \mathbf{then}\ \mathit{cons}(i, r) \\
\qquad\qquad\quad \mathbf{else}\ \mathit{cons}(s, \mathit{sort}^*(\mathit{rest}(x,s), i, \mathit{cdr}(r))) \\
\qquad\qquad \mathbf{end}
\end{array}
$$

(eliminate redundant argument $x$ to $\mathit{sort}^*$ to get $\mathit{sort}'$)

$$
\begin{array}{l}
\mathit{sort}'(i,r) \\
\quad = \mathbf{if}\ \mathit{null}(r)\ \mathbf{then}\ \mathit{cons}(i,\mathit{nil}) \\
\qquad \mathbf{else\ let}\ s = \mathit{car}(r) \\
\qquad\qquad \mathbf{in\ if}\ i \le s\ \mathbf{then}\ \mathit{cons}(i, r) \\
\qquad\qquad\quad \mathbf{else}\ \mathit{cons}(s, \mathit{sort}'(i, \mathit{cdr}(r))) \\
\qquad\qquad \mathbf{end}
\end{array}
$$

(miscellaneous: **let** can be either unfolded or not)
$sort'(i,r)$
  $=$ **if** $null(r)$ **then** $cons(i,nil)$
    **else if** $i \leq car(r)$ **then** $cons(i,r)$
      **else** $cons(car(r), sort'(i, cdr(r)))$

**Analysis.** For $x$ of length $n$, the time complexity of $sort(x)$ is $O(n^2)$, and thus $sort(cons(i,x))$ takes $O(n^2)$ time. But $sort'$ inserts the element $i$ into the sorted list $r$ in $O(n)$ time. Thus, we have obtained a linear speedup by making use of the cached result $r$.

## 7 Discussion

In our presentation, we have emphasized conveying basic ideas and exposing basic problems. At several places in the transformation, algorithms need to be further studied in order to make the transformational approach efficient. Moreover, effective algorithms are also important for achieving a higher degree of incrementality in the derived programs, which is the main concern of this paper.

**Issues related to the degree of incrementality.** Informally, we say that a derived program has a higher degree of incrementality if more of its computation is avoided by using the cached result. There are two main issues in the transformational approach that relate to the degree of incrementality achievable in derived programs.

First, unfolding strategies affect the degree of incrementality that can be derived. Since unrestricted unfoldings may lead to nontermination, conditions are attached to unfolding rules. However, at the same time that these conditions restrict unfoldings, they may also mask opportunities for discovering a higher degree of incrementality. In $\mathcal{G}_o$, we only unfold calls to a function at most once for each of its favorable argument patterns. Thus, some computations depending purely on $x$ may not be separated out in Step 1 and hence can not be replaced by their substitution candidates in Step 3. In $\mathcal{G}_a$, we only unfold function calls that are expandable. At a result, some auxiliary residual expressions with substitution candidates may not be found by $\mathcal{G}_a$. Therefore, in both transformations, more lenient unfolding conditions could be studied to help achieve a higher degree of incrementality in the derived programs.

Second, heuristics for time comparisons affect the effectiveness of the transformations and hence the degree of incrementality that can be discovered. At several places in the transformations, especially in the $\mathcal{S}_t$ transformation, we run into situations where we need to decide whether $t(e_x) \leq_P t(e_r)$ or $t(e_x) \leqq_P t(e_r)$, and if so, replace $e_x$ with $e_r$. A quick heuristic may simply give negative answers to most comparisons. Thus, no replacement would occur in these situations and there would be less use of cached results. So good timing heuristics are important for the derivation of a higher degree of incrementality. Note, however, that a more accurate heuristic may take a long time, making the transformation inefficient. Therefore, fast heuristics are also important for the efficiency of the transformation.

On the other hand, these issues are of concern because the problems are intractable in the general setting. If we give up generality, a lot of work can be done to frame the classes of domain problems that need to be solved and/or the features of programming languages that can be used for coding the domain problems. Then we can study efficient algorithms for these frameworks without sacrificing the degree of incrementality that can be derived. Of course, relating this potential work with existing work in incremental computation is itself another important task.

**Applicability and limitation.** The transformational approach presented here can only derive a set $F'$ of function definitions such that in the set $F'$, $f_0'$ is a differentiated version of $f_0$, and each function other than $f_0$ in the original set $F$ is either eliminated in Step 4 (if all function calls to them are either unfolded in Step 1 or substituted out in Step 3) or kept unchanged in $F'$.

The reason for this limitation is as follows: Basically, $\mathcal{G}_o$ of Step 1 naively decides whether a function application is unfolded or not using rule $f_e$. For any function, if rule $f_e$ does not apply at one of its calls and this function application does not have a substitution candidate, then the definition of this function is needed as a whole and is kept unchanged henceforth. During the whole transformation, the function $f_0$ is derived towards its differentiated version $f_0'$ under the change $\oplus$. But this is only made possible through the introduction of the function $f_0^*$ with the application of transformation $\mathcal{D}_t$. For any function other than $f_0$, say $g$, if $g$ is defined recursively and there exists some function applications of $g$ that do not have substitution candidates, then without an application of transformation $\mathcal{D}_t$ on $g$, its definition has to be kept unchanged for recursive calls to itself. In summary, the limitation of the approach above is the result of the naive decision about function unfoldings by the rule $f_e$ of $\mathcal{G}_o$ as well as the naive separation and sequential execution of the four basic steps as presented above.

**Overcoming the limitation.** Suppose we are given a function $f_0$ in the context of a set $F$ of function definitions and an input change operation $\oplus$, and we want to derive a differentiated version $f_0'$ of $f_0$ in the context of a set $F'$ of function definitions. In general, any function $f$ in $F$ may have its differentiated versions in $F'$ from the derivation. These differentiated versions may also be incremental versions of $f$ under certain change operations caused by $\oplus$. In order to obtain such incrementality in the derived programs, we need to modify the basic unfolding rules and combine the four basic steps in a more organic way.

We propose the following idea of *focusing* and *switching*: During the transformation, the focus is on the function application that is presently transformed, say the function is $f$, and the goal is to apply the four-step transformation to this application of $f$. If there is no function call in $f$, then, all four steps can be applied subsequently and we are done with this application of $f$. Otherwise, if $f$ has a function call at some point, say to the function $g$, then we first apply the four-step transformation on the part of $f$ before the call point to $g$, then we switch attention to the application of $g$, which means that the goal now is to focus on this application of $g$ and apply the four-step transformation on it. This procedure is applied recursively to all function calls. When we are done with this application of $g$, we switch our focus back from $g$ to $f$ to continue the transformation on $f$. We apply the four-step transformation on the part of of $f$ after the call point to $g$ until we meet another function call or we finish the transformation on this application of $f$. Of course, we start with function $f_0$.

To implement the above approach, we propose that the four-step transformation presented in Section 5 be modified as follows: Associate with each function $f$ in $F$ a list of its differentiated versions, initially empty. These differentiated versions are introduced at some calls to $f$ that depend on both $x$ and $y$. The transformation works as follows: At a call $f(e_1, ..., e_n)$ that depends on both $x$ and $y$, if we can not use any of the differentiated versions of $f$ currently in $f$'s list (in a fashion similar to the way that $\mathcal{D}_t$ uses its rule $f_s$ to replace a call to $f_0$ with a call to $f_0^*$), then we introduce a new function definition $f^*$ for this call (in a fashion similar to the way Step 2 introduces $f_0^*$), add $f^*$ to the list of differentiated versions of $f$, and proceed to transform on the unfolded function body of this $f^*$. Of course, more special care is needed to deal with recursive function applications.

A thorough presentation of the focusing-and-switching technique sketched here is forthcoming.

**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[3]. 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 approach presented above, we can obtain a differentiated version $\hat{f}'$ of $f$ 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

[3] From this paragraph on, for notational convenience, we use $f$ to denote $f_0$.

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 Step 0 in deriving incremental programs. 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.

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 $x \equiv \langle x_1, x_2 \rangle$, 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:

$$f(x_1, x_2) = \quad len(x_1) * len(x_2)$$

where $len$ is a function that returns the length of a list.

Let $y \equiv \langle y_1, y_2 \rangle$ 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 our derivation approach, since $f(cons(y_1, x_1), cons(y_2, x_2))$ is equivalent to $(len(x_1)+1) * (len(x_2)+1)$ and thus to $len(x_1) * len(x_2) + len(x_1) + len(x_2) + 1$, and the value of $len(x_1) * len(x_2)$ is $r$, we obtain a differentiated 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 $\hat{r}_1$:

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

where *triple* is a constructor with corresponding selectors $1st$, $2nd$, and $3rd$. It is easy to see that using our derivation approach, we can obtain a differentiated 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:

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

where *pair* is a constructor with corresponding selectors $fst$ and $snd$. It is easy to see that using our derivation approach, we can obtain a differentiated 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 get speedup. Although there is no universally applicable data structure, some apply to a broad class of problems [9]. 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.

## 8 Related Work

A comprehensive guide to the literature on incremental computation has appeared in [28]. Despite the relatively diverse categories discussed in [28], most of the work can be divided into three classes.

The first class includes particular incremental algorithms that deal with particular input changes to particular problems. Examples are incremental parsing[14, 17], incremental attribute evaluation [31, 45, 18], incremental data-flow analysis[34, 33], incremental circuit evaluation[2], incremental constraint solving[43, 12], *etc.* The study of dynamic algorithms, e.g., [46, 29, 11], 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 for particular problems, general evaluation frameworks are pursued for achieving incremental computation automatically, e.g., incremental computation via function caching[27], formal program manipulations like traditional partial evaluation[41, 40, 10], and incremental computation as a program abstraction[16], *etc.* Note that, when attribute grammars are used as a general description language for applications, the attribute evaluation framework can be viewed as falling into this class too. The characteristic of work in this class is that a general incremental evaluation model is run for any given application program and no explicitly incremental version of the program is derived and run autonomously by a standard evaluator. For this reason, 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[41, 26].

In the third class, explicitly incremental programs are derived from non-incremental programs by formal transformation techniques such as finite differencing[24, 47]. Typically, 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. What is not provided is a method for deriving an incremental program from a non-incremental program written in a standard programming language like Pascal or Lisp.

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[20, 24]. Their work generalizes Cocke's method of strength reduction[7] and provides a convenient framework in which to implement a host of program transformations including Earley's "iterator inversion"[8]. They develop a set of rules for differentiating set-theoretic expressions and combine these rules using a chain rule to derive inexpensive incremental programs. Such techniques are indispensable as part of an optimizing compiler for programs written in very high-level languages like SETL or APL[23, 6]. The APTS[22] program transformation system has been developed for such purposes. Our technique differs from that work 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. Using this approach, incrementalities can be *discovered* in existing programs written in standard languages.

KIDS[37, 38] is a semiautomatic program development system that aims to derive programs from high-level specifications, as is APTS. The system performs algorithm design[39], deductive inference, program simplification, partial evaluation, finite differencing, data type refinement, *etc.* 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 $F(x)$ with respect to $E(x)$ results a function $F(x,c)$ with the invariance $c = E(x)$ and changes function calls $F(U(x))$ to $F(U(x), E(U(x)))$. Simplification uses the invariance to simplify $F$ by replacing occurrences of $E(x)$ with $c$ and, in particular, aims to compute $E(U(x))$ using the invariance $c = E(x)$. As is pointed out in [37, 38], the real benefit of this optimization comes from the potential use of $c = E(x)$ in the computation of $E(U(x))$. However, no special technique is provided except the observation that distributive laws can often be applied to $E(U(x))$ yielding $U'(E(x))$ and then $U'(c)$. Note that the essence of this optimization is to compute $E$ (used by $F$) incrementally under the change $U$ (caused by $F$), which is exactly what our work is aimed at. Also note that $U(x)$ is not an accurate description of the change expression, which can actually have other free variables, say $y$. As far as we know, KIDS has no formal notion of incremental programs of $E(x)$ with respect to $U(x,y)$, as we give with our notation for $f(x)$ with respect to $x \oplus y$, and, other than the distributive laws, offers no special methods for achieving incrementalities of $E(U(x,y))$ using $c = E(x)$, as we do for computing $f(x \oplus y)$ using $r = f(x)$ with our transformations $\mathcal{G}_o$, $\mathcal{D}_t$, $\mathcal{S}_t$ and $\mathcal{G}_a$. The Munich CIP project[25] has a finite differencing strategy similar to that of KIDS.

Each of the three classes of work discussed above emphasizes a different, although important, aspect of incremental computation. But none can be regarded as a general model that subsumes the others. A general model $\mathcal{M}$ for incremental computation would be one that, given a non-incremental program $f$ written in some language $\mathcal{L}$ and an input change $\oplus$ (which is also describable in $\mathcal{L}$), derives $f'$, an incremental version of $f$ under $\oplus$. Such a model can be viewed as a general model that addresses all three approaches discussed above, 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 the particular problems being studied. Work in the second class can be regarded as a specialization of $\mathcal{M}$ to deal with a special class of input change, namely, the change operation $\oplus$ that always takes an old input $x$ and returns a new input $x'$ with part of $x$ retained in $x'$ and the rest of $x'$ being new parameters $y$. Work in the third class is exactly the study of $\mathcal{M}$ where the language $\mathcal{L}$ for describing the programs and the changes is limited to very high-level languages.

Our work attacks the problem of deriving incremental programs from non-incremental programs written in a standard 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 effective approaches can be developed to derive efficient incremental programs by combining particular program transformation techniques. By studying 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 [28] on incremental computation.

## 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] B. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark, October 1987.

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

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

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

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

[9] 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.

[10] 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.

[11] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, November 1985.

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

[13] Y. Futamura and K. Nogi. Generalized partial evaluation. In *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988. In [4].

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

[15] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, second edition edition, 1989.

[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] 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.

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

[19] Y. Liu. Theoretical results on deriving incremental programs. Unpublished notes, 1992.

[20] 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.

[21] 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.

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

[23] 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.

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

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

[26] W. Pugh. Comments on "Incremental computation via partial evaluation". Private communications, March 1991.

[27] 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.

[28] 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.

[29] M. Rauch. Fully dynamic biconnectivity in graphs. In *Proceedings of the 33rd Annual IEEE Symposium on FOCS*, pages 50–59, Pittsburgh, Pennsylvania, October 1992.

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

[31] 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.

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

[33] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: examples and counterexamples. *Science of Computer Programming*, 11(1):1–15, 1988.

[34] 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.

[35] D. Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the 3rd ESOP*, pages 361–376, Copenhagen, Denmark, May 1990. Springer-Verlag. LNCS 432.

[36] M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *ACM Transactions on Programming Languages and Systems*, 4(2):196–225, April 1982.

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

[38] D. R. Smith. KIDS - a knowledge-based software development system. In M. R. Lowry and R. D. MacCartney, 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.

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

[40] 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).

[41] 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.

[42] A. Takano. Generalized partial computation for a lazy functional language. In *Proceedings of the Symposium on PEPM*, pages 1–11, Yale University, June 1991. Published as SIGPLAN Notices, 26(9).

[43] 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.

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

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

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

[47] 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.