GOAL-DIRECTED PROGRAM TRANSFORMATION

BEN WEGBREIT
XEROX PALO ALTO RESEARCH CENTER
3333 COYOTE HILL ROAD
PALO ALTO, CALIFORNIA 94304

ABSTRACT

Program development often proceeds by transforming simple, clear programs into complex, involuted, but more efficient ones. This paper examines ways this process can be rendered more systematic. We show how analysis of program performance, partial evaluation of functions, and abstraction of recursive function definitions from recurring subgoals can be combined to yield many global transformations in a methodical fashion. Examples are drawn from compiler optimization. list processing, very high level languages, and APL execution.

KEY WORDS AND PHRASES:

program transformation, program analysis, partial evaluation. optimizing transformations, compiler optimization, analysis of programs, execution analysis, simplification, generalization, evaluation in context, very high level language, list processing, Lisp, APL.

CR CATEGORIES:

4.12, 4.22, 4.6. 5.24, 5.25

1. INTRODUCTION

Optimizing transformations provide a means for converting a clear, well-articulated but inefficient program into one with equivalent results but better performance characteristics. Certain optimizing transformations have been used for some years in the compilers for algebraic languages. A 1972 survey by Cocke and Allen [3] lists and classifies approximately twenty such transformations.

Languages in which the built-in operations act on composite objects such as arrays (e.g., APL [16]), sets (e.g., SETL [21]), or relations (e.g., VERS [12]) give rise to the need for additional classes of optimizing transformations. There are three reasons for this. (1) The substantial gulf which separates language from underlying machine makes it possible for the programmer inadvertently to write simple programs for which unoptimized execution yields poor performance. (2) The pervasive use of composite objects often makes such programs the most natural expression: easiest to write, debug, and modify. (3) Much of the appeal of these languages is due precisely to the decoupling of expression from implementation. For these reasons, the potential payoff from an appropriate set of optimizing transformations for these languages is substantial.

This paper shows how the process of program transformation can be rendered systematic. Our thesis, in brief, is that program transformation can be made goal-directed. It is possible to analyze programs so as to obtain expressions for their execution performance and the complexity of their output. Based on discrepancies between these, performance goals are established. These goals are used to direct the process of program transformation--which is carried out by *local simplification*, *partial evaluation* of recursive functions, *abstraction* of new recursive function definitions from recurring subgoals, and *generalization* of expressions as required to obtain compatible subgoals. Thus, high-level goals or intentions are used to guide and give coherence to the operations of local activities.

We expand on this brief description below.

Our intention in this paper is three-fold. (1) We show how this approach provides a framework in which many of the optimizing compiler transformations can be placed and systematized. In this framework, related transformations not commonly implemented in optimizing compilers are seen to have a natural place. (2) We show how many of the optimizing transformations for very high level languages [2] can be obtained in a straightforward way from the program. Thus, these techniques may find a role in the future processors of such very high level languages. (3) We hope to make a small step in identifying and explicating techniques by which those software-engineers who must be concerned with efficiency can carry out their design in a systematic fashion. Considerable attention has been given in recent years to the systematic development of well-structured programs, formalizing the practices of outstanding programmers. Comparable attention should, perhaps, be paid to the fundamental techniques which underlie the systematic development of high-performance programs.

2. BASIC IDEAS

One means for transforming a program into an equivalent one with better performance relies on program analysis to single out the appropriate portions of the program to be rewritten. By *program analysis*, we mean the derivation of closed-form algebraic expressions which describe execution behavior. These expressions specify the program's computation cost (e.g., execution time, amount of storage used, number of I/O requests) and the program's output characteristics (e.g., size of the result, textual source of allocated storage comprising the result, probability of the result satisfying a given predicate) as a function of input characteristics [23].

Using program analysis techniques, our transformation approach proceeds as follows: (1) Obtain some idea of the minimum computation cost required to produce the input/output mapping being realized by the program. (2) Analyze the program to determine its computation cost, and relate components of the cost to specific segments of the program text. (3) Find those program segments whose computation costs are not accounted for in the estimates of minimum cost. These segments are potential sources of computational waste and are therefore designated as *targets for simplification.* (4) If the targets so designated contain multiple possibilities for simplification, focus attention on the program segments having the greatest analyzed cost: Insofar as the program can be transformed to realize a significant performance improvement, it must be by simplifying these segments.

The central idea we wish to present is that this approach can be rendered systematic and thus has a place in programming methodology as well as serving as a basis for mechanical program transformation.

2.1. NOTATION

Several of our example programs will be written in a syntactic variant of Lisp, using the following notation:

- The empty list is denoted by nil.

- Cons(x,y) constructs a list in which the first element is x and y is the list of all elements except the first.

- H, read as "Head", is a prefix operator which extracts the first element from a non-empty list; to avoid parenthesis clutter, the argument to H is not parenthesized, e.g. H Cons(x,y) = x.

- T, read as "Tail", is a prefix operator which extracts all elements except the first from a non-empty list; T Cons(x,y) = y.

- Null(x) is defined as x=nil.
- Conditional expressions are written as
  if p_1 **then** e_1
  else if p_2 **then** e_2
  ...
  else e_n
- Function definitions are written as *fname(parameter$_1$, ... ,parameter$_k$)* <= *definingform*

2.2. EXAMPLE--CONCATENATION OF THREE LISTS

Given the following definition of Append,

```
A(x,y)  <=  if Null(x) then y
            else Cons(Hx, A(Tx,y))
```

consider appending x to y to z by the expression A(A(x,y), z).
Analysis:
Program analysis shows that the execution cost is proportional to 2·|x|+|y| where |x| is the length of the list x. It is useful to differentiate between the inner and outer calls on Append. Let these be denoted by A_1, and A_2 and let the associated calls on Cons be $Cons_1$ and $Cons_2$. Then program analysis shows that $A_1(A_2(x,y), z)$ executes |x| calls on $Cons_2$ and |x|+|y| calls on $Cons_1$. Analysis of the *output* of $A_1(A_2(x,y), z)$ shows that its length is |x|+|y|+|z|. Differentiating the source of the Cons-cells which comprise the output, analysis shows that |x|+|y| calls come from $Cons_1$ and that |z| calls come from the input z. Comparing the execution costs (internal work) to the output (accountable work), it is seen that the |x| calls on $Cons_2$ represent wasted effort. These calls are being executed internally but cannot be accounted for in the output. The task of transformation is to rewrite the program so as to remove the calls on $Cons_2$. We refer to $Cons_2$ as the *target for simplification*.

Transformation:
To remove a Cons, we can apply the *local simplification rules*

$$H\ Cons(\alpha,\beta) \rightarrow \alpha$$
$$T\ Cons(\alpha,\beta) \rightarrow \beta$$

To obtain an opportunity to apply these, we expand the definition of Append to the point where H and T appearing in the program can be applied to $Cons_2$. Start with:

(2.1) $A_1(A_2(x,y), z)$

Expand $A_2(x,y)$, i.e., replace the call by an instantiated body, resulting in:

A_1(**if** Null(x) **then** y
else $Cons_2(Hx, A_2(Tx,y))$, z)

Distribute the conditional, i.e., bring the conditional expression from the argument position to outside the call on A_1, so that A_1 is applied to the result of each conditional clause. This yields:

if Null(x) **then** $A_1(y,z)$
else $A_1(Cons_2(Hx, A_2(Tx,y)), z)$

Expand the second call on A_1 and then *simplify*. We call this *partial evaluation*, and the steps in this case are as follows: Consider $A_1(Cons_2(Hx, A_2(Tx,y)), z)$. Let γ denote $Cons_2(Hx, A_2(Tx,y))$, so we are considering $A_1(\gamma,z)$. Expanding A_1 results in **if** Null(γ) **then** z **else** $Cons_1(H\gamma, A_1(T\gamma,z))$. Since γ is not Null, it follows that $A_1(\gamma,z) = Cons_1(H\gamma, A_1(T\gamma,z))$. Further, we can apply the *local simplification rules* for Cons to γ, so that $H\gamma \rightarrow Hx$ and $T\gamma \rightarrow A_2(Tx,y)$. Thus, we have removed one instance of $Cons_2$--a step toward our goal. We have $A_1(\gamma,z) = Cons_1(Hx, A_1(A_2(Tx,y), z))$. Thus, (2.1) has been partially evaluated to yield:

(2.2) **if** Null(x) **then** $A_1(y,z)$
else $Cons_1(Hx, A_1(A_2(Tx,y), z))$

Because one $Cons_2$ is absent, the evaluation of (2.2) requires, in the case of a non-Null x, one fewer $Cons_2$ than the evaluation of (2.1). Thus, (2.2) is a slightly improved way of computing (2.1).

Whenever we are confronted with evaluating an expression of the form (2.1), it is desirable to use (2.2) in its place. Observe that the form (2.1) appears as a subexpression of (2.2) with the *substitution* of Tx for x, i.e., as $A_1(A_2(Tx,y), z)$. We have identified a subproblem, $A_1(A_2(Tx,y), z)$ which is identical in form to a more global problem, $A_1(A_2(x,y), z)$. We call this identification *subgoal abstraction* and use it to define a new recursive function in which the performance improvement of (2.2) is systematically achieved. We introduce F(x,y,z) to stand for $A_1(A_2(x,y), z)$. Then (2.1) and (2.2) become

```
(2.1')  F(x,y,z)
(2.2')  if Null(x) then A_1(y,z)
        else Cons_1(Hx, F(Tx,y,z))
```

Ignoring subscripts, we have

```
F(x,y,z)  <=
if Null(x) then A(y,z)
else Cons(Hx, F(Tx,y,z))
```

as an executable definition of F. Since the above derivation preserves correctness, it follows that A(A(x,y), z) = F(x,y,z). Analysis shows that the execution cost for F is linear in |x|+|y| and, in particular, the number of Cons executed is |x|+|y|. Thus, the goal has been attained.

2.3. FURTHER EXAMPLES

Some additional examples will illustrate that this approach can yield interesting results. We show the problem, analysis, and final program--omitting derivations.

2.3.1 The following program compares two arrays *B[1:n]* and *C[1:n]* and sets the Boolean variable *same* to **true** if they are pairwise equal

```
j←0; same←true;
while j≠n
do (j←j+1; same ← same ∧ B[j]=C[j])
return same
```

Analysis shows that this requires n steps and that the result is a Boolean scalar. It is therefore possible that the scalar could be computed in fewer steps. The entire loop is the target for simplification. The ∧ operator has the *local simplification rule*

x ∧ y → **if** x **then** y **else false**

which avoids computing y when the value of x determines the result. We write the above loop as a recursive function, and apply our transformation method using the local simplification rule. We obtain a new function which, when put into iterative form, is

```
j←0; same←true;
while j≠n ∧ same
do (j←j+1; same ← B[j]=C[j]);
return same;
```

This requires *n* steps in the worst case and 1 step in the best case. If the probability that B[j]=C[j] is β, program analysis shows that the average number of steps is $(1-\beta^n)/(1-\beta)$. Observe that this is bounded from above by $\min(n, 1/(1-\beta))$.

2.3.2 The following APL[1]"one-liner" has the value 1 if n is prime and 0 otherwise: 2=+/0=(ιn)|n. (Reading from right to left, this may be rendered as: Consider n and the sequence 1,2,...,n; obtain the remainders after dividing the elements of the sequence into n; find the elements with 0 remainder; count their occurrences; and test the count to see if it is 2. If so, then n is prime.) Analysis shows that this constructs three temporary arrays each of length n, but that the output is a single scalar. Transformation takes place in three stages each of which eliminates one unneeded array. The final result is 2=H(1,n) where

```
H(k,n) <=
if k>n then 0
else if (n mod k)=0 then 1+H(k+1,n)
else H(k+1,n)
```

which constructs no temporary arrays: It simply counts the number of times the remainder of n divided by k is 0, for k from 1 to n.

3. TECHNIQUES

The preceding derivations employ three techniques: *program analysis*, *partial evaluation*, and *subgoal abstraction*. These activities may be roughly characterized as: analyzing the program's resource expenditure and output to find appropriate targets for simplification; rewriting portions of the program to realize a performance improvement on the *first* execution of the program's loops; recognizing subproblems and using these to form recursive programs such that the performance improvement is attained on *every* execution of the program's loops. We now examine these in greater detail.

3.1. PROGRAM ANALYSIS

Program analysis obtains closed-form expressions which describe execution behavior as a function of input characteristics, e.g., worst-case execution time as a function of the input-length. A system which automatically carries out program analysis for simple programs is discussed in [23]. Techniques, implementation issues, and limitations are discussed there. In the interest of brevity, we confine our discussion here to outlining the sorts of analysis which can be produced in this way.

Analysis is carried out for three cases--best, worst, and average (under some assumptions about input distributions). It is useful to write the closed-form expressions describing execution cost in a partially factored form, separating out the dependence on input characteristics from the dependence on machine implementation. For example, the execution time of Append is written as $c_0+c_1 \cdot n$ where n is the length of the first argument to Append and c_0, c_1 are implementation constants. The implementation constants are written as linear arithmetic expressions of the form $r_1 \cdot p_1 + \ldots + r_k \cdot p_k$ where the r_i's are rational numbers and the p_k's denote costs of executing primitive operations. For the case of Append,

$$c_0 = fncall + 2 \cdot vref + null$$
$$c_1 = car + cdr + cons + 3 \cdot vref + fncall$$

The lower case spelling of a primitive operation stands for that operation; *fncall* denotes the action required to invoke a non-primitive operation; *vref* denotes access to a variable.

When analysis is used for the purpose of directing program transformations, it is useful to distinguish the source of a primitive operation based on the defining function in which it appears. Thus, $cons_A$ denotes those Cons-cells created by the execution of the Cons operation textually inside Append (c.f., the definition of Append, Section 2.2). To do this, we distinguish the symbolic "costs" of elementary operations based on their textual position: a Cons textually inside function A is treated, for the purpose of analysis, as if it were distinct from a Cons textually inside function B. In the case where a defined function, F, appears more than once in the expression to be transformed, it is useful, for the purpose of analysis, to treat each occurrence of F in the expression as if it were a distinct function. Suppose F is recursive and the expression is F(G(x), F(y, z)). This is treated as if it were $F_1(G(x), F_2(y, z))$. All calls to F from F_1 (and from functions called by F_1) are treated as calls on F_1. Similarly, F_2 is treated as calling F_2 recursively. A function call "within" the inner F is therefore labeled fncall_{F_2}. Thus, we can distinguish the cost of executing the inner F from the cost of executing the outer F (once its arguments have been evaluated).

This representation of execution was chosen for its generality. From it, we can obtain the answer to specific questions by assigning appropriate values to the primitive operation symbols. For example, to obtain total execution time on a given machine, let each primitive symbol stand for the execution time of the corresponding operation. To obtain the total number of Cons cells created, let cons_i (for all i) be 1 and let the other elementary operation symbols be 0. In analogous fashion, we may obtain the number of *fncall*'s (function calls) out of a specific function, or the computation time spent within a specific group of functions.

Several other sorts of analysis are also of interest. One class is *output analysis*, e.g., determining the length of a function's output if an array or list, or its range if a scalar. In the case of list length, it is useful to distinguish the textual source of Cons-cells which occur in the output, e.g. an output list might be of length |x|, but this is more usefully expressed as $1 \cdot \mathit{cons}_R + (|x|-1) \cdot \mathit{cons}_A$. Such expressions are obtained by computing the output list length as a polynomial, in a way similar to the execution cost analysis. Another class of analysis is concerned with internal operations, e.g., the probability of a Boolean-valued procedure returning the value true; such analysis is required as auxiliary data in computing computation cost. These are obtained, and the results are expressed using methods similar to those for execution cost.

The results of analysis are employed in two ways. The first method exploits discrepancies between the complexity of a program's output and the computation cost of obtaining that output. Such discrepancies represent wasted work and the corresponding program components are therefore targets for simplification. When this singles out a specific portion of the program, it provides a sharp criterion.

In some cases, this may not be specific enough; to further narrow the target, a second method is employed. This consists of hypothesizing plausible goals on the basis of current performance and using the results of analysis to determine the program components causing discrepancies between these goals and the computation cost. For example, if a program runs in n^2 steps, a plausible goal is n steps and the target for simplification is narrowed to those sub-regions responsible for the n^2 behavior. If the goal of n steps is attained, then a second, more stringent, goal will be tried, i.e., fewer steps in the best case. Thus, at each step, the plausible goal is taken to be the next significant performance level. This selects as the *target for simplification* those program components which must be simplified if that performance level can be attained by our transformation method. We stop when the computation cost is commensurate with the complexity of the generated output, or when transformations fail to attain a goal.

3.2. PARTIAL EVALUATION

Partial evaluation consists of rewriting portions of a function to exploit knowledge of its arguments. In the simplest case, a partial evaluator takes a function P of n formal parameters $x_1,...,x_n$ along with values $a_1,...,a_k$ for the first k actual arguments and constructs a new function P' such that $P'(b_{k+1},...,b_n) = P(a_1,...,a_k, b_{k+1},...,b_n)$ for all sets of b_j. That is, P' is a variant of P, specialized to the case where the first k parameters are known constants. To the extent that P' is a simplified version of P, its computation cost is smaller.

Here we employ an extension of this idea: Rather than knowing the actual values of certain formal parameters, we know the function which constructs them. Let $Q_1,...,Q_k$ be defined functions (of one argument, for simplicity). Our partial evaluator will typically take an expression such as $P(Q_1(y_1),...,Q_k(y_k), y_{k+1},...,y_n)$ along with the definitions of $Q_1,...,Q_k$ and construct a new function P' such that $P'(b_1,...,b_k, b_{k+1},...,b_n) = P(Q_1(b_1),...,Q_k(b_k), b_{k+1},...,b_n)$. That is, P' is a variant of P, specialized to the case where the first k parameters are known to be computed by $Q_1,...,Q_k$. To the extent that P' combines the computations of the first k arguments with each other and with the execution of P, P' runs faster than the sequential execution of $Q_1,...,Q_k$ followed by P.

It is useful to distinguish four facets of partial evaluation: *expanding function definitions*, *distributing conditionals*, *simplifying*, and *evaluating in context*.

Expansion replaces a function call by a suitably instantiated copy of the function definition. If P has formal parameters $x_1,...,x_n$, then the complete expansion of $P(Q_1(y_1),...,Q_k(y_k), y_{k+1},...,y_n)$ is obtained as follows:

(i) Let R_i be the result of instantiating the body of Q_i with argument y_i, for $i=1,...,k$.

(ii) In the body of P, substitute R_i for x_i $(i=1,...,k)$ and y_j for x_j $(j=k+1,...,n)$.

It is undesirable to carry out an expansion of all defined functions (even to one level), since this blows up the program size and makes difficult the recognition of recurring subexpressions needed for subgoal abstraction. Instead, we adopt the following method: *defined functions are expanded only if they can be reduced to constants or so far as necessary to expose the target for simplification to local simplification rules.* This resolves into three expansion criteria:

(E1) The function call which contains the target is expanded.

(E2) The surrounding function is expanded as necessary to obtain surrounding context for the local simplification rule.

(E3) If all the arguments (and free variables) of a function call are constant then the function call is expanded.

Thus, to eliminate a Cons by the local simplification rule H $\text{Cons}(\alpha,\beta) \rightarrow \alpha$, we expand the function call which contains that Cons (criterion E1) and also expand the surrounding function call so as to obtain an H operation which may be applied to the Cons (criterion E2).

Typically, a function expanded in this way contains conditional expressions, e.g., controlling recursion. Suppose such an expanded function occurs as the argument to an outer function, say F. We then have an expression such as

```
F(α, if p₁ then e₁
     else if p₂ then e₂
     ...
     else eₙ, γ)
```

where α, the conditional expression, and γ are the first, second, and third arguments to F. *Distributing the conditional* consists of bringing the conditional tests out of the argument position to the surrounding scope. This yields a conditional expression in which F is applied to the result of each conditional clause:

```
if p₁ then F(α,e₁,γ)
else if p₂ then F(α,e₂,γ)
...
else F(α,eₙ,γ)
```

This creates new, specialized function calls--$F(\alpha,e_1,\gamma)$, $F(\alpha,e_2,\gamma)$, ... ,$F(\alpha,e_n,\gamma)$. Because they are specialized, their further partial evaluation may lead to simplifications.

To illustrate the steps of partial evaluation, we return to the example of Section 2.2 and consider $A_1(A_2(x,y), z)$. Since the Cons in A_2 is chosen as the target for simplification, A_2 is selected for expansion by criterion E1. From the definition of A_2, we obtain:

A_1(**if** Null(x) **then** y
  else $Cons_2$(Hx, A_2(Tx,y)), z)

Distributing the conditionals, we obtain

if Null(x) **then** A_1(y,z)
else A_1($Cons_2$(Hx, A_2(Tx,y)), z)

The target for simplification, $Cons_2$, is now in an argument position. Expansion criterion E2 selects the surrounding function, A_1, for expansion. Thus, $A_1(Cons_2(Hx, A_2(Tx,y)), z)$ is expanded into

if Null($Cons_2$(Hx, A_2(Tx,y))) **then** z
else $Cons_1$(H $Cons_2$(Hx, A_2(Tx,y)),
        A_1(T $Cons_2$(Hx, A_2(Tx,y)), z))

which is ripe for simplification.

In regard to simplification, we take the following operational point of view. Let $|\alpha|$ be the cost[2] of computing expression α. Then expression δ is *simpler* than expression α if $|\delta| < |\alpha|$ for some assignment of values to variables and $|\delta| \leq |\alpha|$ for all assignments of values to variables. By this criteria, the following are simplifications:

$p_1 \wedge p_2 \rightarrow$ **if** p_1 **then** p_2 **else false**
H Cons(e_1,e_2) $\rightarrow e_1$
Null(Cons(e_1,e_2)) $\rightarrow$ **false**
if false then e_1 **else** $e_2 \rightarrow e_2$
if p **then** e **else** e $\rightarrow$ e

Applying these sorts of local simplification rules to the above expression results in

$Cons_1$(Hx, A_1(A_2(Tx,y), z))

Thus, the partial evaluation of $A_1(A_2(x,y), z)$ yields **if** Null(x) **then** A_1(y,z) **else** $Cons_1$(Hx, A_1(A_2(Tx,y), z)). We denote this as

A_1(A_2(x,y), z) $\simeq$
if Null(x) **then** A_1(y,z)
else $Cons_1$(Hx, A_1(A_2(Tx,y),z))

That is, $\alpha \simeq \delta$ means that α can be partially evaluated to yield δ and, thus, if the computation α terminates then the computation δ terminates and yields the same answer.

An additional facet of partial evaluation, not illustrated by the above example, is *evaluation in context*. This consists of using information derived from conditional expressions to assist in local simplification. Evaluation in context arises when an expression embedded within a conditional is selected for expansion--all the predicates on test branches leading to that expression are known to be true. Consider, for example, the expression b=M(b,y) where M computes the maximum of the set $\{b\} \cup y$, as follows:

M(b,y) <=
if Null(y) **then** b
else if b<Hy **then** M(Hy,Ty)
else M(b,Ty)

Expanding M(b,y) in b=M(b,y) and simplifying yields

if Null(y) **then true**
else if b<Hy **then** b=M(Hy,Ty)
else b=M(b,Ty)

where b=b has been simplified to **true**. Next, we expand the expression b=M(Hy,Ty). In so doing, we can use results of the tests leading to this point, so we know: $\sim$Null(y) $\wedge$ b<Hy. We call these tests *context conditions for the expansion of b=M(Hy,Ty)*. Expanding this in context yields

```
if Null(y) then false
else if Hy<HTy then b=M(HTy,TTy)
else b=M(Hy,TTy)
```

This follows because value of the first conditional expression, b=Hy, can be simplified to **false** in the context b<Hy.

To express the use of context conditions in partial evaluation, we extend the above notation and write

α {**in context** p} $\simeq$ δ

Similarly, we extend the definition of *simpler* to include context conditions: δ is *simpler* than α in context p if $|\delta|<|\alpha|$ for some assignment of values to variables which satisfies p and $|\delta|\leq|\alpha|$ for all assignments of values to variables which satisfy p.

3.3. SUBGOAL ABSTRACTION

Subgoal abstraction consists of identifying subproblems which are identical in form to more global problems and using this identification to construct the definitions of recursive functions. Suppose that

$\alpha \simeq$ **if** p **then** e **else** $\Gamma(\beta)$

when α and β are expressions and Γ is an expression involving β. Suppose that there is some substitution[3] Θ which carries α into β, i.e., $\alpha\Theta=\beta$. We say that α is the *goal*, β is the *subgoal*, and β is a substitution instance of α. Rewriting the above,

$\alpha \simeq$ **if** p **then** e **else** $\Gamma(\alpha\Theta)$.

Let ξ be the set of variables in α. Let $F(\xi)$ be defined by

(3.1) $F(\xi) \Leftarrow$ **if** p **then** e **else** $\Gamma(F(\xi\Theta))$

It follows that $\alpha \simeq F(\xi)$. That is, if the computation α terminates then the computation $F(\xi)$ terminates and their values are equal.

The reason for introducing such a definition is to obtain a performance improvement. Hence, we construct such a definition F and use it to compute α only when F is computationally *simpler* than α. If $|\alpha| \geq |$**if** p **then** e **else** $\Gamma(\beta)|$ for all assignments of values to variables, then $|\alpha| \geq |F(\xi)|$; if also $|\alpha| > |$**if** p **then** e **else** $\Gamma(\beta)|$ for some assignment of values to variables, then $|\alpha| > |F(\xi)|$ for that assignment; thus, $F(\xi)$ is *simpler* than α. In the general case, comparing the costs of α with "**if** p **then** e **else** $\Gamma(\beta)$" is carried out by analyzing the computational costs of the latter expression. Since some of its constituents have been previously analyzed and since the analysis technique reuses the analysis of constituents when dealing with a larger expression in which they are contained, such analysis is generally easier than the original analysis of α.

In a commonly occurring case, a cost comparison can be carried out more directly. To explain this, it is necessary to first clarify the relation between partial evaluation and computation cost. If $\alpha \simeq \delta$, then δ must terminate whenever α terminates, but there is no assurance that δ's cost is less than α's. In obtaining δ from α, two counterposing phenomena are at work: (1) Local simplification tends to make δ simpler than α. Thus if α contains "if p' **then** e' else e''" and this is simplified to e', then δ will be simpler by the cost of p' plus the cost of an **if**. (2) Duplicating actual arguments when expanding functions tends to make δ more costly than α when the arguments are complex expressions and must be executed more than once. Thus, if α contains F(Cons(G(x)+1, x)) and if F is expanded and simplified to "**if** P(G(x)) **then** G(x)+2 **else** Cons(x, G(x))" then δ will be more costly since it executes G(x) twice in the expanded body of F rather than once as the argument to F.[4] Because the combined effect of these two phenomena may be complex, the relation of α to δ is determined, in the most general case, by analyzing δ.

Often, however, the second phenomenon does not occur. After local simplification, the arguments to an expanded function appear at most once on each execution path through the expanded portion of the function body. In such cases, the cost of δ differs from the cost of α only insofar as local simplifications may have taken place. If there have, in fact, been any simplifications, then δ is necessarily simpler than α. If a new function F is defined as specified in (3.1), then F is known to be a better way of computing the same result than α computes.

It is important to appreciate that the *goal* for subgoal abstraction need not be the original top-level problem. In general, the goal α will be some subexpression which arises in the course of partial evaluation, and the subgoal β will be the matching subexpression. Let p be the *context condition* for the evaluation of α. The criteria for subgoal abstraction are:

(SA1) α {**in context** p} $\approx \Delta(\beta)$
(SA2) there is a substitution θ such that $\alpha\theta=\beta$
(SA3) $\Delta(\beta)$ is *simpler* than α in context p
(SA4) pθ is true at the points in Δ where β appears

The last of these criteria may require further explanation: In order for a subgoal β to be identical in form to a goal α, the context conditions used in partially evaluating α must be true at each appearance of β in Δ. We refer to this as *checking the context conditions.*

An example will illustrate the importance these considerations. Let M be the maximum of {c}∪y defined

```
M(c,y) <=
if Null(y) then c
else if c<Hy then M(Hy,Ty)
else M(c,Ty)
```

Evaluation in context shows that

```
b=M(c,y) {in context b<c}  ≈
if Null(y) then false
else if c<Hy then b=M(Hy,Ty)
else b=M(c,Ty)
```

We match b=M(Hy,Ty) against b=M(c,y) with the substitution θ={Hy/c, Ty/y}. Checking the context condition b<c under the substitution θ requires showing that b<Hy at the expression b=M(Hy,Ty) which is true since b<c $\wedge$ c<Hy implies b<Hy. Similarly, we match b=M(c,Ty) against b=M(c,y) with the substitution {Ty/y}; the context condition, b<c, is easy to check since it is unchanged by this substitution. Thus, subgoal abstraction can be carried out. We let F(b,c,y) stand for b=M(c,y) {**in context** b<c} meaning that F is defined only when its first argument is less than its second. We have

```
F(b,c,y) <= if Null(y) then false
            else if c<Hy then F(b,Hy,Ty)
            else F(b,c,Ty)
```

Simplifying this, F(b,c,y) ≈ **false** since the only way F can terminate is by returning **false**. Thus b=M(c,y) {**in context** b<c} ≈ **false**. This may be read as b<c ⊃ b≠maximum({c}∪y). Stated as a theorem, this is unremarkable. However, in program optimization, one does not have explicit statements of desirable theorems as input. That the transformation method obtains this directly from the expression and definitions is of interest.

3.4. GENERALIZATION

A somewhat subtle point in subgoal abstraction is the way in which argument positions are *generalized.* We first consider the generalization of constant arguments. Consider an expression of the form F(0, G(x,y)) and suppose that the target for simplification is inside G, so that G is to be expanded followed by a partial evaluation of F. It would not be desirable to uniformly replace all constants by new individual variables--here, replacing 0 by some new z_i--since F might be simplified in its partial evaluation in the case that its first argument is known to be 0. An extreme case would be where F(0, G(x,y)) is partially evaluated to a constant. On the other hand, there are cases in which matching is blocked by the presence of different constants in the same argument position of a goal expression and a subgoal. For example, partial evaluation of F(0, G(x,y)) might lead to an expression in which F(Hx, G(Tx,Ty)) appears. Hx cannot be matched against 0, so subgoal abstraction is inhibited.

When a match fails because of the presence of a constant in the goal, *generalization* is employed: Argument positions which are constant in the goal and different in the subgoal are replaced by new individual variables. Then the generalized goal expression is partially evaluated. If the result is similar to the previous result, the match will succeed. For example, suppose F(0, G(x,y)) is generalized to F(z_1, G(x,y)) and that partial evaluation of this leads to an expression in which F(z_1+Hx, G(Tx,Ty)) appears. The match is successful with the substitution {z_1+Hx/z_1, Tx/x, Ty/y}.

A similar situation is caused by the multiple appearance of an individual variable in a goal expression. Consider, for example, E(I(k,n), k), where the variable k appears twice. It would be undesirable to adopt the uniform policy of generalizing this to E(I(z_1,n), z_2) and then attempting to optimize this, for it might be the case that E(I(z_1,n), z_2) has a simple partial evaluation and subgoal abstraction for z_1=z_2 but not otherwise. It would be equally undesirable to accept only subgoal matches of the form E(I(e_1,e_2), e_1) since it might be the case that no such subgoals occur. Suppose, for example, that the first "near match" in the partial evaluation of E(I(k,n), k) was the subexpression E(I(k+1,n), k) and that all subsequent near matches had the form E(I(k+j,n), k) for j=2,3,... We adopt the same solution here as for constants: The matching process constructs a list of substitutions. If a match fails because the substitutions for a variable are incompatible, i.e., e_1/x and e_2/x where $e_1 \neq e_2$, then the conflicting appearances of the variable in the goal expression are generalized to distinct individual variables.

In summary, the strategy for generalization is to delay so doing until required by subgoal abstraction, to generalize as dictated by the match, and then to determine the effect of this generalization by repeating the partial evaluation.

An example will show the importance of *generalization* and how generalization interacts with *evaluation in context*. Consider, for example, the Fibonacci function.

F(n) <= **if** n=0 ∨ n=1 **then** 1
          else F(n-1)+F(n-2)

Analysis shows that F(n) takes exponential time. We start with the right hand side of the definition.

if n=0 ∨ n=1 **then** 1 **else** F(n-1)+F(n-2)

Analysis shows that the cost of F(n-1) is the largest component, so it is selected for partial evaluation. Expansion uses the context condition n≠0 ∧ n≠1. After distributing conditionals and simplifying, the result is

if n=0 ∨ n=1 **then** 1
else if n=2 **then** 2
else 2·F(n-2)+F(n-3)

Taking F(n-1)+F(n-2) as the goal and 2·F(n-2)+F(n-3) as the subgoal, we attempt to match. This fails, since the constant 2 does not match the implicit constant 1 in 1·F(n-1). We generalize the goal to k·F(n-1)+F(n-2). Partially evaluating k·F(n-1)+F(n-2) {**in context** n≠0 ∧ n≠1} we get

if n=2 **then** k+1 **else** (k+1)·F(n-2)+k·F(n-3)

Again, the match fails due to a constant argument in the goal and again we generalize. Taking k·F(n-1) + j·F(n-2) {**in context** n≠0 ∧ n≠1} as the goal and partially evaluating, we get

if n=2 **then** k+j **else** (k+j)·F(n-2)+k·F(n-3)

The match is now successful with the substitution {k+j/k, k/j, n-1/n}. Under this substitution, the context conditions are true at the calls on F, so we can carry out subgoal abstraction. Let G(k,j,n) stand for k·F(n-1)+j·F(n-2) {in context $n \neq 0 \wedge n \neq 1$}. Then define

```
G(k,j,n) <= if n=2 then k+j
              else G(k+j, k, n-1)
```

Thus, we have that F(n) can be computed as

```
if n=0 ∨ n=1 then 1 else G(1,1,n)
```

which executes in linear time.

4. FURTHER EXAMPLES

In this section, we present three examples to illustrate particular points of interest. Each example is labeled with the points it illustrates. We confine our exposition to a statement of the original program, the key points of the processing, and the result. Omitted steps are either straightforward, or repetitions of points illustrated elsewhere.

4.1. TREATMENT OF WHILE LOOPS, SPECIAL-PURPOSE LOCAL SIMPLIFICATION RULES, EVALUATION IN CONTEXT, CHECKING CONTEXT CONDITIONS IN SUBGOAL ABSTRACTION

Consider the while loop

(4.1) **while** P **do if** Q **then** R **else** S

where the value of Q is unaffected by R and S, i.e., the test is taken in the same direction on the i+1st iteration as on the ith. Let ξ be a vector of the variables appearing in P,Q,R, or S. It turns out that processing is simplified if such iterative programs are converted to functional form, viz. F(ξ), where

```
(4.2) F(ξ) <=
          if ~Pξ then ξ
          else if Qξ then FRξ
          else FSξ
```

provided that P and Q have no side effects. Let the number of times the loop is executed be n. The contribution of Q to the computation cost is |Q|·n. Suppose analysis shows this is large so that Q becomes the target for simplification. Since the value of Q is unchanged by S and R; we have two special-purpose *local simplification rules.*

(4.3) QRξ → Qξ and QSξ → Qξ

Transformation proceeds as follows: Start with (4.2) and expand the functions which contain the target for simplification--the inner calls on F.

```
F(ξ) ≈ if ~Pξ then ξ
         else if Qξ then
             (if ~PRξ then Rξ
              else if QRξ then FRRξ
              else FSRξ)
         else (if ~PSξ then Sξ
               else if QSξ then FRSξ
               else FSSξ)
```

We now proceed to *evaluate in context* and apply local simplification rules. Using (4.3), QRξ=true in the context Qξ=true while QSξ=false in the context Qξ=false. Thus

```
F(ξ) ≈ if ~Pξ then ξ
         else if Qξ then
             (if ~PRξ then Rξ else FRRξ)
         else (if ~PSξ then Sξ else FSSξ)
```

Comparing this to (4.2), we have

```
FRξ {in context Qξ}
≈ if ~PRξ then Rξ else FRRξ

FSξ {in context ~Qξ}
≈ if ~PSξ then Sξ else FSSξ
```

We can match FRRξ against FRξ with the substitution θ={Rξ/ξ}. For the match to succeed, it is also necessary to check that the *context conditions* of the goal, FRξ, are also true for the subgoal, FRRξ. Here, this requires checking that the context condition, Qξ, is true after the substitution θ at the point where FRRξ is invoked. Since $(Q\xi)\theta = QR\xi$, this is manifestly true. Matching FSSξ against FSξ is similar. Letting $F_r(\xi)$ and $F_s(\xi)$ stand for FRξ and FSξ respectively, we have the following definitions:

```
$F_r(\xi)$ <= if ~P$\xi$ then $\xi$ else $F_r$R$\xi$
$F_s(\xi)$ <= if ~P$\xi$ then $\xi$ else $F_s$S$\xi$
```

Substituting these into (4.2) and converting to iterative form, the final result is:

```
if P then {if Q
        then (R; while P do R)
        else (S; while P do S)}
```

The contribution of Q to the computation cost has been reduced from $n \cdot |Q|$ to $|Q|$, so the goal has been attained. The transformation of (4.1) to this form is generally termed *loop unswitching* [3] in compiler optimization. That this is a straightforward application of the general transformation method is of interest. The utility of carrying out the derivation in functional form should be apparent.

4.2. SUCCESSIVE TRANSFORMATIONS WITH INCREASINGLY STRINGENT PERFORMANCE GOALS, THEOREM PROVING TO ENABLE A LOCAL SIMPLIFICATION RULE

We use the following definitions for set membership, M, and set union, U, where sets are represented as non-repeating lists:

```
M(b,y) <= if Null(y) then false
          else if b=Hy then true
          else M(b,Ty)

U(x,y) <= if Null(x) then y
          else if M(Hx,y) then U(Tx,y)
          else Cons(Hx, U(Tx,y))
```

Consider the expression M(b, U(x,y)), i.e., $b \in x \cup y$ in more conventional notation. Analysis shows that this has a best case time proportional to $|x|$, and a worst-case time of $|x| \cdot |y|$, using $|x|$ Cons-cells. The result is a Boolean. Thus, the elimination of the Cons-cells is taken as the goal; in particular, the Cons in U is the target for simplification. We start with M(b, U(x,y)), *expand* the function call U which contains the target for simplification, *distribute the conditional*, and *partially evaluate* the third invocation of M--the only one which simplifies. The result is

```
if Null(x) then M(b,y)
else if M(Hx,y) then M(b, U(Tx,y))
else if b=Hx then true
else M(b, U(Tx,y))
```

We have found a subgoal: M(b, U(Tx,y)) in its two occurrences can be matched against the original expression M(b, U(x,y)). This allows definition of a function F(b,x,y) to stand for M(b, U(x,y))

```
F(b,x,y) <= if Null(x) then M(b,y)
            else if M(Hx,y) then F(b,Tx,y)
            else if b=Hx then true
            else F(b,Tx,y)
```

Analysis shows this has a best-case time of $|x|$ and a worst-case time of $|x| \cdot |y|$ but now uses no Cons-cells. Our first goal has been attained.

This can be carried one important step further. The worst-case factor of $|y|$ is due to the expression M(Hx,y): This becomes the next target for simplification. We have the local simplification rule

if p **then** e **else** e $\rightarrow$ e

This could be employed to eliminate M(Hx,y) if we could exchange the order of the second and third clauses of the conditional. Such an exchange is legal so long as the value of the program is not affected, i.e.,

$$\textbf{if } p_1 \textbf{ then } e_1 \textbf{ else if } p_2 \textbf{ then } e_2 \textbf{ else } e_3$$
$$\approx$$
$$\textbf{if } p_2 \textbf{ then } e_2 \textbf{ else if } p_1 \textbf{ then } e_1 \textbf{ else } e_3$$

provided that if p_1 is true then p_2 terminates and

$p_1 \wedge p_2 \supset e_1{=}e_2$

That is, when both predicates apply the values produced are identical. To eliminate M(Hx,y), we must prove

M(Hx,y) $\wedge$ b=Hx $\supset$ F(b,Tx,y)=**true**

This is a simple theorem and, in fact, has been proved using the program verifier described in [6]. Thus, we have

```
F(b,x,y) <=
if Null(x) then M(b,y)
else if b=Hx then true
else F(b,Tx,y)
```

which has constant time in the best case and time |x|+|y| in the worst case, thus attaining the second goal. Note that the final program is equivalent to "**if** b∈x **then true else** b∈y", as expected.

4.3. GENERALIZATION, DECOMPOSITION OF COMPLEX EXPRESSIONS

This example has its origin in the processing of APL. We use several APL operators which may be unfamiliar to the reader. For ease of exposition, we restrict usage to scalar and vector arguments and assume that all vectors are conformable as required; thus, their definitions are simplified:

⍳n	the vector 1,2,...,n
x,y	the concatenation of x with y
+\y	the partial sums of y, i.e., the vector (y_1, y_1+y_2,...,y_1+y_2+...+y_n).
x/y	the compression of y by x, i.e., selects those elements y_i such that x_i=1 and forms a new vector of the selected elements.
~y	the negation of y; the i-th element of the result is 1 if y_i is 0 and 0 if y_i is not 0.
x|y	the mod operation. If x is a scalar and y a vector, the result is a vector of elements (y_i mod x).
	The other binary operations on scalars are extended in the same way.
∧/y	the and-reduction of y, i.e., the logical-and of all elements of y.

To simplify the discussion and carry it out in the same framework as the other examples, we treat APL vectors as if they were lists. Thus ⍳n is treated as I(n,1) where I is the function defined:

```
I(n,k) <= if k>n then nil
          else Cons(k, I(n,k+1))
```

The storage expenditure for ⍳*n* is therefore *n* Cons-cells, which is isomorphic--under our representation--to a vector of *n* elements as expended in an actual APL implementation.

Suppose y is an array of 1's and 0's. The following expression, suggested by Alan Perlis, tests whether all sequences of 1's are of even length: ∧/~2|(~y,0)/+\y,0. This may be read, from right to left as: consider the vector y concatenated with a 0; form a vector of partial sums; select from that vector all elements whose corresponding element in the vector (y,0) is 0; take the remainders of that vector when divided by 2; construct a vector whose i-th element is 1 or 0 as the i-th remainder is 0 or 1; form the logical-and of all elements. That logical-and will be true if and only if all sequences of 1's in y are of even length. This constructs seven temporary arrays and makes eight passes over y and the temporaries.

In our Lisp notation, this is written as

```
(4.4)  R(N(E(C(N(A(y,Cons(0,nil))),
                  S(0, A(y,Cons(0,nil)))))))
```

where

```
A(x,y) <= if Null(x) then y
          else Cons(Hx, A(Tx,y))

S(k,y) <= if Null(y) then nil
          else Cons(k+Hy, S(k+Hy,Ty))

N(y) <= if Null(y) then nil
        else Cons(Not(Hy), N(Ty))

C(x,y) <= if Null(x) then nil
          else if Hx=0 then C(Tx,Ty)
          else Cons(Hy, C(Tx,Ty))

E(x) <= if Null(x) then nil
        else Cons(Hx mod 2, E(Tx))

R(x) <= if Null(x) then true
        else if Hx=0 then false
        else R(Tx)
```

Optimization proceeds from the inside out: transforming argument expressions, substituting the transformed arguments in place of the original ones, and using these in transforming enclosing operations. We begin with S(0, A(y,Cons(0,nil))), which corresponds to +\y,0. Analysis shows that this computation takes 2(|y|+1) Cons-cells, that the length of the output is |y|+1, and that the cells constituting the output come from S. The executions of Cons in A are being wasted. These become the target for simplification.

Taking S(0, A(y, Cons(0,nil))), expanding A and partially evaluating S, results in an expression containing S(Hy, A(Ty, Cons(0,nil))). The first occurrence of 0 in the goal must correspond to Hy in the subgoal, so the match fails. *Generalization* of the first argument position to a new individual variable k is required. Observe that the second occurrence of 0 in the goal matches a 0 in the subgoal, so this is unaffected by generalization. After generalization, the new expression under consideration is S(k, A(y, Cons(0,nil))).

Expanding A, partially evaluating S, and abstracting on a subgoal, results in S(k, A(y, Cons(0,nil))) = F(k,y) where F is defined:

```
F(k,y) <= if Null(y) then Cons(k,nil)
          else Cons(k+Hy, F(k+Hy,Ty))
```

This requires |y|+1 calls of Cons to yield a result of |y|+1 new cells so the goal has been attained.

Next, F(0,y) is substituted for S(0, A(y, Cons(0,nil))) in (4.4) and the decomposition process is repeated. The second expression for optimization is N(A(y, Cons(0,nil))). Again, analysis shows that the executions of Cons in A are being lost. Transformation yields G(y), where

```
G(y) <= if Null(y) then Cons(1,nil)
        else Cons(Not(Hy), G(Ty))
```

Next, the expression C(G(y), F(0,y)) is considered. Analysis shows that the executions of Cons in both G and F are being lost. Transformation yields H(0,y), where

```
H(k,y) <= if Null(y) then Cons(k,nil)
          else if Hy≠0 then H(k+Hy Ty)
          else Cons(k+Hy, H(k+Hy, Ty))
```

Successive steps consider E(H(0,y)), then N of that result, and then R of that result. In all, the optimization process is carried out six times. The final program is J(0,y), where

```
J(k,y) <=
if Null(y) ∧ k mod 2≠0 then false
else if Null(y) then true
else if Hy≠0 then J(k+Hy, Ty)
else if k mod 2≠0 then false
else J(k,Ty)
```

which constructs no temporary arrays and makes, at most, a single pass over y. This may be directly transformed to the iterative program:

```
k←0;
while ~Null(y) do
      {if Hy≠0 then (k←k+Hy; y←Ty)
      else if k mod 2≠0 then return false
      else y←Ty};
if k mod 2≠0 then return false
              else return true
```

5. CONCLUSION

5.1. RELATION TO OTHER WORK

Mechanical program analysis is discussed in [23]. An interactive system which provides assistance to the analyst-user in estimating program efficiency is discussed in [10]. A number of partial evaluators have been implemented for various purposes [8,17,20]. A good survey of partial evaluators and their applications may be found in [5]. Program transformations which preserve correctness with respect to given assertions are discussed in [13]. The notion of loop expansion followed by abstraction to obtain a computational advantage is discussed in [15], in the context of generating efficient code for machines with parallel operation capabilities. More recently, [7] and [18] have employed the idea that a recursive function call can be formed when, in the course of working on a problem, a subgoal is generated that is identical in form to the top-level goal. The use of transformations--"beating" and "drag-along"--to optimize the execution of APL programs is discussed in [1]. Further studies in the optimized interpretation of APL expressions are presented in [4].

The major contribution of this work is in the use of program analysis to direct the transformation process. Using analysis and performance goals to select a target for simplification, and then using this to direct the program expansion steps during partial evaluation seems to be a natural and useful technique. Another contribution is the use of context conditions in partial evaluation to establish enabling conditions for local simplifications and, associated with this, the checking of context conditions in subgoal abstraction. A third contribution is the treatment of generalization in subgoal abstraction. Delaying generalization until required and then generalizing as dictated by the match appears to be a promising approach.

5.2. PROSPECTS

While the techniques we have presented can yield some interesting results, it would be a mistake to overestimate their capabilities. They are limited in effect to the *transformation* of one program to a better one. Cases in which the input/output mapping can be better realized by a radically different algorithm are beyond the scope of this method. For example, we can see no way to transform a definition of bubble-sort to a version of quick-sort. Where change of algorithm is required, program synthesis [18] from input/output specifications appears to be a more natural way to proceed--particularly, if such synthesis were guided by considerations derived from mechanical program analysis [23].

Even within the province of these techniques, there are notable lacuna which invite further investigation. As an example, consider a variation on Example 2.3.2; an APL expression which counts the number of primes less than n is: +/2=+/[1]0=(ιn)∘.|ιn. (Read this from right to left as: Consider the n by n matrix obtained by considering the remainders of all pairs of elements from the arrays (1,2,...,n) and (1,2,...,n); test for equality of the remainders with 0 and form a new matrix of the test results; sum the columns of the resulting matrix; test for equality of the sums with 2; count the number of columns whose sum is exactly 2.) This constructs two n by n temporary matrices and two vectors of length n. The result is a scalar. Analysis and subsequent transformation yields M(n,1) where

```
M(n,k) <=
if k>n then 0
else if H(1,n,k,2) then 1+M(n,k+1)
else M(n,k+1)

H(j,n,k,s) <=
if j>n then s=0
else if k mod j=0 then H(j+1,n,k,s-1)
else H(j+1,n,k,s)
```

This achieves a considerable storage economy since it constructs no temporary matrices or vectors. However, two defects remain. First, the function H(j,n,k,s) does not terminate until j>n, which requires time *n*. Inspection shows that if s is ever negative then H must be false. Thus, inserting a leading test, "**if** s<0 **then false**", would leave the input/output mapping unchanged but typically lead to a performance improvement. Second, the test j>n can be sharpened to j>k, since k mod j≠0 for j>k. However, we can find no entirely satisfactory set of transformations that would lead to these changes.

1 In the interest of brevity, we explain APL only to the extent required for understanding the examples. The original definition by Iverson is given in [16]; a description of the APL system may be found in [19]. Our notation for APL expressions differs from the APL system's in that we use lower case letters for variable names.

2 There should be no confusion between $|\alpha|$ to denote the cost of computing the expression α and $|x|$ to denote the length of the list x. Context and the argument type will indicate which is intended.

3 The following usages are standard in formal logic. A substitution is a set of the form $\{e_i/\nu_i|i=1,...,n\}$ where the ν_i are non-repeating variables and the e_i are expressions. Let Θ be a substitution and α an expression. Then $\alpha\Theta$ is the expression obtained from α by simultaneously replacing each occurrence of ν_i by e_i.

4 It should be pointed out that there are evaluation techniques [22] which defer evaluation of arguments until they are needed and store the result so an argument is evaluated at most once. However, such techniques require that the argument be used in the body exactly as it appears as an actual parameter. In partial evaluation, we wish to carry out simplifications, e.g., H(Cons(G(x)+1, x))-1 → G(x), so that such techniques are not directly applicable. More recent studies [14] show promise of being extendable to such situations, but additional research seems required to clarify the relation between deferred evaluation, local simplification, and function expansion.

ACKNOWLEDGMENTS

My interest in the technique of partial evaluation coupled with subgoal abstraction was stimulated by enjoyable discussions with Rod Burstall and John Darlington; these discussions raised several of the questions answered in this work. Examples 2.3.1 and 4.1 are due essentially to Edsger Dijkstra who presented [11] the versions before and after transformation as being "hard to compare"; the challenge afforded by these examples was a motivating force in exploring the techniques presented here. Several of the APL examples were provided by Alan Perlis; our ongoing discussions as to the relative merits of transformations vs. his "ladder" evaluation techniques for APL have been provocative and fruitful.

REFERENCES

1. Abrams, P. *An APL Machine*. SLAC-14, Stanford Linear Accelerator Center, Feb. 1970.

2. ACM Sigplan Symp. on Very High Level Languages. *Sigplan Notices*, 9, 4, April 1974.

3. Allen, F.E. and Cocke, J. A catalogue of optimizing transformations. In R. Rustin (Ed.) *Design and Optimization of Compilers*, Prentice-Hall, 1972, 1-30.

4. Battarel, G. et al. Optimized interpretation of APL statements. In P. Gjerlov, H.J. Helms, and J. Nielsen (Eds.) *APL Congress 73*, North-Holland, 1973, 49-57.

5. Beckman, L. et al. A partial evaluator and its use as a programming tool. Dept. of Computer Sciences, Uppsala University, Sweden, Nov. 1974.

6. Boyer, R.S. and Moore, J.S. Proving theorems about Lisp functions. *JACM*, 22, 1 (Jan. 1975), 129-144.

7. Burstall, R.M. and Darlington, J. Some transformations for developing recursive programs. *Int. Conf. on Reliable Software*, IEEE Computer Society, April 1975, 465-472.

8. Chang, C.L. and Lee, R. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

9. Cheatham, T.E. and Wegbreit, B. A laboratory for the study of automating programming. *AFIPS Conf. Proc.*, Vol. 70 (Spring 1972), 11-21.

10. Cohen, J. and Zuckerman, C. Two languages for estimating program efficiency. *CACM*, 17, 6 (June 1974), 301-308.

11. Dijkstra, E.W. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (Eds.) *Structured Programming*, Academic Press, 1972, 1-82.

12. Earley, J. High level operations in automatic programming, in [2].

13. Gerhart, S.L. Correctness preserving program transformations. *Second ACM Symposium on Principles of Programming Languages*, Jan. 1975, 54-66.

14. Henderson, P. and Morris, J.H. A lazy evaluator. CSL, Xerox Palo Alto Research Center, August 1975.

15. Holt, A.W. et al. Final report for the information system theory project. Applied Data Research, Inc., New York, Feb. 1968.

16. Iverson, K.E. *A Programming Language*. John Wiley and Sons, Inc., 1962.

17. Lombardi, L.A. and Raphael, B. Lisp as the language for an incremental computer. In E.C. Berkeley and D.G. Bobrow (Eds.) *The Programming Language LISP: Its Operation and Applications*, MIT Press, Cambridge, 1964, 204-219.

18. Manna, Z. and Waldinger, R. Knowledge and reasoning in program synthesis. Stanford Research Inst., Menlo Park, CA, Nov. 1974.

19. Pakin, S. *APL/360 Reference Manual*. Science Research Associates, Inc. 1968.

20. Sanderwall, E.A. Programming tool for management of predicate-calculus-oriented data bases. *Proc. Second Int. Joint Conf. on Artificial Intelligence*, British Computer Society, 1971.

21. Schwartz, J.T. Automatic and semiautomatic optimization of SETL, in [2].

22. Vuillemin, J. Correct and optimal implementations of recursion in a simple programming language. *JCSS*, 9, 3 (Dec. 1974), 332-353.

23. Wegbreit, B. Mechanical program analysis. *CACM*, 18, 9 (Sept. 1975), 528-539.