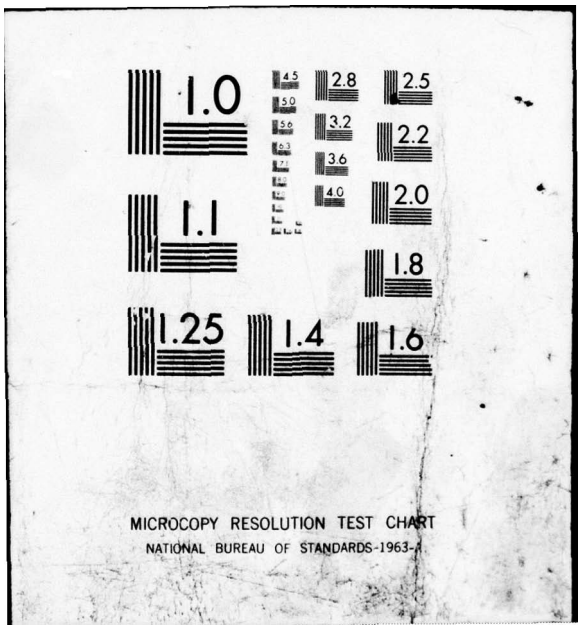
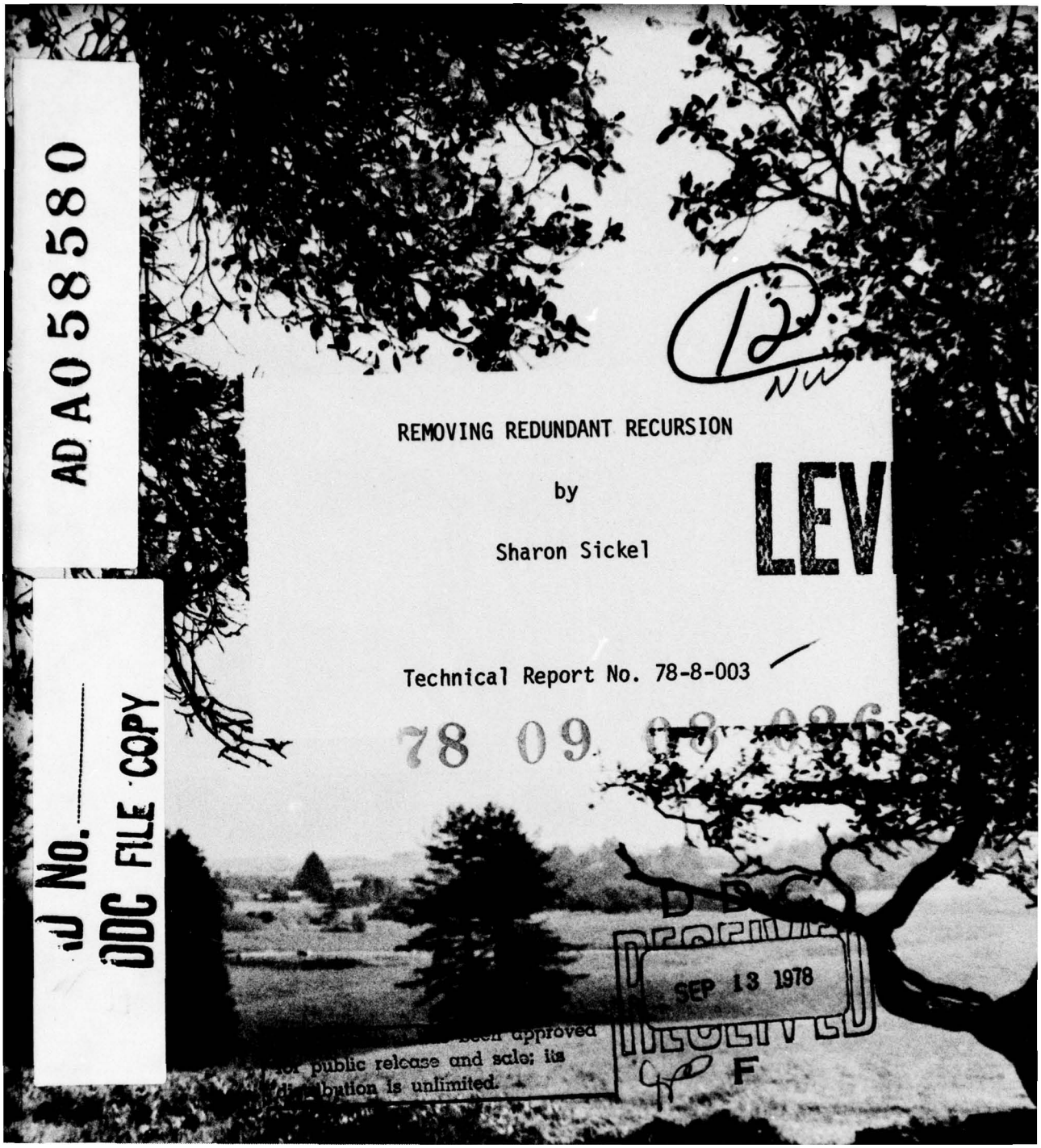
AD-A058 580 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
REMOVING REDUNDANT RECURSION.(U)
AUG 78 S SICKEL N00014-76-C-0682
UNCLASSIFIED TR-78-8-003 NL

1 OF 1
AD A058 580

END
DATE FILMED
11-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A


AD A058580
12
NL
REMOVING REDUNDANT RECURSION
by
Sharon Sickel
LEV
Technical Report No. 78-8-003
78 09 08 086
DDC FILE COPY
SEP 13 1978
public release and sale; its
bution is unlimited.
F

REMOVING REDUNDANT RECURSION

by

Sharon Sickel

Technical Report No. 78-8-003

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

(6) REMOVING REDUNDANT RECURSION

5. TYPE OF REPORT & PERIOD COVERED

(9) Technical rept.

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

(10) Sharon Sickel

8. CONTRACT OR GRANT NUMBER(s)

(15) N00014-76-C-0682

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Information Sciences
University of California
Santa Cruz, California 95064

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

(11) 1 Aug 78

11. CONTROLLING OFFICE NAME AND ADDRESS

Office of Naval Research
Arlington, Virginia 22217

12. REPORT DATE

August 1, 1978

13. NUMBER OF PAGES

27

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Office of Naval Research
University of California
553 Evans Hall
Berkeley, California 94720

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

(12) 33 p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(14) TR-78-8-~~DD~~3

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number) and phrases:

Recursion, non-linear recursion, logic programming, analysis of programs.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents an algorithm that rewrites into efficient form some recursive functions that contain redundant calls (e.g. the Fibonacci function). Burstall and Darlington have optimized such recursions via their unfold, abstract, fold process [3]. This paper generalizes ~~that~~ method and eliminates the need for the user intervention. ~~We give~~ formal definitions for the optimization of functions for both linear and multi-dimensional data types. The algorithm depends upon the arguments of the recursive function being

(continued on next page)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014 6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 350

20. (Continued)

defined by a structural induction, and defines a measure of distance on arguments that determines when and how the rewrite can be carried out. This paper is not concerned with the implementation issues arising from the relative efficiency of recursive and iterative mechanisms in programming languages, but rather with restructuring the algorithms themselves.

REMOVING REDUNDANT RECURSION

by

Sharon Sickel

Information Sciences

and

Crown College

University of California

Santa Cruz

This work was supported by Office of Naval Research Contract # 76-C-0682

ABSTRACT

This paper presents an algorithm that rewrites into efficient form some recursive functions that contain redundant calls (e.g. the Fibonacci function). Burstall and Darlington have optimized such recursions via their unfold, abstract, fold process [3]. This paper generalizes that method and eliminates the need for the user intervention. We give formal definitions for the optimization of functions for both linear and multi-dimensional data types. The algorithm depends upon the arguments of the recursive function being defined by a structural induction, and defines a measure of distance on arguments that determines when and how the rewrite can be carried out. This paper is not concerned with the implementation issues arising from the relative efficiency of recursive and iterative mechanisms in programming languages, but rather with restructuring the algorithms themselves.

1. Background

Intuitively, a function is recursive if it is defined in terms of itself. More precisely, suppose we have known functions g and h, and write

$$f(x) = g(f(h(x)),$$

then f is said to be recursive. The recursion takes the parameter value x, modifies it in some way via h, applies f and then uses the result in some further computation g. In a more familiar special case, we might have

$$f(x) = g(f(x-1))$$

where h merely subtracts one from the argument.

The recursive formula, alone, is not sufficient to define f; we must always have at least one basis case which is a value of f that is directly computable without reference to f.

Programming languages implement recursion of this sort via procedures in which each application of the definition consumes some time (and perhaps space). In this environment, reducing the number of applications leads to increased efficiency. This is the primary objective of this paper. There has been a great deal written about transforming recursion into iteration [e.g. 1,2] and the analysis of recursion [e.g. 3,4,6]. Burstall and Darlington [3] introduced the concept of removing redundant recursive calls from the computation altogether by transforming the program into an equivalent, optimized form. This work is extended and formalized here.

We shall first define recursion formally for constructed data types, and give some examples of recursive functions. We then formally define the rewriting algorithms and the classes of recursions that can be rewritten to reduce the number of function applications and give examples of such rewritings.

The domain D of a constructed data type consists of a set of constants A and a set of constructor functions S together with all values from repeated applications of constructor functions to the constants. Suppose s_i is a constructor function and x is in D. Then we define $s_i(x) > x$. The transitive closure of the relation ">" defines a partial order (sometimes a total order) on D. For example, if S = {successor} and A = {0}, then D = {0,successor(0),successor(successor(0)),...} = Z, the non-negative integers. The intuitive ordering of the non-negative integers is the same as the order defined above.

Assume that a and ai are constants, s and s_i are constructor functions, and x and x_i are variables that can take on any value in the domain.

We define three types of functions:

1) Successor

$s(x_1) = x_2$ i.e. any constructor function.

2) Constant

$c(x_1,\ldots,x_n) = a$

3) Pick-out

$p(x_1,\ldots,x_n) = x_i$ for any particular i, $1 \le i \le n$. For example, the identity function.

Two methods of building functions:

4) Composition

$f(x_1,\ldots,x_n) = g(h_1(x_1,\ldots,x_n),\ldots h_m(x_1,\ldots,x_n))$ where

$g: D^m \rightarrow D$

$h_i: D^n \rightarrow D$ for $1 \le i \le m$

i.e. the function f is defined wholly in terms of the composition of other functions.

5) Primitive Recursion: for $n \ge 1$,

$$f(a,x_2,\ldots,x_n) = g(x_2,\ldots,x_n) \qquad \text{(basis case)}$$

$$f(s(x_1),x_2,\ldots,x_n) = h(x_1,\ldots,x_n, f(s_1(x_1),x_2,\ldots,x_n), f(s_m(x_1),x_2,\ldots,x_n)) \qquad \text{(induction case)}$$

where

$s_i(x_1) < s(x_1)$ for $1 \le i \le m$

While this definition of primitive recursion appears more general than usual

[7], it is merely a convenience and does not increase the power of primitive recursion.

A primitive recursive function is defined for a basis value. One of the arguments of f is used to "control" the computation. If it equals a basis value, then the function is evaluated as a constant or as the value of another function. If the value of the control variable x is not a basis value, then f is expressed in terms of f applied to a smaller value than x.

We define the set of primitive recursive functions:

1) Successor, Constant, and Pick-out functions are primitive recursive.
2) If f is composed of primitive recursive functions then f is primitive recursive.
3) If f is defined by primitive recursion in terms of only itself and other previously defined primitive recursive functions then f is primitive recursive.

Primitive recursive functions always terminate since they are defined over constructed data types and remove at least one level of structuring with each recursive application.

The Fibonacci function defining the sequence 1, 1, 2, 3, 5, ... where each new number is the sum of the preceeding two can be defined as follows:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(x+2) = fib(x+1) + fib(x)$$

using two basis cases and a primitive recursion. Or, suppose we have a domain D where A = {empty_tree} and S = {graft: D x Z x D → D}. The constructed data type is the set of binary trees with nodes having non-negative integer values.

The function defined below sums the values of all the nodes of a tree.

$$sum(empty\_tree) = 0$$

$$sum(graft(t_1,v,t_2)) = sum(t_1) + v + sum(t_2)$$

2. Recursion Elimination

Recursive functions can be classified as linear or non-linear according to $m = 1$ or $m > 1$ in the definition of primitive recursion. The factorial function, fact: $Z \rightarrow Z$, defined by

$$fact(0) = 1$$

$$fact(n+1) = (n+1) * fact(n)$$

is linear. The Fibonacci function is non-linear.

Recursive functions can also be classified as redundant or not redundant according to whether the function is ever applied more than once to the same input in the course of evaluating some particular case. Factorial is not redundant, and Fibonacci is redundant.

If a recursive function is redundant, there is the possibility of collapsing the repeated function evaluations. In its simplest form, a course-of-values induction [5], by saving all previous values, gives the model demonstrating the possibility. Course-of-values induction requires, in general, an unbounded amount of storage. The method presented here, when applicable, requires only a bounded amount of storage.

A linear recursion is never redundant since if a value were to be repeated, the recursion would have been improperly defined, violating the condition that the subcall be on a simpler argument than the original argument. Thus, we are interested here only in non-linear redundant recursions. One way to make them not redundant is to rewrite them as linear recursions.

A function of n arguments can be defined as a predicate of n+1 arguments by letting the predicate stand for the relation between arguments and value. Suppose F is a predicate that is true iff F(x,f(x)) for all x in the domain of function f. Then F is a definition of f. Such predicates can be recursively defined. For example, the predicate FACT(x,y) defined by:

$$FACT(0,1) \leftarrow$$

$$FACT(x+1,(x+1)*y) \leftarrow FACT(x,y)$$

gives the factorial function.

The implication "$\leftarrow$" is used the same as the relation "=" in primitive recursion, thus, we can extend our definition of primitive recursion to predicates.

The function $d(n) = 2^n$ can be defined by

$$d(0) = 1$$

$$d(n+1) = d(n) + d(n)$$

which is non-linear and redundant. In predicate form it becomes

$$D(0,1) \leftarrow$$

$$D(n+1,y+y) \leftarrow D(n,y)$$

which is linear (hence not redundant). This is the simplest case of recursion elimination. Note that it did not depend upon the properties of "+"; $d(n+1) = h(d(n),d(n))$ for any function h becomes $D(n+1,h(y,y)) \leftarrow D(h,y)$.

More interesting is the Fibonacci function for which a linearization was found by Burstall & Darlington [3]. The definition presented previously gives rise to the predicates

$$FIB(0,1) \leftarrow$$

$$FIB(1,1) \leftarrow$$

$$FIB(n+2,x+y) \leftarrow FIB(n+1,x) \wedge FIB(n,y)$$

which are still non-linear and redundant. However, we modify the definition of FIB to be

$$FIB(0,1,\Omega) \leftarrow$$

$$FIB(1,1,1) \leftarrow$$

$$FIB(n+1,x+y,x) \leftarrow FIB(n,x,y)$$

where Ω denotes the value "undefined". The meaning of FIB is:

$$FIB(n,fib(n),fib(n-1)) \text{ for } n \geq 0.$$

The new computation of Fibonacci is not redundant. This paper generalizes upon the above method.

3. Method

We make the following assumptions about the functions we are trying to optimize. Any exceptions to these assumptions will be noted explicitly in the various cases. Descriptions of criteria are in terms of a function f whose recursive formulas are expressions:

$$f(x) = g(f(x_1),\ldots,f(x_n)),n \geq 2,g \text{ not dependent on } f$$

1. Recursion based on decomposition. The recursive calls to the function should have inputs that are substructures of the original input, i.e. the result of peeling away at least one level of constructor applications.§ For example,

$$sum(graft(t_1,r,t_2)) = sum(t_1) + r + sum(t_2)$$

---

§ All that is really essential is that the domain be countable and have a lower bound. If that is the case, a structural construction can be defined.

satisfies recursion based on decomposition for binary trees since t_1 and t_2 are substructures of the original. However, the function

$$f(2*i) = f(i) + f(i+1)$$

on natural numbers does not satisfy the constraint, since the breakdown of input 2*i is not based on the constructor, successor, for natural numbers. This latter example would also fail condition 3 below.

2. Multiple, simpler calls. The recursive definition of the function should make two or more subcalls to the function, each of which has an input simpler than the original, i.e.

$$f(x) = g(f(x_1),f(x_2),\ldots,f(x_n)) \quad \text{where}$$
$$x > x_i, 1 \leq i \leq n.$$

For convenience, it may also be assumed that $x_i \neq x_j$ if $i \neq j$. If redundancy of this sort occurred, one could remove it by replacing function g by function g' that computes the same mapping, but accepts fewer arguments.

E.g. if $f(s(n)) = g(f(n),f(n))$ where $g(x,y) = x^2+y$, then we could rewrite f: $f(s(n)) = g'(f(n))$ where $g'(x) = x^2+x$.

3. Bounded span. Intuitively, span is the structural distance between two data objects, i.e. the amount of construction required on the second argument to obtain the first. E.g. where g is a constructor span(g(g(x,y),z), y) = 2 since y is a subterm of g(g(x,y), z), inside 2 levels of construction.

Given constant b and constructor c, span is formally defined:

$$\text{span}(b,y) = \infty \text{ for } b \neq y$$
$$\text{span}(x,x) = 0$$
$$\text{span}(c(x_1,\ldots,x_k),y) = 1+\min_{i=1}^{k}(\text{span}(x_i,y))$$

For example, span(g(x,g(x,y)),x) = 1, demonstrating the disambiguation if the subterm in question appears twice.

The requirement for bounded span, means that there is a bounded distance from the original call of the recursive function to each of the inputs of its subcalls, i.e. for function f to satisfy this criterion, there exists an integer, I, that is constant for f such that for

$$f(x) = g(f(x_1),\ldots,f(x_n))$$

$$\max_{i=1}^{n}(\text{span}(x,x_i)) \leq I.$$

The importance of bounded span is that it implies that there is a finite limit to the amount of previous knowledge required at any point to compute f.

If a subcall is not a bounded distance from a call, but is a bounded distance from the basis, then we may want to revise the function definition to include these bottom cases, by replacing the recursive calls to those cases by their computed values.

4. Single constructor. Our definition of primitive recursion made no requirement that there be a single constructor function. All of the data types we have seen have a single constructor, e.g. successor for natural numbers and graft for trees. This criterion will be varied later.

5. Linearly constructed objects. A linear constructor takes a single argument. All objects built from a linear constructor, c, have the form $c^k(b)$ where b is a basis constant and $k \geq 0$. Binary trees are not linearly constructed. This criterion will also be varied later.

6. Single parameter functions. At the beginning of this section, f is described as a function taking only one argument. That argument is the one that drives the recursion. Variations on this criterion will be discussed later.

7. Single recursive formula. The definition for f may contain only one recursive formula. The following function, g, would be excluded:

$$g(0) = 1$$
$$g(s(x)) = g(x) \quad \text{if } x \text{ odd}$$
$$g(s(s(x))) = g(x) + g(s(x)) \quad \text{if } x \text{ even}$$

This criterion will be varied later.

Case 1. Simplest case: assumes all of the above. Let the single constructor be c. As a shorthand notation, let $\underbrace{c(c\ldots c}_{m}(x)\ldots)$ be denoted x+m. Then

$f(x) = g(f(x_1),\ldots,f(x_n))$ can be rewritten as

$f(x+k_n) = g'(f(x+k_{n-1}),\ldots,f(x+k_2),f(x+k_1),f(x))$

where $k_i > k_{i'}$ if $i > i'$ and $j = k_n$. I.e. the arguments to the sub-calls of f are in decreasing order, $j = \max_{i=1}^{n}(\text{span}(x,x_i))$, and g' similar to g but with reordered arguments.

Knowing the values of f applied to the j previous values less than x may not be a necessary condition for computing f(x), but it is a sufficient one. The following predicate form of the function f carries along the last j values computed. Then to compute f(x+1), all required information is contained in f(x), so a single recursive call is required. All except the oldest of the j values is passed along to be stored with x+1 for use in the next computation.

Now to formalize the construction of the optimized predicate:

if for basis constant a

$$f(a) = a_0$$

$$f(a+1) = a_1$$

$$\vdots$$

$$f(a+j-1) = a_{j-1}$$

$$f(x+j) = g'(f(x+k_{n-1}),\ldots,f(x+k_2),f(x+k_1),f(x))$$

then define predicate F taking j+1 arguments:

$$F(a,a_0,\Omega,\ldots,\Omega) \leftarrow$$

$$F(a+1,a_1,a_0,\Omega,\ldots,\Omega) \leftarrow$$

$$F(a+2,a_2,a_1,a_0,\Omega,\ldots,\Omega) \leftarrow$$

$$\vdots$$

$$F(a+j-1,a_{j-1},\ldots,a_1,a_0) \leftarrow$$

$$F(x+j,g''(z_{j-1},\ldots,z_1,z_0),z_{j-1},\ldots,z_1) \leftarrow$$
$$F(x+j-1,z_{j-1},\ldots,z_1,z_0)$$

g" is a function similar to g but it takes exactly j arguments which is possibly more than the number required by g. The added values are simply ignored.

For example, consider the original example that motivated this kind of optimization, Fibonacci.

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(x+2) = fib(x+1) + fib(x)$$

The new predicate is:

$$FIB(0,1,\Omega) \leftarrow$$

$$FIB(1,1,1) \leftarrow$$

$$FIB(x+2,z_1+z_2,z_1) \leftarrow$$
$$FIB(x+1,z_1,z_2)$$

Modulo Optimization of Case 1. In the previous case, we have kept the results of f applied to all values over the full span, even though some may never be needed. The reason to keep all values is to guarantee that values needed further up the chain will exist. For example,

$$f(x+3) = f(x+1) \cdot f(x)$$

does not need f(x+2), but that value needs to be kept so it can be passed on, because the computations of f(x+4) and f(x+5) require it.

If we had some way to guarantee that certain values would never be needed, we could optimize the predicate form. For example,

$$f(0) = a_0$$
$$f(1) = a_1$$
$$f(2) = a_2$$
$$f(3) = a_3$$
$$f(x+4) = f(x+2) + f(x)$$

Then values of f on even input will be used only for larger even values and similarly for odd values. So long as the shift distance between any two successive increments, i.e. the k_i's, and between k_{n-1} and j, is a non-zero multiple of a given constant, then we can reduce the chain of temporaries.

In general, given definition of f as follows:

$$f(0) = a_0$$
$$f(1) = a_1$$
$$\vdots$$
$$f(j-1) = a_{j-1}$$
$$f(x+k_n) = g(f(x+k_{n-1}),\ldots,f(x+k_1),f(x))$$

where $k_i > k_{i'}$ if $i > i'$ and $k_i > 0$ for all i, and $j = k_n$,

we define

$$modulo(f) = gcd(k_n, k_{n-1}, \ldots, k_1)$$

where "gcd" denotes greatest common divisor. Letting m denote modulo(f), then we can represent f as predicate F, described semantically:

$$F(x+k_n, f(x+k_n), f(x+(k_n-m)), \ldots, f(x+2m), f(x+m))$$

F takes p+1 arguments where p = j/m.

The full definition for F is:

Termination cases, for $0 \leq t \leq j-1$:

$$F(t, a_k, a_{k-m}, \ldots, a_k \bmod m, \Omega, \ldots, \Omega) \leftarrow$$

Recursive formula:

$$F(x+j, g'(z_p, \ldots, z_1), z_p, \ldots, z_2) \leftarrow$$
$$F(x+j-m, z_p, z_{p-1}, \ldots, z_1)$$

where g' is a function similar to g, but it takes, perhaps, more arguments, since every m_th value between x and x+j is included.

For example, given f:

$$f(0) = 0$$
$$f(1) = 10$$
$$f(2) = 20$$
$$f(3) = 30$$
$$f(4) = 40$$
$$f(5) = 50$$
$$f(x+6) = f(x+2) + f(x)$$

span(x+6,x) = 6

modulo(f) = gcd(6,2) = 2

arguments = (6/2)+1 = 4

and F is:

$$F(0,0,\Omega,\Omega) \leftarrow$$

$$F(1,10,\Omega,\Omega) \leftarrow$$

$$F(2,20,0,\Omega) \leftarrow$$

$$F(3,30,10,\Omega) \leftarrow$$

$$F(4,40,20,0) \leftarrow$$

$$F(5,50,30,10) \leftarrow$$

$$F(x+6,(z_2+z_1),z_3,z_2) \leftarrow$$

$$F(x+4,z_3,z_2,z_1)$$

If we had not done this optimization, we would have needed six values of f in the predicate rather than three.

Case 2. Use above assumptions except that more than one recursive formula is allowed in the definition of f. Define:

span(f) = the maximum of the spans of the individula formulas

modulo(f) = the greatest common divisor of the modulos of the individual formulas

Then construct an F predicate for each recursive formula (plus basis cases, of course) but use these general definitions of span and modulo for each individual formula. Then add necessary basis cases that will fill in for formulas with smaller spans.

The following example shows two recursive formulas and their corresponding predicate. Since the basis cases are handled the same as before we have not bothered to include them here. Function f has two recursive formulas d_1 and d_2. (The two formulas may provide alternate methods of computation, perhaps based on some case breakdown, e.g. a criterion on the input. Presumably, they do not allow different answers for a given input.)

d_1: $f(x+6) = g(f(x))$

d_2: $f(x+9) = g'(f(x+6),f(x))$

$span(f) = max(span(d_1),span(d_2))$

$= max(6,9) = 9$

$modulo(f) = gcd(modulo(d_1),modulo(d_2))$

$= gcd(6,3) = 3$

Then the two recursive predicates are:

$$F(x+9,g(z_2),z_3,z_2) \leftarrow$$
$$F(x+6,z_3,z_2,z_1)$$
$$F(x+9,g'(z_3,z_1),z_3,z_2) \leftarrow$$
$$F(x+6,z_3,z_2,z_1)$$

Now since we have increased the lowest value which the recursive case can take as input from 6 to 9, we need to add basis cases:

$$F(6,g(a_0),a_3,a_0) \leftarrow$$
$$F(7,g(a_1),a_4,a_1) \leftarrow$$
$$F(8,g(a_2),a_5,a_2) \leftarrow$$

where $a_i = f(i)$ *for basis cases at least as high as 5.*

Case 3. Multiple parameter functions. In our definition of primitive recursion we allowed multiple arguments to the functions, but only one argument drove the recursion. For our purposes here, additional arguments are simply carried along, i.e. if f(y) maps to $F(y_1,\ldots,y_k)$, then $f'(x_1,\ldots,x_n,y)$ maps to $F'(x_1,\ldots,x_n,y_1,\ldots,y_k)$ where the arguments $x_1,\ldots,x_n$ are passed from one call of F' to the next.

However, consider the possibility that more than one argument drives the

recursion, e.g.

$$f(x+1,y+1) = h(f(x+1,y),f(x,y+1),f(x,y))$$

There clearly is redundancy here, since f(x,y) will be called as a subgoal by each of f(x+1,y) and f(x,y+1). However, we do not optimize here because there is no way to define a total ordering on pairs of natural numbers such that for all x and y there exists some constant I such that:

$$(x+1,y+1) > (x+1,y) > (x,y)$$

and

$$(x+1,y+1) > (x,y+1) > (x,y)$$

and

$$span((x+1,y+1),(x,y)) \leq I.$$

Therefore, the example fails the hypotheses.

There are some functions that appear to be driven by multiple arguments but could easily be mapped onto functions that are driven on a single argument. For example,

$$f(x+1,y+1) = h(f(x,y))$$

could be described

$$f'((x,y)+1) = h'(f'(x,y))$$

where the ordered pair is incremented as a unit. And in fact, this is key to the decisions 1) of whether a function driven by multiple arguments is primitive recursive or 2) of the applicability of our reduction procedures.

In general, a functional description whose recursion is driven by n arguments is primitive recursive if there is a total ordering on n-tuples of the data-type such that the input of the goal is strictly greater than each of its subgoals. Then replace each reference to an n-tuple by a reference to

its index in the ordering. That makes the description follow the syntax of primitive recursion.

If one wants to remove redundancy in such a function, one maps it onto a function on the natural numbers through the enumeration of its n-tuples, then proceeds as for single parameter functions.

Case 4. Multiple constructors. When there is more than one constructor function, the definitions get complicated to the extreme. It also may be too far-fetched to consider except in some particular uses of combinations.

If constructors are used only in certain combinations, it might be possible to rephrase in terms of a single constructor.

If multiple constructors are used but never mixed, it may be possible to use the simplification techniques on each constructor separately.

Otherwise, there is no method given for removing the redundancies with which we are concerned here.

Case 5. Use original assumptions except that the constructed objects may be multi-dimensional, i.e. non-linear. For example, consider the data-type Family Tree. The simplest family tree is a person; so the set of constants is the set of people. We use constructor function gr, which is similar to graft used previously, with added semantics that the left sub-tree is the mother's sub-tree and the right sub-tree is the father's sub-tree.

The expression gr(gr(Amy,Ann,Ted),Sue,Jack) denotes the family tree shown in Figure 1.

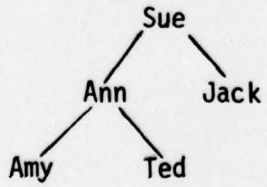


Figure 1: Sue's family tree.

Now we define a Fibonnaci-like function on family trees.

$$f(gr(gr(x,y,z),u,v)) = h(f(gr(x,y,z)),f(x))$$

That is, f is a function that is called recursively on the mother's sub-tree and the maternal grandmother's sub-tree.

We remove the redundancy here in predicate F having semantics

$$F(gr(gr(x,y,z),u,v),f(gr(gr(x,y,z),u,v)),f(gr(x,y,z))),$$

i.e. $F(x,f(x),f(x\text{'s mother}))$

and formally define the recursive component of F as:

$$F(gr(x,y,z),h(w_1,w_2),w_2) \leftarrow$$
$$F(x,w_2,w_1)$$

Example 2 is only slightly different from the first.

$$f(gr(gr(t_1,r_1,t_2),r_2,gr(t_3,r_3,t_4)))$$
$$= h(f(gr(t_1,r_1,t_2)),\ f(gr(t_3,r_3,t_4)),f(t_1),f(t_3))$$

The input is shown in tree-form in Figure 2, and arrows point from the goal to its sub-goals.

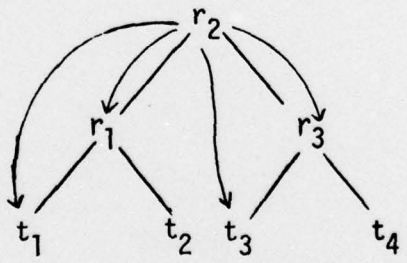


Figure 2: Dependency relationships in Example 2.

In this example, the sub-goals lower than level one that need to be remembered are all first arguments, i.e. mothers. So, we only need to change the newly

computed value in the last version of F and add a recursive call for the second argument to get the F for this new function, i.e.

$$F(gr(x,y,z),h(w_2,w_4,w_1,w_3),w_2) \leftarrow$$

$$F(x,w_2,w_1)$$

$$\wedge\ F(z,w_4,w_3)$$

The resulting recursion is still non-linear, but the redundancy is gone. The multiple calls at the sub-goal level are on disjoint objects, i.e. the original input gr(x,y,z) is partitioned into three objects, x, y, and z, and recursive calls are made on x and z.

In general, if a function on a constructed object $c(x_1,\ldots,x_n)$ is defined in terms of recursive calls to $x_1,\ldots,x_n$ *and no other objects*, then the structure has been cleanly partitioned and there is no redundancy of the type we are concerned with here. It is when the sub-goal inputs overlap that the redundancy occurs.

Considering another example, Example 3:

$$f(gr(gr(t_1,r_1,t_2),r_2,gr(t_3,r_3,t_4)))$$

$$= h(f(gr(t_1,r,t_2),f(gr(t_3,r_3,t_4)),f(t_1),f(t_4)),$$

illustrated in Figure 3.

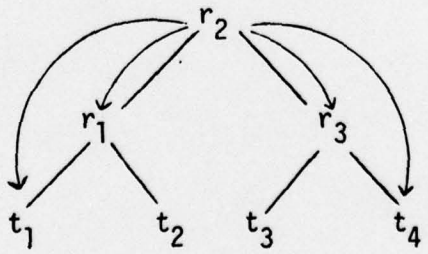


Figure 3: Dependency relationships in Example 3.

In this case, the values below level one that must be remembered are in both argument positions, i.e. a first argument (t_1) and a second argument (t_4). Since when we compute f on any given input we don't know how that input may be called from other places, we need to remember all potentially required values. So, F in this case semantically is:

$$F(x,f(x),f(x\text{'s mother}),f(x\text{'s father}))$$

Formally:

$$F(gr(x,y,z),h(w_3,w_6,w_2,w_4),w_3,w_6) \leftarrow$$

$$F(x,w_3,w_2,w_1)$$

$$\wedge\ F(z,w_6,w_5,w_4)$$

Example 4:

$$f(gr(t_1,r_1,gr(gr(t_2,r_2,t_3),r_3,t_4)))$$

$$= h(f(t_1),f(gr(gr(t_2,r_2,t_3),r_3,t_4)),f(t_2)),$$

as shown in Figure 4.

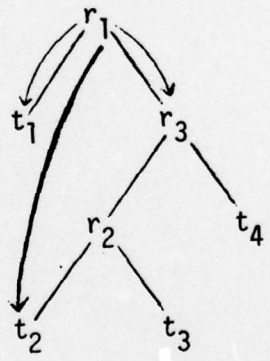


Figure 4: Dependency relationships in Example 4.

In this case, we need only remember a left descendant, but we have to remember

it two levels away. Semantically F is:

F(x,f(x),f(x's mother),f(x's maternal grandmother))

Formally:

$$F(gr(x,y,z),h(w_3,w_6,w_4),w_3,w_2) \leftarrow$$

$$F(x,w_3,w_2,w_1)$$

$$\wedge\ F(z,w_6,w_5,w_4)$$

Notice that in each of the four examples, above, the optimized form is not only more computationally efficient, but also has added clarity, since the input argument is, in each case, either a basis object (not shown) or the simplest form of constructed object, i.e. only the outer structure must be shown in the function definition.

The previous examples provide examples of 1) span and 2) breadth. Span is the same as before: the distance from the input to the smallest sub-goal. The formal definition given in section 3 still holds. Breadth is defined informally as the collection of parameter positions of the sub-goals that are deeper than level one.

Given $f(c(x_1,\ldots,x_n)) = h(f(y_1),\ldots,f(y_k))$

Breadth(f) = $\{i \mid B(c(x_1,\ldots,x_n),y_j,i)$ for some j, $1 \leq j \leq k\}$

where

$$B(c(x_1,\ldots,x_n),x_i,i) \leftarrow 1 \leq i \leq n$$

$$B(c(x_1,\ldots,x_n),y_j,i) \leftarrow y_j \neq x_m \text{ for any } m,\ 1 \leq m \leq n$$

$$\wedge\ B(x_1,y_j,i)$$

$$\vdots$$

$$B(c(x_i,\ldots,x_n),y_j,i) \leftarrow y_j \neq x_m \text{ for any } m,\ 1 \leq m \leq n$$

$$\wedge\ B(x_n,y_j,i)$$

Then the set of remembered values required is a tree of depth = (span-1) and branching determined by breadth. For example, if Breadth = {1,2,4}, and span = 3 then the tree of temporaries is as shown in Figure 5.

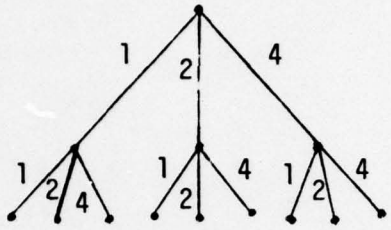


Figure 5.

For Example 4, Breadth = {1}, Span = 3. The tree of temporaries is of depth 2 and does not branch, i.e. as in Figure 6.

Figure 6.

The semantic description of F in general is:

F(x,f(x), tree of history including all descendants indexed by Breadth, (Span-1)deep).

Formally, the recursive case of F is:

$$F(c(x_1,\ldots,x_n),h'(f(x_1),T_1 \ldots f(x_n),T_n),T) \leftarrow$$
$$F(x_1,f(x_1),T_1) \wedge \text{REMOVE\_LEAVES}(T_1,T_1')$$
$$\vdots$$
$$\wedge F(x_n,f(x_n),T_n) \wedge \text{REMOVE\_LEAVES}(T_n,T_n')$$
$$\wedge \text{Construct\_tree}(f(x_1),T_1',f(x_2),T_2',\ldots f(x_n),T_n',T)$$

and the basis cases are defined appropriately as far as there is history, and Ω, the value "undefined", is used to fill out the tree to the leaves.

Case 6. Up to this point we have restricted discussion to definition of a single function. We now extend it to the case of multiple functions being computed simultaneously on a single input. There are two cases; one is substantially easier than the other.

In the first case, the functions are independent, i.e. they are defined recursively but not in terms of each other, and the parameters for the subgoals are identical. For example, assume the data type of binary trees of integers, and define two functions, one of which finds the sum of the nodes of the tree and the other computes the product.

$$\text{sum}(\text{empty\_tree}) = 0$$

$$\text{sum}(\text{graft}(t_1,v,t_2)) = \text{sum}(t_1) + v + \text{sum}(t_2)$$

and

$$\text{prod}(\text{empty\_tree}) = 1$$

$$\text{prod}(\text{graft}(t_1,v,t_2)) = \text{prod}(t_1) * v * \text{prod}(t_2)$$

We can compute both functions with one pass over the tree using predicate

$$\text{SUMPROD}(\text{empty\_tree},0,1) \leftarrow$$

$$\text{SUMPROD}(\text{graft}(t_1,v,t_2),z_1 + z_3,z_2 * z_4) \leftarrow$$

$$\text{SUMPROD}(t_1,z_1,z_2)$$

$$\wedge\ \text{SUMPROD}(t_2,z_3,z_4)$$

Formally, for functions $f_1,\ldots,f_n$ such that for the recursive formulas of the f_i's:

$$f_i(x) = h_i(f_i(y_1),\ldots,f_i(y_m))$$

$$\leftrightarrow$$

$$f_j(x) = h_j(f_j(y_1),\ldots,f_j(y_m))$$

for all i, j, $1 \le i, j \le n$, and for all x, y_k, $1 \le k \le m$.

and the domain of the f_i's contains only constant a, then construct predicate F:

$$F(a,f_1(a),\ldots,f_n(a)) \leftarrow$$

$$F(x,h_1(z_{11},\ldots,z_{m1}),\ldots,h_n(z_{1n},\ldots,z_{mn})) \leftarrow$$

$$\wedge\ F(y_1,z_{11},\ldots,z_{1n})$$
$$\vdots\quad\vdots$$
$$\wedge\ F(y_m,z_{m1},\ldots,z_{mn})$$

Now consider the case where the functions are either not independent or they don't generate exactly the same parameters to the subgoals, e.g. consider functions fact and factlist:

$$fact(0) = 1$$

$$fact(n+1) = (n+1) * fact(n)$$

$$factlist(0) = [fact(0)]$$

$$factlist(n+1) = fact(n+1) \otimes factlist(n)^{\S}$$

We can compute the functions together in predicate F, having semantics F(x,fact(x),factlist(x)), as follows:

$$F(0,1,[1]) \leftarrow$$

$$F(n+1,z,z \otimes z_2) \leftarrow$$

$$F(n,z_1,z_2)$$

$$\wedge\ z = (n+1) * z_1$$

---

§ $\otimes$ denotes an infix "cons" of an object to a list.

Formalizing this mapping, assume a set of functions $S = \{f_1,\ldots,f_n\}$, and n recursive definitions:

$$f_i(x) = h_i(g_{i1}(y_{i1}),\ldots,g_{im}(y_{im}))$$

where $g_{ij} \in S$, and $y_{ij} \leq x$ and if $y_{ij} = x$, then $g_{ij} = f_m$ where $m < i$.

I.e. the functions are mutually defined and if any function has a subgoal whose input is not decreased, then the function of that subgoal must be computed first. E.g. in the above example, fact(n) must be computed before factlist(n).

Given the above hypotheses, we construct predicate F having semantics $F(x,f_1(x),\ldots,f_n(x))$ as follows:

$$F(a,f_1(a),\ldots,f_n(a)) \leftarrow$$

$$F(x,w_1,\ldots,w_n) \leftarrow$$

$$\begin{aligned}
&\wedge\ F(y_{11},z_{111},\ldots,z_{11n})\\
&\quad\vdots\\
&\wedge\ F(y_{nm},z_{nm1},\ldots,z_{nmn})\\
&\\
&\wedge\ w_1 = h_1'(z_{111},\ldots,z_{nmn})\\
&\wedge\ w_2 = h_2'(z_{111},\ldots,z_{nmn},w_1)\\
&\quad\vdots\\
&\wedge\ w_n = h_n'(z_{111},\ldots,z_{nmn},w_1,\ldots,w_{n-1})
\end{aligned}$$

where h_i' is similar to h_i but may accept, and ignore, extra arguments. What, in reality, should happen is that the original h_i's are used and applied to the relevant subset of values.

The previous two constructions allow simultaneous computation of multiple

functions. Many redundancies are automatically omitted, but we may be able to further optimize by treating the resulting predicates as functions $D \rightarrow D^n$ and applying improvements described in cases 1-5.

Conclusions

We have defined some concepts of optimizing redundant recursion and given algorithms to perform the optimization. One result is that the computation of the function runs faster, (perhaps exponentially), and the recursive structure is simplified.

REFERENCES

1. Auslander, M.A. and Strong, H.R. Systematic Recursion Removal. Comm. ACM 21,2 (February 1978), 127-134.

2. Bird, R.S. Notes on Recursion Elimination. Comm. ACM 20,6 (June 1977), 434-439.

3. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs. D.A.I. Research Report No. 19, Dept. of Artificial Intelligence, University of Edinburgh (1976).

4. Manna, Z. and Waldinger, R. Structured Programming with Recursion. Rpt. STAN-CS-77-640, Comp. Sci. Dept., Stanford University (1978).

5. Mendelson, E. Introduction to Mathematical Logic. D. van Nostrand Co., Princeton, N.J. (1964).

6. Morris, James H., Jr. Another Recursion Induction Principle. Comm. ACM 14,5 (May 1971), 351-354.

7. Yasuhara, Ann. Recursion Function Theory & Logic. Academic Press, New York (1971).

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 Copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 Copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 Copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 Copy

Office of Naval Research
Branch Office, Boston
Bldg. 114, Section D
666 Summer Street
Boston, MA 02210
1 Copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, ILL 60605
1 Copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 Copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 Copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380
1 Copy

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 Copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 Copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
1 Copy