Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189

http://amresearchreview.com/index.php/Journal/about

REMOVING RECURSION FROM PROGRAMS WITH ARRAYS

Tanvir Ahmed
Department of Computer Science & Software Engineering, Grand Asian University Sialkot, 7 KM Pasrur Road, Sialkot Punjab, Pakistan
Email: drtawaraich@gmail.com

Ishfaq Ali
Department Of Computer Science, Faculty Of Sciences, National Textile University Main Campus Faisalabad Pakistan Email: ishfaqalimalik@gmail.com

Rai Samee Ullah*
Department of Computer Science & Software Engineering, Grand Asian University Sialkot, 7 KM Pasrur Road, Sialkot Punjab, Pakistan Email: sameetatla@gmail.com

ABSTRACT

Transformational programming is the methodology of program development which proceeds through application of successive correctness preserving transformation rules until ultimately a desired and efficient implementation is achieved. It starts from the formal specification and ends with the optimized code. During this process numerous intermediate versions are typically developed. One of the intermediate stages of the process can involve transforming a recursive algorithm into an iterative one by applying recursion removal transformational rules. In this paper, removing recursion form programs involving arrays is discussed and rules to transform the situations of arithmetic operations application on the contents of the elements and manipulating these elements are developed. Different applications of these rules are also discussed in this paper.

Key Words: Transformational Programming, Transformation Rules, Efficient Programs, Optimized Code, Intermediate Stages, Recursion, Folding, Unfolding And Iteration.

Introduction

Program transformation has proved to be a valuable technique for developing reliable programs [Knuth 1974], [Burstall & Darlington 1977]. Using this technique, we can start with a simple and thus easy to understand specification and obtain a for more detailed but possibly more efficient program by applying a sequence of transformations which preserve equivalence. These transformations are the schemas which act as tools to reshape programs from one version to the other, performing the same operation without any deviation. They may be very simple, directly understandable by the users or very complex, difficult to fol- low. The simple rules consists of only two parts: the input template and the output tem- plate. Generally,

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

they are applied on the syntactic statements. Some of the syntactic state-ments are inter-related semantically and same operation cab be carried out by different syn-tactic terms like, interchanging assignments, absorption and expansion in control structure, etc . The complex transformation rules are divided into three parts: input template ∑, output template∑', and the applicability conditions Xs, as this notation used by Huet and Lang in [Huet & Lang 1978]. The input template is the jaw of the transformational rule, and the rule is only applicable if shape of the program where the transformational rule is to be applied resemble with it. The output template is the equivalent version of the input term-plate doing the same thing with more efficiency as its input template doing provided the conditions of application are satisfied. The program schemes ∑ and ∑' include the variables for syntactical elements. The condition part X consists of a set of propositions or attributes to express properties of the free variables contained in the program schemes.

The introduction of recursion as a conceptual tool in the design of algorithms has revolutionized program development methods and transformational programming enjoys the effective use of it. The recursive equations are extracted from the formal (axiomatic and algebraic) specifications which are then moulded into the executable code involving recur-sions. Later this recursive version of the code is transformed into an equivalent iterative version through the disciplined optimization of removing recursion and improving the efficiency of it. Jumping from the recursive to iterative stage is carried out by exploring some rules which could be applied as and when required.

Gerard Huet and Bernard Lang in their paper [Huet & Lang 1978] tried to generalize the 10 transformational rules of recursion removal designed by Darlington in his Ph.D. thesis [Darlington 1972] and they reduced their number to six, including some additional possible cases also. These rules are not sufficient to satisfy all the demands since they are not applicable in the situations involving a proper order of accessing the contents of the arrays. At the same time Martelli in [Martelli 1978] highlighted the program development through successive transformations for the processing of lists. He has complicated the situation, through involving another abstract data type, stack, in removing the recursion. Moreover, the way he has developed the algorithm is tilted totally towards mathematics.

For the purpose of this paper, we looked into the situation to diffuse the worries of complexity and found there are some transformational rules existing which are simple but more powerful without involving the other data types, like stack etc., to eliminate recursion and getting an equivalent efficient version in iterative style. There are two different rules in particular that we look at here and discuss in detail along with their applications on some peculiar examples.

Recursion and Arrays

Why recursion?

A fundamental technique in the design of efficient algorithms is recursion which is solving a problem by solving smaller versions of the same problem and must always be based upon some condition to satisfy and become well-defined recursion. This means that recursion is the calling of some program again and again directly or indirectly until the con-dition is satisfied which shows that unfolding and then folding

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

occurs during this process. Unfolding is the unwinding of algorithm and folding is the collection of all instances in the process of winding. The number of instances is determined through the number of calls made during the execution. Each instance has its own unique stores for formal parameters and local variables. Programs or their modules which exhibit this behaviour is known as the recursive ones, in which, there is one case where the value is given and one case where induction is used to express the solution in terms of values already defined.

Introduction of stepwise refinement technique for program development where we break the main problem into sub-parts and follow the top-down design methodology, we start with a very simple definition of the problem and then basing upon the data structures, we base our designs through breaking the problems into a very simple equation. Gen- erally problem is intrinsically recursive because the data structures involved are recursively defined. The recursive definition of the subprogram directly emerges from the statement of the problem and the correctness proof of such subprograms is almost trivial. Introduction of recursion in design of algorithms has revolutionized the program development methods and offer good promise for tomorrow, when we would like to see the development of pro-grams with maximum involvement of computer system relieving the programmers from laborious repetitive work. With its introduction in design of system the simplicity and con-ceptual clarity is improved. [Colussi 1984].

The requirements of the system desired is expressed in form of specifications consist-ing of instances and are expressed in form of equations. The instances on left-hand-side of these equations as sub- expressions are equivalent to right-hand-side and look like recursive equations which are then moulded into the executable code involving recursions. Later this recursive version of the code is transformed into an equivalent iterative version through dis-ciplined optimization of removing the recursion and improving the efficiency of it.

Recursion in Arrays

The algorithms having arrays as parameters exhibit a very interesting observation. For example, we have an array A as below: -

$A=[a_0,a_1,a_2 \text{ - - -}, a_n]$

If we pass this array as parameter to some algorithm and carry out some arithmetic operation recursively in proper order, say starting from a_0 to a_n, then we find that the accessing of the elements is carried out twice, once during unfolding and other during the folding. The actual calculation starts when all elements are accessed first time and the b a s e value is found on which the folding starts.

If we access the elements in the same fashion for equivalent iterative style of the pro-gram then we end up with discrepancy in response due to the peculiar nature of the recur-sion generated. As the calculation started when last element a_n is also accessed and from there it comes down folding the results. Transformational programming at one stage uses the transformational rules to eliminate the recursion from the recursive functions to make it efficient in form of iterative style.

Rules developed

The development of transformational rules to transform the recursive algorithms into

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

iterative ones involving arrays as the parameters without involving other abstract data type, like introduction stacks etc were long overdue The peculiar nature of the recursion gen-erated in such cases was needed to be encountered. Because if we keep on unfolding the function until the last value then the base value on which the folding starts is the start point of the calculations. This process of piling up the stack during the unfoldment of recur- sion can be avoided through applying the same loop but in a reverse fashion. In this way the efficiency of the program is achieved through eliminating the overloading and undesired part of computations. And it is subjected to the functions utilizing the dynamic (uncon-strained) arrays. Therefore, we have defined the data type LIST as the unconstrained array defined in the main program utilizing the functions.

We now look at the transformation rules developed. Both the rules are expressed in Ada because the recursion removal from programs in Ada is desired. In all cases it is tacti-caly assumed that the "f" in both rules is 'well- defined', which means that the recursion ultimately terminates. The first rule deals with the all subprograms having only an array as the parameter while the other works out the cases where an array and an element typed parameter are fed as the parameters to the subprograms. It is exciting to note that the rules developed do not demand any applicability condition to be satisfied. Whenever the input jaw of these rules is fixed at any program segment then they behave like universal rules for all the arithmetic operations like situations. Similarly, the function 'h' is interpreted as an operator of some kind. The parameters of functions are taken as the input parameters which are manipulated through it. The proofs of their equivalence are also included after each. The utilization of them is attempted to guage through offering their flavours in different situations.

First Transformational Rule

This rule is applicable to transform the functions having only one parameter – ARRAY. In this rule the facilities of the arrays provided in Ada like, attributes etc. are used to access the elements of it. The function "h" is the main operation carried out on its elements which may be single or composite consisting of more than one.

Recursive part of the transformation rule :

```
function f(x : ARRAY) return DATA is begin
if x 'LAST < x'FIRST
then return unit;
        else return h(x(x'
FIRST),f(x(INDES'SUCC(x'FIRST)..x'LAST)));
    end if end f;
```

Iterative part of the transformation rule

```
function f'(x : ARRAY) return DATA is
s : DATA : = unit;
i : INDEX : = x'LAST;
begin
while not (i < x' FIRST) loop s : = h(x(i), s);
i : = INDEX'PRED(i);
```

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189

http://amresearchreview.com/index.php/Journal/about

end loop; return s;
end f';
Applicability conditions
(i) f(x(x'LAST .. x'FIRST)) = unit

Proof :

To justify their equivalence, we must have a loop invariant for the iterative part of the template.
s = f(x(i+1..x'LAST))
[Gries 1981] and [McGettrick 1982] may be referred to get the answer "how".

We look at the iterative part of the template and see the following different stages through the cut point and we must prove the loop invariant is true for all the following situations
Start to cut point
Cut point back to cut point
Cut point to end
function f' (x : ARRAY) return DATA is s : DATA : = unit;
i : INDEX : = x'LAST;
begin

(CUT POINT)

while not (i < x'FIRST) loop s : = h (x(i), s);
i : INDEX' PRED(i);
end loop; return s;
end f';

Stage 1 : input assertion to cut point

We require to establish the initial truth of the loop invariant at the cut point; the loop invariant is
s = f(x(i+1 .. x' LAST))
We use the back substitution is used to drag the assertion back over the statements
i.e.
s : unit; -- an initializing value (unit is always = 0 or 1) i : x' LAST
Through back substitution the verification condition becomes out to be unit = f(x'LAST+1 .. x' LAST))
From the recursive schema

unit = f(x'LAST+1 .. x' LAST)) which is a base case, which is an applicability condition.

Stage 2: cut point to cut point

To accomplish this stage of the proof it must be shown that, assuming the truth of the invariant at cutpoint is true and when control next returns to the cutpoint the invariant is again true. In effect we have to show the assertion at the cutpoint is indeed a loop invariant.
On the first occasion that we reach the loop the following comes out:-

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

--{ s = f(x(INDEX' SUCC(i) .. x' LAST)) & not(i < x'FIRST) }.....(1)

So loop invariant now becomes h(x(i),s) = f(x(i .. x' LAST))

s:= h(x(i), s)

and the invariant now becomes s = f(x(I .. x'LAST))

i : = INDEX'PRED(i)

--{ s = f(x(INDEX' SUCC(i) .. x' LAST)) }

Through back substitution we end up with

s = f(x(INDEX'SUCC(i) .. x' LAST)) & not(i < x' FIRST)

we had s : = h(x(i), s)

therefore,

h(x(i), s) = f(x(i .. x'LAST)) .....(2)

Take LHS of equation (2), and put the values of s from eq(1), we get h(x(i), s) = h(x(i), f(x(INDEX'SUCC(i) … x'LAST)))

= f(x(i .. x'LAST)) form the recursive definition of f.

Stage 3: cut point to final assertion

s=f(x(INDEX'SUCC(i) .. x' LAST)) & i = INDEX'PRED(x'FIRST)

when we put them together we get

s = f(x(INDEX'SUCC(INDEX'PRED(x'FIRST) .. x' LAST)) s = f(x (x'FIRST .. x' LAST))

s = f(x)

Hence this rule stands right.

Applications of the rule

To determine its utility let us apply on different situations to solve the problem of recursion removal for problems involving array type structures.

Addition of the elements of the array

This rule can be applied on algorithm for adding the elements of the array, say

x = [x0.x1.x2,… .....,xn]

The recursive version of the algorithm is :-

function add_element (a:LIST) return ITEM is begin

if a'LAST < a'FIRST

then return 0;

else return a(a'FIRST) + add_element(a(a'FIRST+1..a'LAST)); end if'

end add_element;

Let us apply the above mentioned rule through matching the rule on the recursive part; we proceed as below:-

(1) Matching the recursive function with the recursive part of transformation rule

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189

http://amresearchreview.com/index.php/Journal/about

No	Actual Recursive Schema	Recursive part of the rule
1	LIST	ARRAY
2	ITEM	DATA
3	x	x
4	add_element(x)	f(x)
5	0	unit
6	x(x'FIRST)+add_element(x(	h(x(x'FIRST),f(x(INDEX'SUCC
7	INDEX'SUCC(x'FIRST)..x'LAST)	(x'FIRST)..x'LAST)

Mapping the second order variables in the iterative part of the transforma- tion rule:

```
function  add_element (a:LIST) return ITEM is s : ITEM := 0;
i : INDEX := a'LAST;
begin
while not(i < a'FIRST) loop s := a(i) + s;
i := INDEX'PRED(i);
 end loop; return s;
end add_element;
```

Similarly, other operations like, multiplication ("*") and exponentiation ("**") or any other operation can be used instead of addition ("+").

Polynomial calculation

Polynomials calculation through applying Horner's rule can be considered one of the example. We look for calculating the polynomial, three parameters are involved : First, a list of constants [a0, a1, , an]; second, base of the polynomial (x); and third, the power of the polynomial (n). If we have an array "a" of n+1 elements, called LIST, where n is the power of the polynomial then the recursive version of the program is:-

```
function poly (a:LIST; x: ITEM) return ITEM is begin
if a'LAST < a'FIRST
then return 0;
    else return a(a'FIRST) + x* poly (a(a'FIRST+1..a'LAST),x); end if'
end poly;
```

Let us apply the above-mentioned rule through matching the rule on the recursive part; we proceed as below: -

(2) Matching the recursive function with the recursive part of transformation rule

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189

http://amresearchreview.com/index.php/Journal/about

No	Actual Recursive Schema	Recursive part of the rule
1	LIST	ARRAY
2	ITEM	DATA
3	a	x
4	poly(a,x)	f(x)
5	0	unit
6	a(a'FIRST)+x*poly(a(	h(x(x'FIRST),f(x(
7	INDEX'SUCC(x'FIRST))	INDEX'SUCC(x'FIRST)
8	..a'LAST),x)	..x'LAST)

Mapping the second order variables in the iterative part of the transforma- tion rule:

```
function poly (a:LIST; x: ITEM) return ITEM is s : ITEM := 0;
i : INDEX := a'LAST
begin
while not(i < a'FIRST) loop s := a(i) + x*s;
i := INDEX'PRED(i);
 end loop; return s;
end poly;
```

Successive fraction problem

Following expression states the problem of successive fraction

1/(a0 +1/(a1 +1/(a2 + 1(a3 + 1/(a4.....))

We look at the equation and note that for calculating the continued fraction, two parameters are involved : first, a list of constant [a0, a1, …..... an]; second, the power of. If we have an array "b" of n+1 elements, called LIST, having [a0, a1, ……., an] where n is the power of the expression then the recursive version of the program is :-

```
function suc_frac (b:LIST) return ITEM is begin
if b'LAST < b'FIRST
then return 0.0;
        else return (1.0/b(b'FIRST) + suc_prod(b(INDEX'SUCC(b'FIRST)..
b'LAST))));
end if;
end suc_frac;
```

To determine its iterative version the rule is applied as:

(1) Matching the recursive function with the recursive part of transformation rule

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189

http://amresearchreview.com/index.php/Journal/about

No	Actual Recursive Schema	Recursive part of the rule
1	LIST	ARRAY
2	ITEM	DATA
3	b	x
4	suc_frac	f(x)
5	0.0	unit
6	(1.0	b(b'FIRST) + suc_frac
7	(b(SUCC(x'FIRST))	INDEX'SUCC(x'FIRST)
8	..b'LAST)))	..x'LAST))
9	h(x,y)	1.0

Mapping the second order variables in the iterative part of the transformation rule:-

function suc_frac (b:LIST) return ITEM is s : ITEM : 0.0;
I : INDEX := b' LAST;
begin
while not(i< b'FIRST) loop s := 1.0/(b(i) + s);
i := INDEX'PRED(i);
end loop; return s;
end suc_prod;

Second transformation Rule

This rule is applicable to transform the functions having two parameters – the first is of type ARRAY, and the second is of type DATA. The function "h" is the main operation carried out on its elements which may be single or composite consisting of more than one.

Representation of the rule
Recursive part of the transformation rule :
function f(y : DATA; x : ARRAY) return DATA is begin
if x' LAST < x'FIRST
then return y;
else return h(f(y, x(INDEX'SUCC(x'FIRST).. x'LAST)), x'FIRST));
end if; end f;

Iterative part of the transformation rule :
function f'(y : DATA; x : ARRAY) return DATA is s : DATA := y;
i : INDEX := x'LAST
begin
while not(i < x'FIRST) loop s := h(s, x(i)));

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189

http://amresearchreview.com/index.php/Journal/about

```
i := INDEX'PRED(i);
 end loop; return s;
end f;
```

Applicability conditions
No applicability conditions

Proof

To justify their equivalence we must identify a loop invariant for the iterative part of the template. We claim this is:-
s = f(y,x(i+1 .. x'LAST))
[Gries 1981] and McGettrick 1982] may be referred to get the answer "how" We look at the iterative part of the template and see the following different stages through the cut point and we must prove the loop invariant is true for all the following situations

Start to cut point
Cut point back to cut point
Cut point to end

```
function f'(y : DATA; x : ARRAY) return DATA is s : DATA := y;
i : INDEX : = x'LAST;
begin
                                                    (CUT POINT)
|
while not (i < x'FIRST) loop s : = h (s(i)));
i : INDEX' PRED(i);
                                          end loop; return s;
end f';
```

Stage 1 : input assertion to cut point

We require to establish the initial truth of the loop invariant at the cut point;
s = f(y, x(i+1 .. x' LAST))
We use the back substitution is used to drag the assertion back over the statements
i.e.
s : unit; -- an initializing value (unit is always = 0 or 1) i : x' LAST
Through back substitution the verification condition y = f(x'LAST+1 .. x' LAST))
From the recursive schema

y = f(y, x(x''LAST+1 .. x' LAST)) which is a base case, that is if the array has no element then the result is y, which we wanted..

Stage 2: cut point to cut point

To accomplish this stage of the proof it must be shown that, assuming the truth of the invariant at the cut point is true and when control next returns to the cut point the

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

invariant is again true. In effect we have to show the assertion at the cut point is indeed a loop invariant.
On the first occasion that we reach the loop the following comes out:-

--{ s = f(y,x(INDEX' SUCC(i) .. x' LAST)) & not(i < x'FIRST) } …….. (1)

So loop invariant now becomes h(s,x(i)) = f(y, x(i .. x' LAST))
s:= h(s,x(i))
the invariant now becomes s = f(y,x(i .. x'LAST))
i : = INDEX'PRED(i)
--{ s = f(x(INDEX' SUCC(i) .. x' LAST)) }
Through back substitution we end up with

s = f(y, x(INDEX'SUCC(i) .. x' LAST)) & not(i < x' FIRST)
we had s : = h(s, x(i))

therefore,
h(s, x(i)) = f(y, x(i .. x'LAST)) .....(2)

Take LHS of equation (2), and put the values of "s" from eq(1), we geth(s, x(i)) = h(f(y, x(i-1.. x'LAST), x(i) from the recursive definition of f, we get
h(f(y, x(i-1... x'LAST), x(i)) = f(y, x(i .. x'LAST))
which we required
Stage 3: cut point to final assertion when loop terminates tow things comes cut
s=f(y,x(INDEX'SUCC(i) .. x' LAST)) & i = INDEX'PRED(x'FIRST)
when we put them together we get

s = f(y, x(INDEX'SUCC(INDEX'PRED(x'FIRST) .. x' LAST)) s = f(y, x (x'FIRST .. x' LAST))
s = f(x) Hence this rule stands right.

Applications
An algorithm to calculate continued power
The following equation states the problem of successive

$s = ((..(x^{N1})^{N2})^{N3})^{N4})^{N5})---)^{Nn)}$

We look at the equation and note that for calculation two parameters are involved : First, a list of constant [N1, N2, ... Nn]; second, the base x. If we have an array "b" of n+1 elements, called LIST, having [Nn, Nn-1, ……., N1] where n is the total number of power involved in the expression then the recursive version of the program becomes:-

```
function contiuc_power (b:LIST; x: ITEM) return ITEM is begin
if b'LAST < b'FIRST
then return x;
        else return conti_power(b(INDEX'SUCC(b'FIRST).. b'LAST),x) **
b(b'FIRST);
```

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

```
end if;
end conti_power;
```

Let us apply the above mentioned rule through fixing the jaw of the rule on this recursive part, we proceed by:-

(1). Matching the recursive function with the recursive part of the transformation rule

No	Actual Recursive Schema	Recursive part of the rule
1	LIST	ARRAY
2	ITEM	DATA
3	b	x
4	x	y
5	conti_power(b,x)	f(x,y)
6	x	y
7	conti_power (b(INDEX'SUCC	h(f(x(INDEX'SUCC
8	(b'FIRST).. b'LAST)	(x'FIRST) ..x'LAST),y),
9	** b(b'FIRST)	(x'FIRST))
10	x ** y	h(x,y)

Mapping the second order variables in the iterative part of the transformation rule:-

```
function conti_power (b:LIST; x:ITEM) return ITEM is s : ITEM x;
i : INDEX := b' LAST;
begin
while not(i< b'FIRST) loop s := s ** a(i)'
i := INDEX'PRED(i);
end loop; return s;
end conti_power;
```

Conclusion

Concentration and focusing of many researchers like Burstall, Darlington, Huet, Lang and so many others, on removing the recursions from the programs, and enriching this cultivatable area of software engineering is offering enlightment of future hopes of system support for automatic, correct and efficient programs development. A wide spectrum of literature has been produced for this topic like, [Burstall & Darlinton 1977], [Huet & Lnag 1978], [Arsac & Kodratoff 1982], etc. Our development of two transformation rules does not claim the complete exploration of the field of recursion elimination where the arrays are the input parameters to the subprograms – functions. But it is the effort put forward to contribute, how little it may be, in enriching the field. Through examining the different applications of these rules made

Annual Methodological Archive Research Review
http://amresearchreview.com/index.php/Journal/about
Volume 3, Issue 6 (2025)

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

us feel that the rules developed are quite powerful and can handle any arithmetic operation. Their usefulness can be compared through weighing them against the rules developed by the other researchers like, [Huet and Lang 1978]. The exciting thing is the applica- bility conditions are reduced to minimum. If the functions are well defined then for each instant they work perfectly, as desired.

When we talk about system support in program development through successive application of transformations, then two main tasks the system has to perform first, choosing the suitable rule, and second, determining the applicability of the chosen rule. The first task is further divided into comparing the part of program to be transformed with the input of the transformation and selecting the appropriate rule. If the jaw of some rule get fixed on the situation then second task is carried out through proving the applicability conditions of that rule. One of our aims while working on these rules was to reduce the burden of the second task has reducing the number of applicability conditions and relieving the system through minimizing the use of mechanical proof systems. It is a matter of satisfaction for us to point out that there is no major to be satisfied before application of our rules and does not demand any mechanical system to support their application.

REFERENCE

[Clocksin & Mellish 1987]:

Colussi, L., "Recursion as effective step in program development," ACM Transactions on Programming Languages and Systems, vol. 6, No. 1, pp. 55-67, January 1984.

[Cooper 1966]:

Cooper, D.C., "The equivalence of certain computations," The Computer Journal, vol. 9, pp. 45-52, 1966.

[Darlington 1972]:

Darlington, J., "A systematic approach to automatic program improvement," Ph.D. dissertation, Department of Machine Intelligence, University of Edinburgh, Scotland, 1972.

[Darlington 1978]:

Darlington, John, "A systematic of Several Sorting Algorithms," Acta Informatica, vol.11, pp. 1-30, Springer-Verlag, 1978.

[Darlington 1972]:

Darlington, Johan, "An experimental program transformation and synthesis system." Artificial intelligence, vol. 16, pp. 99-121, north- Holland Publishing Com- pany, 1981.

[Dawes 1988]:

Dawes, John, The Professional Programmers Guide to Ada, Pitman, London, 1988.

[De Marco 1978]:

De Marco, T., Structured Analysis and System Specification, Your-don Press, New York, 1978.

[De Morgan 1976]:

De Morgen, R.M., I.D. Hill, and B.A. Wichmann, "Asupplement to the ALGOL 60 Revised Report," The Computer Journal, vol. 19,3, pp. 276-288, 1976.

Online ISSN 3007-3197
Print ISSN 3007-3189
http://amresearchreview.com/index.php/Journal/about

[Dershowitz 1982]:
Dershowitz, Nachum, "Orderings for term-rewriting systems," TheorHetical Computer Science, vol. 17 (3), pp. 279-301, 1982.

[Dershowitz 1983]:
Dershowitz, Nachum, "Applications of the Knuth-Bendix Completion Procedure," TechnicalReport ATR-83(8478)-2, Office oflaboratory Operations, The Aerospace Corporation, 1983.

[Dershowitz 1985]:
Dershowitz, Nachum, "Synthesis by completion," FifthInternational Joint Conference ArtificialIntellegence, vol. 1, pp. 208-214, 1985.

[Dershowitz 1987]:
Dershowitz, Nachum, "Termination of rewriting," J. SymbolicComputation, vol. 3, pp. 69-116, 1987.

[Dick 1984]:
Dick, A.J.J., "Equational reasoning and the Knuth-Bendix Algorithm – An informal introduction," Researchreport DOC 84/21, Department of computing, Im-perial College of Science and Technology, University of London, 180 Queen's Gate London, April 1984.

[Dijkstra 1972]:
Dijkstra, E.W., "Notes on structured programming." In StructuredProgramming by O.J. Dahl, E.W. Dijkstra, C.A.R.Hoare, Academic Pres, London, 1972.

[Dijkstra 1976]:
Dijkstra, E.W., "Why naïve transformation systems are unlikely to work?," EWD-636, privately circulated manuscript, 1976.

[Documents PROSPECTRA 1989]:
"Documents List of the PROSPECTRA Project," PROSPECTRAP.M2-MM-5.11, September, 1989.

[Dzierzgowski & Gregoie 1988]:
Dzierzgowski, Daniel and Eric Gregoire, "Formalizing software development methods," IEEE Computer Society Conferenceon System Design, pp. 230-239, IEEE service centre, 445 Hoeslane, P.O. Box 1331, PISCATAWAY, 11-14 April 1988.

[Earl et. al. 1986]:
Earl, A.N., R.P. Whittington, P. Hitchcock, and A. Hall, "Specifying a semantic model for use in an integrated project support environment," insoftwareEngineering Environments, Sommerville, I.(ed.), Pete Peregrinus, London, 1986.

[Feather 1978]:
Feather, Martin S., "ZAP transformation system: Primer and user manual," Research Report 5.4, Department of ArtificialIntelligence, University of Edin