UNIVERSIDADE DE SÃO PAULO
CAMPUS DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS DE SÃO CARLOS
I.C.M.S.C.

Algorithm development through correct transformations

MASIERO, P.C.

nº 68

Notas do ICMSC - USP

ISSN 0103-2577

Algorithm development through correct transformations

MASIERO, P.C.

nº 68

São Carlos (SP)

1990

ALGORITHM DEVELOPMENT THROUGH CORRECT TRANSFORMATIONS

Paulo Cesar Masiero
ICMSC-USP

Abstract

This paper presents an approach for algorithm development based on stepwise, previously proven correct transformations. The concept of mathematical induction is used to help creating the set of recursive equations which specifies the problem and, in some cases, to validate the equations. The approach is described and some examples are presented.

KEYWORDS:
Algorithms,
Abstract data types,
Transformation rules

1 INTRODUCTION

Safety critical systems as the avionics and embedded systems connected to defense departments systems require their basic components, the algorithms, to be developed in a rigorous manner and error free .

There exist many approaches which attempt to solve this problem, usually they are based, to some extent on mathematical formalization and computer supporting tools. The method presented in this paper is based on ideas of the project PROSPECTRA which, in turn, is based on the concept of algebraic specifications [1,3,7].

The PROSPECTRA project can be summarized as follows: one or more abstract data types are specified within the problem solution space. Then, based on the data types an abstract procedure is specified to solve the problem. The initial specification has the form of a set of axioms on which successive transformation rules are applied until an executable, efficient implementation is reached. The methodology integrates program construction and verification during the development process.

The crucial aspect of the algorithm design process of the method presented in this paper is the equation set synthesis and how to be sure it is correct, that is, it is a formal specification of the algorithm to be implemented. The concept of finite induction in mathematics can help creating the equation set, as will be seen in section 3. Furthermore, in many cases induction can be used to validate the correctness of a specification [6]. Another alternative to validate the specification is the use of a term rewriting system like the "REVE" system [5].

This paper is organized as follows: section 2 presents the method proposed and section 3 presents several examples illustrating the method. In section 4 we make some comments and assess the method presented.

2 A METHOD FOR ALGORITHM DEVELOPMENT

The problem for which we want to develop an algorithm exists in the real world and is described in an informal way. The mental process which is used to create an algorithm or to develop a system, consists usually of starting with a high level, formal specification for the problem and then refine it in a stepwise manner until an

efficient, executable version is reached. In figure 1 this idea is illustrated: going up the stairs represents the synthesis process used to derive a formal specification and going down the stairs represents the refinement process.

The four step method described below is suggested for developing algorithms:

I. Choose the abstract data types which will be used in the problem solution.

The basic abstract data types are very well known: naturals, integers, reals, sequences, mappings, etc. Abstract Data Types built from these basic types may be created when necessary. Abstract data types suitable to the problem at hand should be chosen, based on the informal description of the problem.

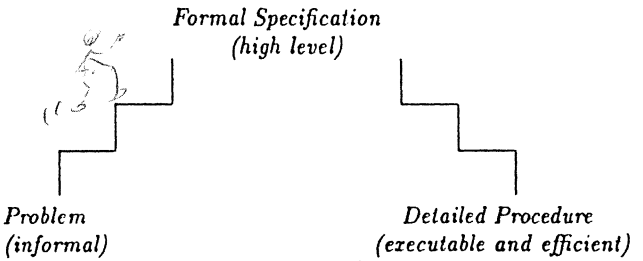

Figure 1

II. Create the recursive equation set which specifies the problem.

To accomplish this objective the concept of finite induction is suggested as the theoretical support for the creative process of developing the recursion equation set. The steps I and II are heavily dependent on the designer's ability to build the problem formal specification from its informal specification. The induction concept can be used to help in "going up the stairs " in figure 1. The application of these ideas will be shown in the next section.

III. Validate the equation set.

The very process of using finite induction to develop the equation set is, in most cases, sufficient to ensure the designer that the equation set is a formal specification of the problem at hand.

Another possible course of action to accomplish this objective is to formulate the specification in terms of a basic set of axioms and, together with a set of

auxiliary axioms, input them to a term rewrite system generator like REVE [5]. The term normalization which can be done by REVE works as a simulation (or a prototype) to validate the specification.

IV. Refine the specification until the desired level is reached.

When the recursive equation set is specified and validated, transformation rules can be applied to refine the solution to the desired level of efficiency. The transformation rules are stored in a rule bank and were proved correct before being entered in the catalogue. PROSPECTRA has a rule bank of this kind but many other rules can be found, as in [4] and [8].

The transformation rules may be classified in three different types:

a. small transformations to normalize the syntactical form of the equations.

This kind of rule is used to make small, step by step transformations. Rules 1, 2 and 3, presented in Appendix 1 are of this kind. Rule number 1, for example, can be used to normalize two conditional expressions in order to have complementary predicates.

b. recursion removal to obtain a more efficient procedure.

The rules 4, 5 and 6 presented in Appendix 1 are of this type. The removal of recursion is done with the objective of obtaining a more efficient procedure.

c. Low level transformations.

These are the more common transformation found in pre-processors or compilers. An example is the transformation from an in-line iteration by a "go to" controlled iteration, or the transformation from a high level syntax to an assembly language. This type of transformation rule will not be considered further here as it is a process which seldon cannot be completely automatized.

3 EXAMPLES

The method described in section 2 is now illustrated with a series of examples. The first of them is the algorithm to convert a binary number to its decimal equivalent. The next one is an algorithm to obtain a one-to-one mapping from an arbitrary function between finite sets and the third is the insertion of a key into a heap. A very simple algorithm, to calculate the greatest common divisor of two numbers is the last example presented to illustrate the role of double assignments in transformations.

Problem : Binary conversion

Let us consider the set $B = \{0,1\}*$. B is composed by the strings with the following structure:

$$b = b_i b_{i-1} ... b_2 b_1 b_0, i \geq 0, b_i \in \{0,1\}$$

The decimal value of b is the number:

$$d = b_i * 2^i + b_{i-1} * 2^{i-1} + ... + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0$$

A polynomial can be more efficiently computed if a transformation is applied yielding a form which is known by Horner's rule :

$$d = ((...((b_i * 2) + b_{i-1}) * 2) + ... + b_1 * 2) + b_0$$

In this case the method's first step namely: find the abstract data types in the solution space is very simple because we need only naturals with the addition and multiplication operations. The binary number can be stored in a list or in an array of natural numbers.

The recursive equation set can be created using the idea of induction over the binary number size. We know that a one digit binary number has the same decimal equivalent. If it has two digits the result will be the least significant digit added to the most significant multiplied by two an so forth.

There is still a problem to be solved since we need to know when to terminate the algorithm. When there are no more digits to be multiplied by two and added the result must be the number zero. This choice is owed to the fact that zero is the neutral element for addition in the group of naturals with the operations addition and multiplication.

The complete problem specification is then:

function c(b:list $\rightarrow$ nat) is

axiom for all u:nat; v:list $\Rightarrow$

c(nil) = 0,	(I)
c(b = u.v) = u + 2 * c(v)	(II)
	end

private

b'FIRST > b'LAST → c(b) = 0;
b'FIRST ≤ b'LAST → c(b) = b(b'FIRST) + 2 * c(b(b'FIRST+1 ...b'LAST));
end;

The notation used in the private part is similar to ADA because the PROSPECTRA project aims at generating code in ADA. The condition b'FIRST > b'LAST is used to verify if a binary number is null or the calculation has reached the end. Additionally, we are supposing that the access to the binary digits is done from the least significant to the most significant.

The proof that this set of equations correctly specifies our problem can be done by induction. An outline of this proof follows:

1. Basic Step. Let b be a binary number containing only one digit. Then, its form will be:

$b = b_0$

Applying the axioms specified above and also using axioms not introduced here, about naturals :

$$c(b.nil) \stackrel{II}{=} b_0 + 2 * c(nil) \stackrel{I}{=} b_0$$

2. Using the induction step, let us consider that the equation set is correct for a number of size i. In order to prove the induction step we need to prove that the equation set is correct for a number of size $i + 1$. Let b be this number:

$$b = b_0 b_1 b_2 ... b_{i-1} b_i = b_0 B, (where B = b_1 b_2 ... b_i)$$

The size of B is i and we know how to calculate its decimal value, which is c(B). But, if we know how to calculate this value, then we know how to calculate:

$$b_0 + 2 * c(B) = b_0 + 2 * c(b_1 ... b_i) = c(b)$$

Another way of validating the recursive equation set is to use a term rewriting system generator like REVE [5]. In Appendix 3 the algorithm to converting binary number is specified in REVE. The rewrite rules :

3. c(nil) → 0, and
6. c(u.v) → u + (s(s(0)) * c(v))

are equivalent to the rules I and II above and the other ones are auxiliary rules used to define addition and multiplication of naturals. The s operator defines the successor of a natural number. In Appendix 3, the normalization of the numbers 111 and 101 is shown. The results are expressed in terms of successors of the number 0.

Now we need to refine the algorithm specified by the recursive equation set. We are going to rewrite the equation in the private part of the procedure, utilizing the "complement" rule (see Appendix 1 to find out the transformation rules). In this example the ADA like notation will be used to derive the algorithm but in the sequel we will use the more abstract notation.

(b'FIRST > b'LAST) → c(b) = 0 ;

not (b'FIRST > b'LAST) → c(b) = b(b'FIRST) +

2 * c(b(b'FIRST+1...b'LAST));

Using now rule number 2, we obtain the following equation with condition:

c(b) = if b'FIRST > b'LAST then 0 else

b(b'FIRST) + 2 * c(b(b'FIRST+1...b'LAST));

This equation can be transformed in a procedure using rule 3:

```
function c(b:list → nat ) is
   begin
      if b'FIRST > b'LAST then
         return 0;
      else
         return b(b'FIRST) + 2 * c(b(b'FIRST+1...b'LAST));
      end if;
   end c;
```

To eliminate the recursion in this kind of procedure we need to invert the calculating order. This is a problem which is studied by Ahmed, in [8]. He has developed and proved correct the transformation rule number 6. The pattern match when applying that rule to the procedure above is the following:

T == 0

$$H(x(i),S) == x(i) + 2 * S$$

Then, we have the iterative procedure below:

```
function c(b:list → nat) is
   S:nat := 0;
   i:index := b'LAST;
   begin
      while i >= b'FIRST then
         S := b(i) + 2 * S;
         i := i-1;
      end loop;
      return S;
   end c;
```

Other transformation rules could be applied to this procedure to refine it, generating ADA code or even assembly code. The transformations were applied manually but we could be supported by an automation tool like the PROSPECTRA system. However, the designer makes most of the major decisions in each step and decides which transformation to apply.

Problem : Finding one-to-one mappings

Let f be a function that maps a finite set A into itself. Find a subset $S \subseteq A$, such that f restricted to S is one-to-one. We also want S to be the greatest set satisfying this requirement. The elements of A can be denoted by 1..n for simplicity. A solution to this problem can be found in [6]. Our solution is similar but it is not the same.

A mapping f can be represented by a set of ordered pairs, as in the example below. It should be noticed that the elements 2 and 5 are not image of any other element in A.

$$f = \{< 1,3 >, < 2,1 >, < 3,4 >, < 4,1 >, < 5,6 >, < 6,4 >\}$$

To solve this problem we need an abstract data type for finite sets which we will call "finset" and another one to represent finite mappings which will be called "grex". Grex uses as subtypes the types data, nat, bool, and finset . The specification of these abstract data types can be found in Appendix 2. The grex type

has its subtype data instantiated by nat in this particular example. The type finset, in addition to the operator and predicates listed in [3], has the procedures "any" and "dif" to get an arbitrary element from the set and to calculate the difference between two sets, respectively. The operator "any" causes the specification of finset to be not sufficiently complete but we need it specified as it is. The data type grex has additionally the procedure "rem" to remove a mapping and the procedures "d" and "im" to obtain the domain and the image of a mapping.

The creation of the recursive equation set is done observing that if the codomain of f has at least one element which is not a mapping from one element of A then this element cannot belong to S, that is, is not in the solution set. Another important observation is the fact that the condition $d(f) \supset im(f)$ is invariant while f restricted to S is not one-to-one. When $d(f) = im(f)$ becomes true the function is one-to-one.

If A has only one element then the mapping must be one-to-one and it is true that $S = d(f) = im(f)$. Let us suppose that the mapping is one-to-one for S containing n-1 elements. Now, if we have a set S with n elements and the mapping f is not one-to-one when restricted to S, then there is an element in the codomain which is not mapped by f from an element i of A. Therefore, we have n elements in the domain being mapped to at most n-1 elements into the codomain, thus it is true that d(f) $\supset$ im(f). This allow us to conclude that this element cannot be in the solution set, hence we remove it from A and also the mapping originating from it. This produces a new mapping with n-1 elements, which by the induction step we know how to calculate. It is possible to remove the element i from A because the condition that f carries on being a function is mantained since there was no element being mapped to i.

In the example above, if the element 2 which is not the image of any other element in A is removed, we have the following mapping:

$$f = \{< 1,3 >, < 3,4 >, < 4,1 >, < 5,6 >, < 6,4 >\}$$

Now 5 is the only element which is not mapped to. If we remove it we have:

$$f = \{< 1,3 >, < 3,4 >, < 4,1 >, < 6,4 >\}$$

The element 6 is now the one which is not the image of any other element and must be removed yielding the mapping:

$$f = \{< 1,3 >, < 3,4 >, < 4,1 >\}$$

Now all elements in the domain are also in the image of f and the algorithm finishes with $S = \{1,3,4\}$. Its specification is the following:

```
function map(f:grex → finset ) is

axiom for all f:grex, A,S: finset; a: data ⇒

   d(f) ⊆ im(f) → map(f) = d(f);                                  (III)
   d(f) ⊃ im(f) → map(f) = map(rem(f,any(dif(d(f),im(f))));       (IV)
                                                        end
   private
      . . .                                             end;
```

The first transformation rule to be applied to the set of recursive equations is rule 1 which arranges the predicates such that the conditions are complementary.

```
    d(f) ⊆ im(f) ⇒ map(f) = d(f);
not (d(f) ⊆ im(f))⇒ map(f) = map(rem(f,any(dif(d(f),im(f))));
```

Now the two equations can be transformed into an equation with a conditional clause:

```
map(f) = if d(f) ⊆ im(f) then d(f) else map(rem(f,any(dif(d(f),im(f)))));
```

The third rule to be applied transforms the equation with conditional clause into a procedure, yielding the following specification:

```
function map(f:grex → finset ) is
   begin
      if d(f) ⊆ im(f) then
         return d(f);
      else
         return map(rem(f,any(dif(d(f),im(f)))));
      end if;
   end map;
```

The next transformation aims to eliminate the recursive call thus yielding a more efficient solution. In this case a transformation rule for eliminating tail recursive functions is the one appropriate to eliminate the recursion (rule 4 - Appendix 1) . The pattern match in this case is:

$S == grex$

$R == finset$

$T(x) == d(x)$

$H(x) == rem(\ x,\ any(dif(d(x),im(x)))\ )$

$B(x) == d(x) \subseteq r(x)$

The final result is:

```
function map(f:grex → finset) is
   VF:grex := f;
   begin
      while d(VF) ⊃ im(VF) then
         VF := rem(VF,any(dif(d(VF),im(VF))));
      end loop;
      return d(VF);
   end map;
```

The basic set of rewrite rules to be input to the REVE system is more complex in this case because the ordering of axiom IV is not naturally from left to right and also because it needs a greater set of auxiliary rules. Rewrite rule systems which deal with conditional equations and also allow the specification of hierarchical abstract data types, as for example the CEC system described in [2] could facilitate the validation of these axioms.

The efficiency of this algorithm could be compared with the one presented in [6] if the set difference operation between two sets would not be recalculated every iteration. A solution similar to the one presented by Manber could be achieved starting from a different set of recursive equations. In order to do that we need auxiliary procedures as for example one for assigning zero to all the elements of an array; other to count the indegree of each codomain element; other to include in a queue all elements with indegree zero, etc. The recursive equations for the first two procedures are:

n = 0 → clear(g,n) = egrex;
n > 0 → clear(g,n) = put(clear(g,n-1),n,0);

n = 0 → degree(f,g,n) = f;
n > 0 → degree(f,g,n) = degree (put(f, get(g,n), 1 + get(f,get(g,n)), g, n-1);

where g,f: grex, n: nat.

Problem : Heap Data Structure

The heap data structure can be defined as an array where each key is guaranteed to be larger than the keys at other specific positions. These keys, in turn, must be larger than other two keys and so forth. This structure can be shown as a "tree" where a node contains a key which is larger than the keys of the node's two sons. A diagramatic example of such a structure is shown in figure 2.

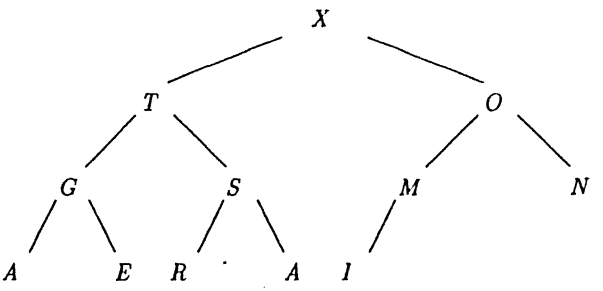

Figure 2

This binary tree can be represented within an array by putting the root at position 1, its sons at positions 2 and 3, the nodes at the next level (the sons of node in position 2) at positions 4 and 5, etc. Hence, the father of a node in position i is in position DIV(i,2), and, the two sons of the node in position i are in position 2*i and 2*i+1. For this problem we also assume that a sentinel with the maximum size allowed is in position 0. The array representation for the heap above is the following:

0	1	2	3	4	5	6	7	8	9	10	11	12
max	X	T	O	G	S	M	N	A	E	R	A	I

To build a heap we will design an algorithm for insertion. Initially, the key to be inserted is put into h[n+1], where h is the array containing the heap. We can again think in terms of induction over the tree level. In the basic step, if the tree is empty (or we are at level zero) we just need to compare the sentinel with the key to find its correct position. In a tree with $k = log_2 n$ levels we compare the key with the element at position DIV(n,2). If this element is greater than the key we insert it at position n and finish, otherwise, we swap these keys and face now the problem of inserting the key in a tree with k-1 levels which we know how to solve using the induction principle.

We will assume we have an insert function which increments n, puts max in h(0) and calls an upheap function which does the actual insertion and fixing procedures.

function upheap((h:grex; n:nat; v:data) → grex) is

axiom for all g:grex; n,m:nat; v:data ⇒

v < get(h,n/2) → upheap(h,n,v) = put(h,n,v)
v ≥ get(h,n/2) → upheap(h,n,v) = upheap(put(h,n,get(h,n/2)),n/2,v);
end

private
. . . end;

where: / represents the integer division,
n is the new position where the key will be inserted,
v is the value of the key to be inserted.

The basic data type used in this problem is grex (finite mapping). As the equations in the private part of the specification are in complementary form, we can transform them into a conditional expression, using the rule number 2 of Appendix 1.

if v < get(h,n/2) then put(h,n,v) else upheap(put(h,n,get(h,n/2)),n/2,v);

Then, we transform the conditional expression to the procedural form:

```
function upheap((h:grex, n:integer, v:data) → grex) is
   begin
      if v < get(h, n/2) then
         return put(h,n,v);
      else
         return upheap(put(h,n,get(h,n/2)), n/2,v);
      end if;
   end upheap;
```

This recursive function can again be matched to the tail recursion removal transformation rule number 4 :

S == grex#nat#data

R == grex'

B(x,y,z) == z < get(x, y/z)

T(x,y,z) == put(x,y,z)

H(x,y,z) == (put(x,y,get(x,y/2)), y/2,z)

The iteration version of the upheap function is the following:

```
function upheap((h:grex, n:nat, v: data) → grex) is
   VH:grex := h;
   VY:nat := n;
   begin
      while v ≥ get(VH,VY/2) loop
         VH := put(VH,VY,get(VH,VY/2)); (*)
         VY := VY/2;
      end loop;
      return put(VH,VY,v);
   end upheap
```

The (*) statement above could be used in the usual syntax of programming languages : VH[VY] := VH[VY/2] if we were using the private part notation to derive the algorithm. Using the abstract data type syntax, the result of applying a

function is assigned to the appropriate variable. The same comment is valid for the mapping example.

Problem : Greatest common divisor

The algorithm for calculating the greatest common divisor discovered by Euclid may be specified and refined easily using the method proposed in this paper. This example shows an interesting aspect of the recursion removal transformation rules.

The recursive equation set for specifying Euclid's algorithm is:

if v = 0 → gcd(u,v) = u;

if not (v = 0) → gcd(u,v) = gcd(v, u mod v);

Applying rules 2 and 3 from Appendix 1, we have the recursive version of this procedure:

```
function gcd((u,v: nat) → nat ) is
   begin
      if v = 0 then
         return u;
      else
         return gcd(v, u mod v);
      end if;
   end gcd;
```

Now, if we use the transformation rule to remove tail recursion (rule 4), the pattern match is:

S == nat#nat

R == nat

B(x,y) == (y = 0)

T(x,y) == x

H(x,y) == (y, x mod y)

It should be noticed that in this case both parameters are modified during

the recursive call thus needing to have both saved. The interpretation of the meta assignment "VX := H(VX)" is: the function H is applied to all parameters which are modified in the recursive call. Then, we have the following transformation for the greatest common divisor algorithm:

```
function gcd((u,v: nat) → nat) is
   VX:nat := u;
   VY:nat := v;
   begin
      while not (VY = 0) then
         (VX,VY) := (VY, VX mod VY);
      end loop;
      return VX;
   end gcd;
```

The right side of the assignment is completely calculated before the actual assignment is performed. To transform this double assignment in a sequence of two assignments we need at least one temporary variable. The direct transformation to the assignment sequence shown below would be incorrect:

```
VX := VY;
VY := VX mod VY;
```

4 CONCLUDING REMARKS

The method presented in this paper has its applicability limited when dealing with algorithm of greater size or complexity. A possible solution in this case is to divide the problem in small problems and create special specifications for each smaller problem. We are currently investigating more complex algorithms than the ones presented in this paper.

An important aspect of algorithm development using a set of recursive equations is the activity of creating the equations. This is not a simple task and depends very much on the designer's skill in dealing with abstract data types and being able to apply the induction principle in a wide range of problems. We believe

the examples shown in this paper have given insigths on how to proceed with this activity, extending the range of cases where the method can be applied successfuly.

A very positive aspect of the method is the support which can be obtained from automated tools to perform the transformations. The PROSPECTRA system currently being developed is a tool of this type.

An interesting research perspective is raised by the possibility of reusing designs: a sequence of transformation rules applied to certain problems could be stored and reused in problems that the designer thinks are similar. The existence of a great number of transformations, around 400 in PROSPECTRA, may make it difficult for the designer to remember even if the transformation process is supported by a tool. This problem should be further investigated.

References

[1] BAUER,F.L; MOLLER,B & PARTCH,H. - An overview of The Munich Project CIP: Computer-Aided, Intuition guided Programming, Technische Universitat Munchen, 1986, 22p.

[2] BERTILING,H. et al - CEC: A System to Support Modular Order-Oriented Specifications with Conditional Equations, PROSPECTRA report M.1.3-R-18.0, 1989, 44 pp.

[3] BROY, M. - The PROSPECTRA methodology: a course in 20 lectures - Part A: formal specification, Report M.2.2.S1-R-20, 1986.

[4] HUET,G. & LANG,B. - Proving and Applying Program Transformations Expressed with Second-Order Patterns, Acta Informatica, 11, 1978, pp 31-55.

[5] LESCANE, P. - Computer Experiments with the "REVE" Term Rewriting System Generator, Proceedings of the 10th POPL Conference, 1983, pp. 99-108.

[6] MANBER,U. - Using induction to design algorithms, Communications of the ACM, Vol. 31 (11), November of 1988, pp 1300 - 1313.

[7] PROSPECTRA - PROSPECTRA System: A Guided Tour, Public Report S.1.I-R-1.1, 1990, 48 pp.

[8] AHMED, T. - (Ph.D. Dissertation), In Preparation, University of Strathclyde, 1990.

5 Appendix 1

Transformation Rules

Rule 1 - Complement

$E1 < E2 \Leftrightarrow not(E1 \geq E2)$
$E1 \leq E2 \Leftrightarrow not(E1 > E2)$
$E1 > E2 \Leftrightarrow not(E1 \leq E2)$
$E1 \geq E2 \Leftrightarrow not(E1 < E2)$

Rule 2 - Conditional Equation to Condition with equation

$C \Rightarrow F = E1;$
$notC \Rightarrow F = E2;$
$\Updownarrow$
$F =$ if C then E1 else E2;

Rule 3 - Equation to Procedure

argument : a complete design specification

F(X:S) = if B(X) then EXP1(X):R else EXP2(X):R ;

result : a recursive procedure

```
function F(X:S → R) is
   begin
      if B(X) then
         return EXP1(X);
      else
         return EXP2(X);
      end if;
   end F;
```

Rule 4 - Tail Recursion to Iteration

```
function F(X:S → R) is
   begin
      if B(X) then
         return T(X);
      else
         return F(H(X));
      end if;
   end F;
```

```
function F(X:S → R) is
   VX: S := X;
   begin
      while not B(VX) then
         VX := H(VX)
      end loop ;
      return T(VX) ;
   end F ;
```

Rule 5 - Linear Recursion to Iteration

```
function F(X:S → R ) is
   begin
      if B(X) then
         return T;
      else
         return G(F(H(X)));
      end if;
   end F;
```

```
function F(X:S → R) is
   VX: S := X;
   VY: R := T;
   begin
      while not B(VX) then
         VX := H(VX);
         VY := G(VY);
      end loop;
      return VY;
   end F;
```

Rule 6 - Recursion with inversion to Iteration

```
function F(X:List → R) is
   begin
      if X'FIRST > X'LAST then
         return T;
      else
         return H(X(X'FIRST), F(X(X'FIRST + 1,...,X'LAST)));
      end if;
   end F;
```

```
function F(X:List → R ) is
    S: R := T;
    i: INDEX := X'LAST;
    begin
       while i ≥ X'FIRST then
          S:= H(X(i),S);
          i:=i-1;
       end loop;
       return S;
    end F;
```

6 Appendix 2

Abstract Data Types: GREX and FINSET

package GREX is

use DATA,NAT,BOOL,FINSET;

type grex is private;

function egrex return grex;
function put (g:grex; n:nat; d:data) return grex;
function isfree (g:grex; n: nat) return bool;
function get (g:grex; n:nat) return data; where not (isfree(g,n));
function rem(g:grex; n:nat) return grex;
function d(g:grex) return finset;
function im(g:grex) return finset;

axiom for all n,m:nat; x,y:data; g:grex ⇒

isfree(egrex,n)=true,
isfree(put(g,n,x),m) = and (not(eq(n,m)), isfree(g,m)),
eq(n,m) = true ⇒ get(put(g,n,x), m) = x,
eq(n,m) = false ⇒ get(put(g,n,x),m) = get(g,m),
eq(n,m) = true ⇒ put(put(g,n,x),m,y) = put(g,m,y),
eq(n,m) = false ⇒ put(put(g,n,x),m,y) = put(put(g,m,y),n,x),
eq(m,n)=true ⇒ rem(put(g,n,x),m) = g,
eq(m,n)=false ⇒ rem(put(g,n,x),m) = put(rem(g,m),n,x),
rem(egrex,n) = egrex,
isfree(g,n) = false ⇒ isel (d(g), n) = true,
isfree(g,n) = false ⇒ isel (im(g), get (g,n)) = true

end

```
package FINSET is

use DATA,BOOL;

type fset is private;

function eset return fset;
function join (s: fset; d: data) return fset;
function iseset (s:fset) return BOOL;
function isel ( s:fset; d:data) return BOOL;
function any ( s:fset) return data;      where not ( iseset (s) );
function dif ( s: fset, s: fset ) return fset;

axiom for all x,y: data; r,s: fset ⇒

iseset (eset) = true,
iseset (join(s,x)) = false,
isel( eset, x) = false,
eq(x,y) = true ⇒ isel(join(s,y),x) = true,
eq(x,y) = false ⇒ isel (join (s,y),x) = isel (s,x),
join ( join (s,x), y) = join( join(s,y), x),
isel(s,x) = true ⇒ join (s,x) = s,
iseset(s) = false ⇒ isel(s,any(s)) = true,
and(isel(s,x), not(isel(r,x))) = true ⇒ isel ( dif(s,r), x) = true

                                                    end
```

7 Appendix 3

Specification and Simulation of the binary conversion algorithm using the REVE system

Rewrite rules:
1. x + 0 → x
2. x * 0 → 0
3. c(nil) → 0
4. x + s(y) → s(x + y)
5. x * s(y) → x + (x * y)
6. c(u . v) → u + (s(s(0)) * c(v))
7. 1 → s(0)

Your system is complete!
→ norm
Please enter the term for which you would like the normal form computed, terminated by < *ESC* >:
c(1.(1.(1.nil)))
The sequence of term reductions leading to the normal form of your term is:
c(1 . (1 . (1 . nil)))
1 + (s(s(0)) * c(1 . (1 . nil)))
s(0) + (s(s(0)) * c(1 . (1 . nil)))
s(0) + (s(s(0)) * (1 + (s(s(0)) * c(1 . nil))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) * c(1 . nil))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) * (1 + (s(s(0)) * c(nil))))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) * (s(0) + (s(s(0)) * c(nil))))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) * (s(0) + (s(s(0)) * 0)))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) * (s(0) + 0))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) * s(0))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) + (s(s(0)) * 0))))
s(0) + (s(s(0)) * (s(0) + (s(s(0)) + 0)))
s(0) + (s(s(0)) * (s(0) + s(s(0))))
s(0) + (s(s(0)) * s(s(0) + s(0)))
s(0) + (s(s(0)) + (s(s(0)) * (s(0) + s(0))))
s(0) + (s(s(0)) + (s(s(0)) * s(s(0) + 0)))
s(0) + (s(s(0)) + (s(s(0)) + (s(s(0)) * (s(0) + 0))))
s(0) + (s(s(0)) + (s(s(0)) + (s(s(0)) * s(0))))
s(0) + (s(s(0)) + (s(s(0)) + (s(s(0)) + (s(s(0)) * 0))))
s(0) + (s(s(0)) + (s(s(0)) + (s(s(0)) + 0)))
s(0) + (s(s(0)) + (s(s(0)) + s(s(0))))

s(0) + (s(s(0)) + s(s(s(0)) + s(0)))
s(0) + s(s(s(0)) + (s(s(0)) + s(0)))
s(s(0) + (s(s(0)) + (s(s(0)) + s(0))))
s(s(0) + (s(s(0)) + s(s(s(0)) + 0)))
s(s(0) + s(s(s(0)) + (s(s(0)) + 0)))
s(s(s(0) + (s(s(0)) + (s(s(0)) + 0))))
s(s(s(0) + (s(s(0)) + s(s(0)))))
s(s(s(0) + s(s(s(0)) + s(0))))
s(s(s(s(0) + (s(s(0)) + s(0)))))
s(s(s(s(0) + s(s(s(0)) + 0))))
s(s(s(s(s(0) + (s(s(0)) + 0)))))
s(s(s(s(s(0) + s(s(0))))))
s(s(s(s(s(s(0) + s(0))))))
s(s(s(s(s(s(s(0) + 0))))))
s(s(s(s(s(s(s(s(0)))))))
Please enter the term for which you would like the normal form computed, terminated by $< ESC >$:
c(1.(0.(1.nil)))
The sequence of term reductions leading to the normal form of your term is:
c(1 . (0 . (1 . nil)))
1 + (s(s(0)) * c(0 . (1 . nil)))
s(0) + (s(s(0)) * c(0 . (1 . nil)))
s(0) + (s(s(0)) * (0 + (s(s(0)) * c(1 . nil))))
s(0) + (s(s(0)) * (0 + (s(s(0)) * (1 + (s(s(0)) * c(nil))))))
s(0) + (s(s(0)) * (0 + (s(s(0)) * (s(0) + (s(s(0)) * c(nil))))))
s(0) + (s(s(0)) * (0 + (s(s(0)) * (s(0) + (s(s(0)) * 0)))))
s(0) + (s(s(0)) * (0 + (s(s(0)) * (s(0) + 0))))
s(0) + (s(s(0)) * (0 + (s(s(0)) * s(0))))
s(0) + (s(s(0)) * (0 + (s(s(0)) + (s(s(0)) * 0))))
s(0) + (s(s(0)) * (0 + (s(s(0)) + 0)))
s(0) + (s(s(0)) * (0 + s(s(0))))
s(0) + (s(s(0)) * s(0 + s(0)))
s(0) + (s(s(0)) + (s(s(0)) * (0 + s(0))))
s(0) + (s(s(0)) + (s(s(0)) * s(0 + 0)))
s(0) + (s(s(0)) + (s(s(0)) + (s(s(0)) * (0 + 0))))
s(0) + (s(s(0)) + (s(s(0)) + (s(s(0)) * 0)))
s(0) + (s(s(0)) + (s(s(0)) + 0))
s(0) + (s(s(0)) + s(s(0)))
s(0) + s(s(s(0)) + s(0))
s(s(0) + (s(s(0)) + s(0)))
s(s(0) + s(s(s(0)) + 0))

s(s(s(0) + (s(s(0)) + 0)))
s(s(s(0) + s(s(0))))
s(s(s(s(0) + s(0))))
s(s(s(s(s(0) + 0))))
s(s(s(s(s(0)))))