MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 110

Proofs of Correctness of Dataflow Programs

by

Joseph E. Stoy
Programming Research Group
Oxford University Computing Laboratory
Oxford, England

This work was supported in part by the National Science Foundation under grant GJ-34671.

September 1974

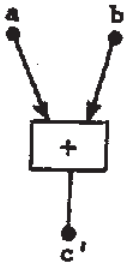
In his classic paper [1], R. W. Floyd described his "method of inductive assertions" for proving the correctness of computer programs described by flowcharts. The method involved attaching an assertion to each link in the chart, describing the state of the machine whenever control reached that point in the program. Formal rules were given for each kind of node in the flowchart, relating the assertions appearing on their input links to those on their outputs.

The approach has been considerably extended in the "axiomatic method" of C. A. R. Hoare [2]. Though Hoare's work is usually presented as applying to conventional programming languages rather than to flowcharts, in effect he has considered the usual well-structured ways of grouping together flowchart components (concatenation, conditional branching, iteration, etc), and provided rules (axioms) giving the relationships between input and output assertions for the aggregate in terms of the relationships holding for the components.

In this paper we apply similar techniques to establishing the correctness of dataflow programs. The reader is referred to Dennis [3] for an introductory description of the language we shall be considering. We too shall confine our attention to well-structured flowchart constructions: that is, we shall be concerned only with "well-behaved" dataflow programs. We explain later our reasons for making this simplification.

In the more usual control-flow kind of flowchart, it would be possible to describe the effect of each node by giving the function mapping an input state to the corresponding output state. But the state of a machine is a complicated sort of value, so it is often preferable to focus attention on some aspect relevant to the particular problem by making an appropriate assertion about the state; so we describe a node, not in terms of the transformation it effects on states, but instead in terms of the transformation on the associated assertions. In other words, instead of working with the set of states themselves, we perform manipulations in a calculus of assertions.

This technique could also be used in dataflow programs: after all, a sequential control-flow program can be regarded as a data-flow program being traversed at any one time by a single token, carrying as its value the state of the whole machine. But the values handled by normal dataflow programs are usually much simpler ones, and it is therefore more natural in most cases to return to the functional method of describing the effect of the various nodes.



So, for example, the action of the addition node is described simply by writing

$$c' = a + b.$$

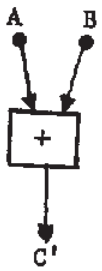
We adopt the convention of decorating with primes the names of output links, and we shall ignore for the present all matters connected with ensuring that values are of appropriate types. In some cases, to be sure, we have to supplement the defining equation by giving an axiom or two to relate the input values and the output values, but in most cases (as here) we can take for granted the axiomatic definition of the particular function (addition over the integers).

During the execution of a program, a succession of tokens appears on each link. Our use of formulae like $c' = a + b$, and the more complicated examples which we use to describe the effect of larger pieces of program, assumes that each output token can be associated with one particular token on each of the input arcs, so that the functional relationship between inputs and output may be described one token at a time. Fortunately, this is the essential property of well-behaved dataflow programs [4]. It is exhibited by all the programs we shall be considering, and by any which satisfy the following constraints:

1. Each operator actor must be well-behaved, in the sense that it produces no more than one output value on each of its output arcs whenever it absorbs one value from each of its input arcs, and the output values depend on nothing other than the corresponding set of input values; and it must be strict, in the sense that it produces no output value until all the corresponding input values have been absorbed.
2. The only composition rules employed are those to be described below.
3. The program and each procedure in it must be well-formed, in the sense that each input arc of actors within them, except for those arcs which form the input of the entire program or procedure, is connected to the output arc of some actor, and similarly that each output arc, except for those forming the output of the entire program or procedure, is connected to the input arc of at least one actor.

Notice that we do not insist that each operator actor produce exactly one output on each output arc for each set of input values. An operator actor or piece of program which fails to deliver a value on some or all of its output arcs for some set of values received on its input arcs is said to be non-terminating. A complete description of such an actor or program should include a specification of the circumstances in which non-termination may occur.

If we wished to extend our approach to cover dataflow programs which are not well-behaved, we would have to regard operator actors and programs as relating sequences of input values to sequences of output values, rather than relating individual values. Thus the addition actor would relate input sequences A and B to the output sequence C', where



$$A = \langle a_1, a_2, \ldots, a_m \rangle$$

$$B = \langle b_1, b_2, \ldots, b_n \rangle$$

$$C' = \langle c_1, c_2, \ldots, c_k \rangle$$

and

$$k = \min(m, n)$$

$$c_i = a_i + b_i \quad (1 \leq i \leq k).$$

Such a formalization would obviously be very much more complicated than our present exercise, and correctness proofs correspondingly more difficult. That is why we are restricting our attention to well-behaved programs; see Kahn [5] for a discussion of the more general case, and of the axioms (of monotonicity and continuity) which components of dataflow programs must satisfy. These axioms are automatically satisfied by well-behaved components, which will henceforth be our sole concern.

Rules for the Dataflow Language

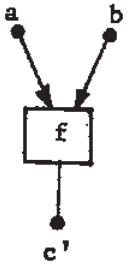
We now give all the rules we shall require for the subsequent examples. We deviate slightly from the language of Dennis in that we omit all mention of the val and ref actors: we prefer to treat such issues as pointers and sharing as matters of implementation, as the prohibition on the modification of structures removes any need to consider them explicitly. We also blur a little the distinction between operators, boolean actors and deciders.

Links

$$x = x'$$

This rule simply allows us to attach several names to the same link. This is sometimes useful, particularly when considering sections of dataflow programs connecting several inputs and outputs in which some of the inputs are passed on unchanged. In such sections we also sometimes encounter input links which terminate "in thin air". This is merely a diagrammatic convenience and has no semantic significance provided, of course, that the overall program is well-formed, in the sense defined in the previous section above.

Actors

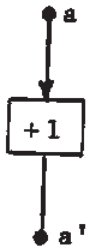


These next rules cover the binary actors, which produce an output value c' from two input values a and b. Note that for the non-commutative actors (- and >, for example) the labelling of the input links must be explicitly specified.

f

+	$c' = (a + b)$
-	$c' = (a - b)$
>	$c' = (a > b)$
=	$c' = (a = b)$
$\wedge$	$c' = (a \wedge b)$

etc.



This is an example of a unary actor, with just one input.

$$a' = a + 1$$

These are the constant actors, with no inputs. n is an integer, and

$$a' = n.$$

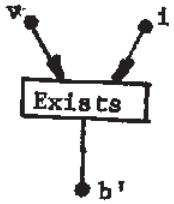
Notice that the constant actors are the only ones which may emit an output value without having first received any input.

Actors for Structure Values

The following actors deal with structure values. If v is such a value, we use v.i to denote the component of v with selector i, if such a component exists. If it does not exist, we say v.i = UNDEFINED.

$$b' = \text{Elem}(v)$$

Elem(v) is true unless v is a structure, in which case it is false.



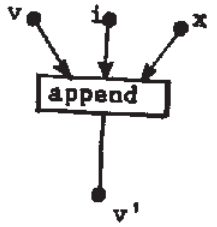
$$b' = v \text{ has } i$$

(v has i) is true if v is a structure which has a component with selector i, false if v is a structure without such a component, and UNDEFINED if v is not a structure, or i not a selector.

a' = v.i

n' = nil

This is another example of a constant actor. nil is the structure with no components.



v' = append i : x to v

(append i : x to v) is a structure which has a component with selector i whose value is x, whether or not the structure v has a component with selector i. All its other components are the same as those of v.

These actors have been described somewhat informally. Their behavior may be specified more precisely by means of the following axioms:

Elem axiom: $Elem(v) \Rightarrow \forall i : v.i = UNDEFINED$

Exists axiom: $\sim(v \text{ has } i) \Rightarrow v.i = UNDEFINED$

nil axiom: $\sim Elem(nil) \wedge (\forall i : nil \text{ has } i)$

append axiom: $v' = (\text{append } i : x \text{ to } v) \Rightarrow$

$$(v' \text{ has } i \wedge v'.i = x \wedge (\forall j : j \neq i : (v' \text{ has } j \Rightarrow v \text{ has } j) \wedge (v'.j = v.j)))$$

Note that if v'.j = UNDEFINED and v.j = UNDEFINED we say that v'.j = v.j. Note also that, as we warned earlier, we are omitting various extra conditions which may be included in the axioms to insist that the input values are of the appropriate types: that is, except for Elem, v must be a structure and i a selector, or else the result is automatically UNDEFINED.

Termination of Actors

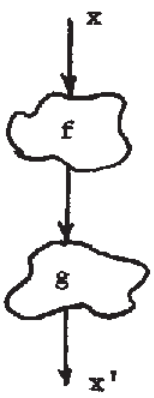
The actors we have described above always terminate; the apply actor, to be described below, is the only operator actor we shall be using which might sometimes fail to terminate. Notice that we are distinguishing the case of non-termination from the case where the output value is UNDEFINED, even though some implementations might conceivably merge the two. We are also leaving open the question of whether the occurrence of an UNDEFINED value anywhere in a program implies that the output of the program as a whole is also UNDEFINED: this is also a matter where implementations might differ.

Composition Rules

We now consider the various ways of combining pieces of dataflow program into larger ones. Frequently these pieces will connect several input with several output links. Formally, these should perhaps be described by functions relating the cartesian product spaces of the various input and output domains. However, in the sequel we shall treat the individual links separately, neglecting to invoke the necessary projection functions explicitly.

Well-formed acyclic networks

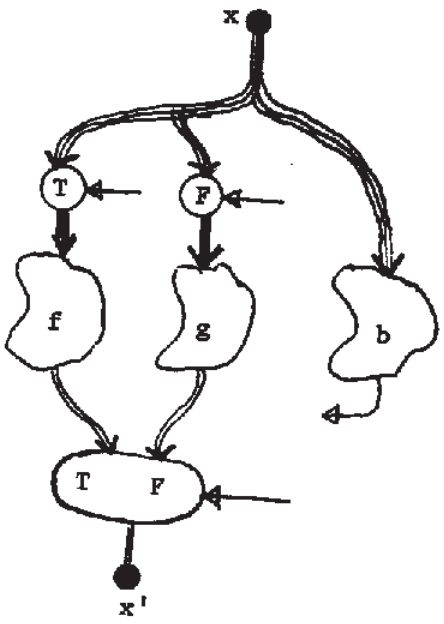
A set of actors (excluding the gate and merge actors) or groups of actors formed according to these rules may be formed into an acyclic net which is well-formed according to the definition given previously. Such a network will have the following property: if any link in the network be cut the overall result will be unchanged, provided a value y be supplied at the new input, where y is the value that will (in due course) emerge at the new output. A simple example is shown in the diagram. If f(x) is the output from group f for input x, and g(y) the output from g for input y, we have



$$(f(x) = y) \wedge (g(y) = z) \Rightarrow x' = z$$

The overall program terminates if all the components terminate.

Conditional

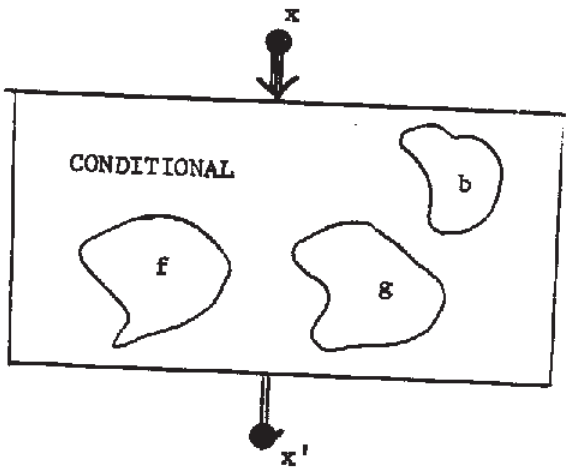


As above, areas f, g and b are pieces of data-flow programs. Area b is assumed to have a single output link, delivering boolean values. The effect is given by:

if $b(x) \Rightarrow f(x) = u$
and $\sim b(x) \Rightarrow g(x) = v$
then $x' =$ <u>if</u> $b(x)$ <u>then</u> u <u>else</u> v

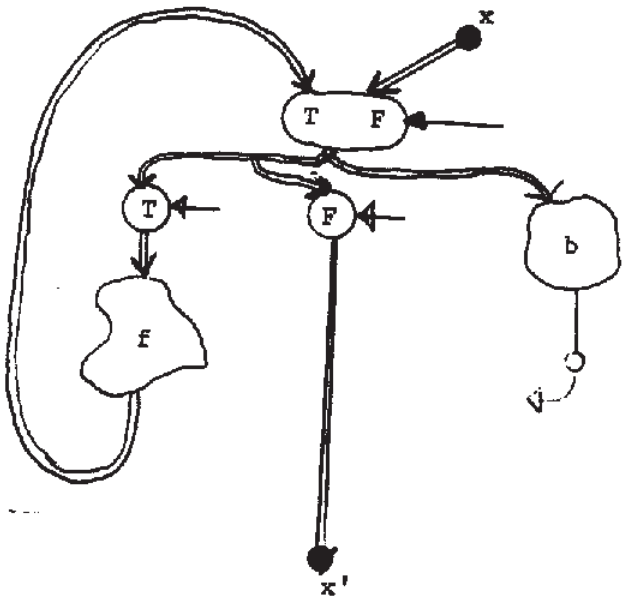
The overall structure terminates if $b(x)$ terminates for all x, and $f(x)$ terminates whenever $b(x)$ holds, and $g(x)$ terminates whenever $b(x)$ does not hold.

For purposes of simplification we introduce the following abbreviation for the diagram above.

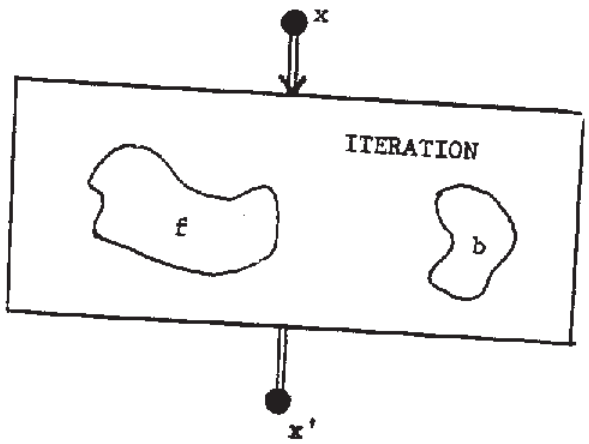


Iteration

This is the means by which loops may be introduced into our diagrams. The conventional diagram of such a schema is as follows:



We shall also use the following simpler form to represent the same schema:



The overall effect of this schema may be described, using λ-notation, as follows:

$$x' = (Y(\lambda g.\lambda y.\ \underline{\text{if}}\ b(y)\ \underline{\text{then}}\ g(f(y))\ \underline{\text{else}}\ y))x$$

where Y is the usual fixed-point operator. A full analysis of the implications of this definition would take us deep into the realm of what has come to be known as Scottery; instead of attempting this, we state two alternative definitions which may be derived from it.

(1) $$x' = f^i(x)$$

where

$$f^i(x) \quad \text{means} \quad \underbrace{f(f(f \ldots f(x) \ldots))}_{i \text{ times}}$$

and i is the least non-negative integer such that $b(f^i(x))$ is false; that is:

$$(0 \le i) \wedge (\sim b(f^i(x))) \wedge (\forall j\colon 0 \le j < i\colon b(f^j(x)))$$

If no such i exists, the program does not terminate and x' is undefined.

This alternative definition is useful only in those cases where we can find a simpler expression for $f^i(x)$, and where the particular value of i satisfying the conditions can be found by inspection.

(2) The second alternative separates the question of the existence of an output (i.e. termination of the program) from the analysis of what properties such an output would have if it did exist. To discuss the properties of the output, we return to the technique of assertions used in the analysis of control-flow programs.

If we can find some predicate P, on the set of input and output values, which is preserved by f, the body of the iteration, then this predicate is also preserved by the iteration as a whole, provided that some final output is produced. P is called the <u>invariant</u> of the iteration. In fact, P need be preserved by f only for those inputs for which b gives the result <u>true</u>; moreover, b will give <u>false</u> when applied to the final output. More formally, writing x and x' as usual for the input and output of the entire iteration, we have:

if $\forall y: P(y) \wedge b(y) \Rightarrow P(f(y))$

then $P(x) \Rightarrow P(x') \wedge \sim b(x')$.

We take this condition to be trivially satisfied if the iteration produces no final output; that is, if x' is undefined for some or all values of x.

To attack the problem of termination, we use the concept of a well-founded set, which is a partially ordered set in which there are no infinite descending chains. Then conditions for termination of an iteration are

$b(x)$ terminates for all x;

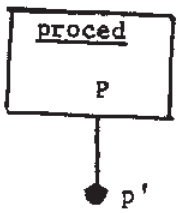
$f(x)$ terminates whenever $b(x)$ holds;

$(P(x) \wedge b(x) \wedge b(f(x))) \Rightarrow f(x) < x$, where $<$ is some ordering under which $\{x \mid P(x) \wedge b(x)\}$ is well founded.

This style of definition corresponds closest to Hoare's axiom for the while-loop [2]; it is the form we shall use in the subsequent examples.

Procedure Application

Our final set of rules concerns the production and use of procedure values. These values are created as the output of a proced actor, represented as shown

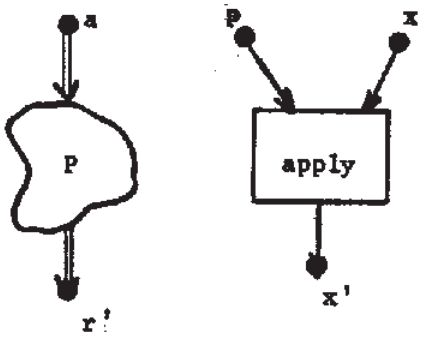


Here P is the name of some particular dataflow program, and we shall say that

p' refers to P

Notice that the proced actors are a subset of the constant actors.

The agent of procedure application is the apply actor. These actors take two inputs, p and x, say. p must refer to some dataflow program, say P. Let us suppose that P given input a produces output v'. Then the rule for the apply actor is as follows:



if p refers to P
and (a = x) ⇒ (r' = y')
then x' = y'

The apply actor terminates for inputs p and x if and only if the corresponding program P terminates for input x.

Recursive Procedures

In a recursive procedure we encounter an application of a procedure value within the dataflow program to which it refers. The overall effect of such a procedure must be proved by induction. More precisely, we must first show that the possible input values for the procedure form a set which is well-founded under some ordering <, since inductive proofs are applicable only to sets with this property. It is then necessary to show that the procedure computes the correct transformation for some input x, using the inductive hypothesis that the correct transformation is computed for all y such that $y < x$.

Sometimes, however, it is easier to do as we did for iteration, and separate the proof of "correctness" from the proof of "termination." In the present case, for the proof of correctness we assume simply that the internal calls of the procedure return the correct results, and we do not impose any conditions on their arguments. Then for termination we prove that when the procedure is applied to some argument x the internal calls will be applied to argument values which are all less than x, according to some ordering under which the set of values is well founded. More formally, let the set I index all occurrences of the apply actor within the procedure program P, so that the inputs and output of any such actor may be named p_i, x_i and x_i', for some $i \in I$. Let $Q(a, r')$ be the property we wish to prove relating the input and output of P. Then the induction rule for the proof of this property is:

if $[\forall i: i \in I: p_i \text{ refers to } P \Rightarrow Q(x_i, x_i')] \Rightarrow Q(a, r')$
then infer $Q(a, r')$.

The conditions for termination are:

(a) [$\forall i : i \in I$: P_i refers to $P \Rightarrow$ apply actor i terminates] $\Rightarrow$ P terminates

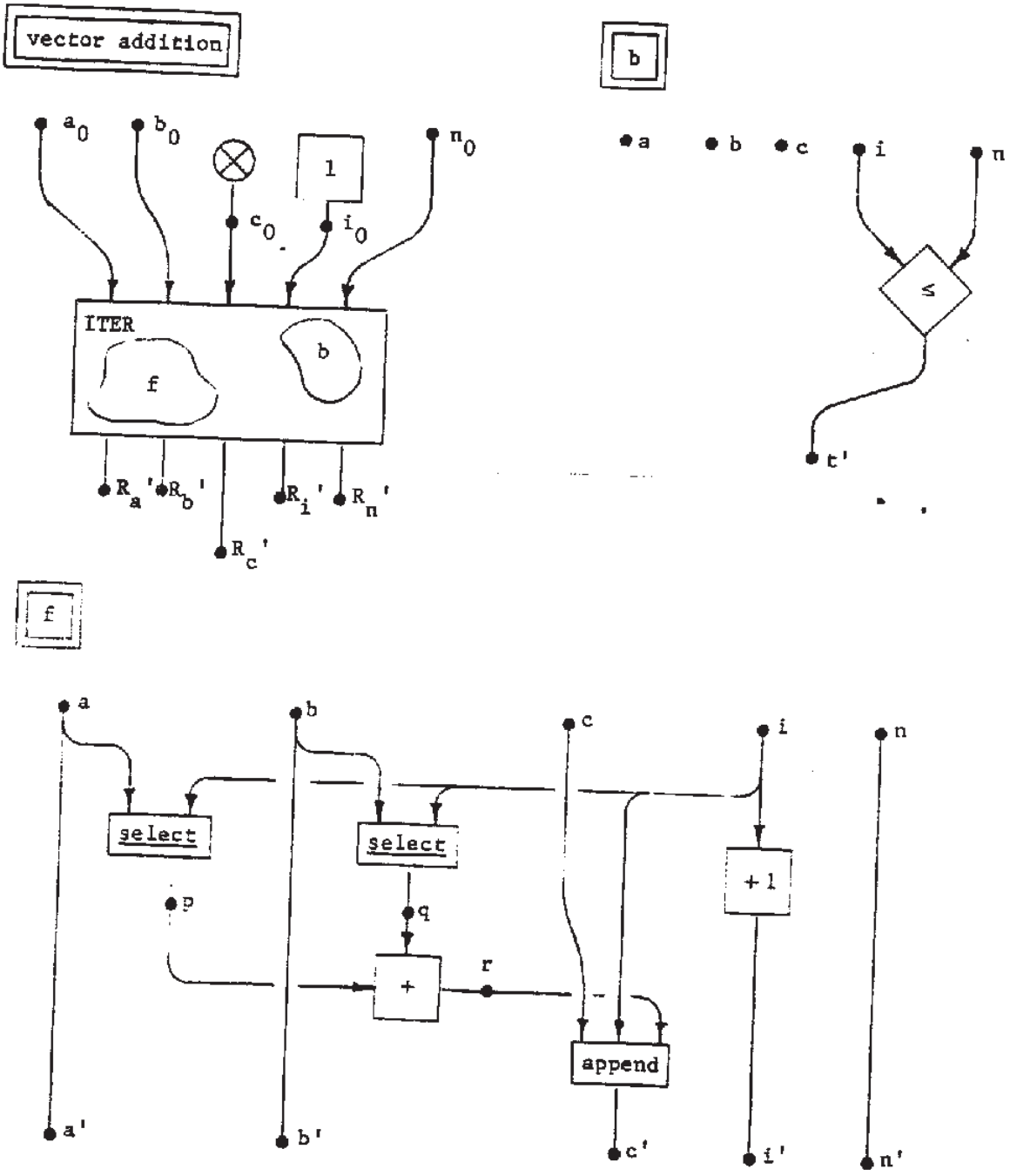
(b) [$\forall i : i \in I$: $x_i < a$] where $<$ is some ordering under which the set of possible a, x_i is well-founded.

It is this approach we shall use in the proof of our second and third examples.

Proofs of Example Programs

Vector Addition

The first example program in Dennis' paper is one for vector addition. The program is a straightforward iteration on a quintuple of values, which is shown in the diagram together with the programs for the test and body of the iteration.



a_0 and b_0 are vectors, with elements

$$a_0.1,\ a_0.2,\ \ldots,\ a_0.n_0$$

$$b_0.1,\ b_0.2,\ \ldots,\ b_0.n_0$$

We must show, in the notation of the diagram, that

$$R_c' = a_0 + b_0$$

that is,

$$\forall j:\ 1 \le j \le n_0:\ R_c' \ \underline{\text{has}}\ j \ \wedge\ R_c'.j = a_0.j + b_0.j$$

We can immediately say, still in the notation of the diagram:

$$b(a,\ b,\ c,\ i,\ n) = t' = (i \le n),$$

and, in the program for f,

$$p = a.i;\quad q = b.i;\quad r = p + q = a.i + b.i$$

(It is a condition of the problem that these components of a and b exist for all required i: we omit formally proving their existence in the induction.) c' is related to c, r and i by the append axiom, so

$$(c'\ \underline{\text{has}}\ i) \wedge (c'.i = a.i + b.i) \wedge (\forall j:\ j \ne i:\ (c'\ \underline{\text{has}}\ j \Leftrightarrow c\ \underline{\text{has}}\ j) \wedge c'.j = c.j)$$

We also have

$$a' = a;\ b' = b;\ n' = n;\ i' = i + 1.$$

We choose the following predicate as the invariant for the iteration:

$$\begin{aligned} P(a,\ b,\ c,\ i,\ n) = \quad & (1 \le i \le n + 1) \\ & \wedge (\forall j:\ 1 \le j < i:\ c\ \underline{\text{has}}\ j \wedge (c.j = a.j + b.j)) \\ & \wedge (a = a_0) \wedge (b = b_0) \wedge (n = n_0) \end{aligned}$$

We must show

(a) $\quad P(a_0,\ b_0,\ c_0,\ i_0,\ n_0)$

where c_0 = nil and $i_0 = 1$: this is trivial, assuming $n_0 \ge 0$.

(b) $\quad P(a,\ b,\ c,\ i,\ n) \wedge b(a,\ b,\ c,\ i,\ n) \Rightarrow P(a',\ b',\ c',\ i',\ n')$

Assume $P(a,\ b,\ c,\ i,\ n) \wedge b(a,\ b,\ c,\ i,\ n)$

$1 \le i$

So $1 \le i + 1$

And $i \le n$

So $i + 1 \le n + 1$

So, since $i' = i + 1$ and $n' = n$,

$$1 \leq i' \leq n + 1 \qquad (\alpha)$$

Now $\forall_j: 1 \leq j < i: c \text{ has } j \wedge (c.j = a.j + b.j)$ (from P(a,b,c,i,n))
And from above:

$$(c' \text{ has } i) \wedge (c'.i = a.i + b.i) \wedge (\forall_j: j \neq i: (c' \text{ has } j \Leftrightarrow c \text{ has } j) \wedge c'.j = c.$$

So $\forall_j: 1 \leq j \leq i: c' \text{ has } j \wedge (c'.j = a.j + b.j)$

i.e., since $i' = i + 1$ and $a', b' = a, b$:

$$\forall_j: 1 \leq j < i': c' \text{ has } j \wedge (c'.j = a'.j + b'.j) \qquad (\beta)$$

Also, since $a, b, n = a_0, b_0, n_0$ (from P(a,b,c,i,n)

and $a', b', n' = a, b, n$

we have $a', b', n' = a_0, b_0, n_0$ (γ)

(α), (β) and (γ) together give P(a', b', c', i', n'). So by the iteration rule we have

$$P(R_a', R_b', R_c', R_i', R_n') \wedge \sim b(R_a', R_b', R_c', R_i', R_n')$$

i.e. $[1 \leq R_i' \leq R_n' + 1$

$$\wedge (\forall_j: 1 \leq j < R_i': R_c' \text{ has } j \wedge R_c'.j = R_a'.j + R_b'.j)$$

$$\wedge (R_a' = a_0) \wedge (R_b' = b_0) \wedge (R_c' = c_0)]$$

$$\wedge [\sim (R_i' \leq R_n')]$$

So $R_i' = R_n' + 1$, and we have

$$\forall_j: 1 \leq j \leq n_0: (R_c' \text{ has } j) \wedge (R_c'.j = a_0.j + b_0.j)$$

as required.

Termination

The test and body of the iteration always terminate.

Now let $v_j = \langle a_j, b_j, c_j, i_j, n_j \rangle$ and let $v_1 < v_2$ iff $(n_1 - i_1) < (n_2 - i_2)$. Then $\{v \mid P(v) \wedge b(v)\}$ is well-founded, since from P(v) we have

$$(n - i) \geq -1.$$

Also, for the program f,

$$n' = n \text{ and } i' = i + 1$$

so

$$(n' - i') < (n - i)$$

So $v' < v$, and termination is assured.

REVERSE

A binary tree is either an elementary value or a structure with two components, called "right" and "left," each of which is a binary tree.

Tree t_1 is contained in tree t_2 ($t_1 < t_2$) if t_1 is a component of t_2 or is contained in a component of t_2. The set of finite binary trees is well-founded under this ordering.

The reverse of a finite binary tree t is the finite binary tree t_1 defined as follows: if t is an elementary value, then t_1 = t; if t is a structure, then

$$t_1.\text{'right'} = \text{reverse } (t.\text{'left'})$$

and

$$t_1.\text{'left'} = \text{reverse } (t.\text{'right'}).$$

We wish to prove that the program Reverse constructs the reverse of its finite binary tree argument T_r. Reverse is defined using the auxiliary procedure Rev_1; the definitions are given in the diagrams on page 18.

We first prove of Rev_1:

let A be such that

A.'r1' refers to Rev_1
A.'list' is a fin.bin.tree;

then A' = Reverse (A.'list')

Inductive hypothesis:

let I = {1, 2} index the two apply actors in the Rev_1 program. Then:

$$\forall i : i \in I: [(p_i \text{ refers to Rev\_1} \wedge A_i.\text{'r1' refers to Rev\_1} \wedge A_i.\text{'list' is a fin.bin.tree}) = A_i' = \text{Reverse}(A_i.\text{'list'})]$$

Now x,y = A, A.'list' (identity, and select)

and A' = if b(x,y) then f(x,y) else g(x,y) (conditional)

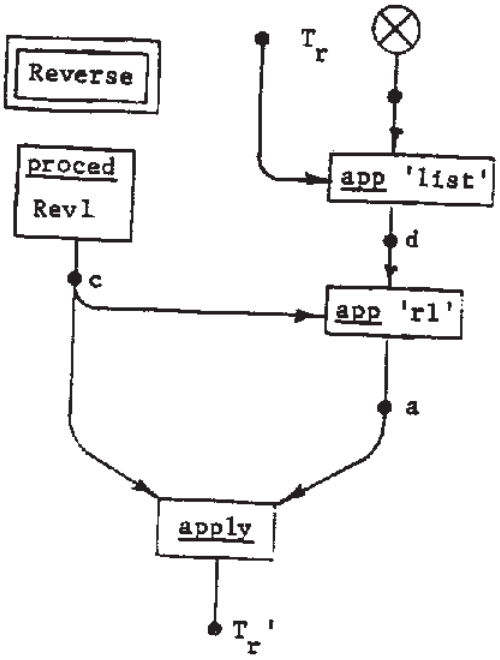
where b(x,y) = Elem(y)

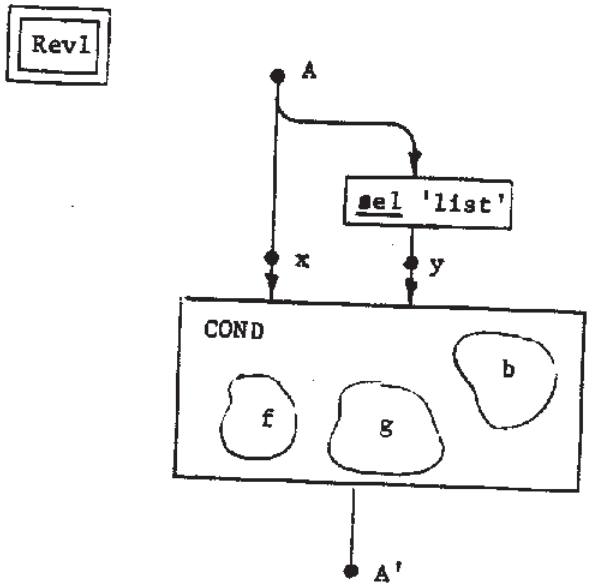
and f(x,y) = y

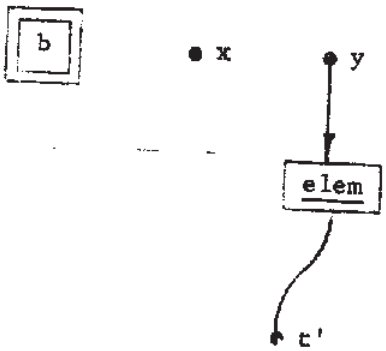
And, in the program for g(x,y):

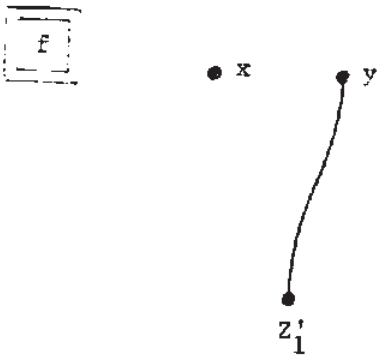
m,n = y.'left', y.'right' (select)

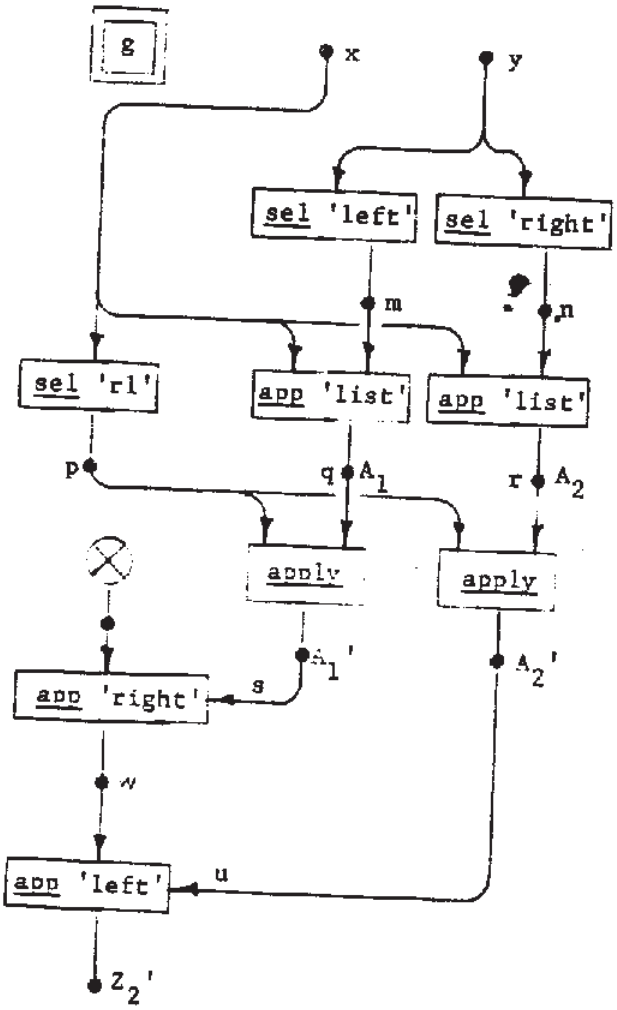
So (q has 'list') $\wedge$ (q.'list' = m) $\wedge$ [$\forall_j$: j $\neq$ 'list': (q has j $\Leftrightarrow$ x has j) $\wedge$ (q.j = x.j)] (append)


Reverse
proced
Revl
T_r
app 'list'
d
c
app 'rl'
a
apply
T_r'


Revl
A
sel 'list'
x
y
COND
b
f
g
A'


b
x
y
elem
t'


f
x
y
Z_1'


g
x
y
sel 'left'
sel 'right'
m
n
sel 'rl'
app 'list'
app 'list'
p
q
A_1
r
A_2
apply
apply
A_1'
A_2'
app 'right'
s
w
app 'left'
u
Z_2'

But (x has 'r1') ∧ (x.'r1' refers to Rev_1) (Assumption)
So q.'list' is a fin.bin.tree (assumption) and q.'r1' refers to Rev_1.
But p = x.'r1' (select)
So p refers to Rev_1 (assumption)
But $p = p_1$, $q = Q_1$ and $A_1' = S$
So, by the inductive hypothesis

s = Reverse(q.'list')

i.e. s = Reverse(y.'left') (since q.'list' = m = y.'left')

Similarly,

u = Reverse(y.'right')

Now (w has 'right') ∧ (w.'right' = s) (append)
and $(z_2'$ has 'left') ∧ (z_2'.'left' = u)
∧ [$\forall_j$: j ≠ 'left': (z_2' has j ⇔ w has j) ∧ (z_2'.j = w.j)]
So z_2'.'left' = Reverse(y.'right')
∧ z_2'.'right' = Reverse(y.'left')
But A' = if Elem(y) then y else z_2'
So A' = Reverse(y) = Reverse(A.'list')
So by the induction rule: A' = Reverse(A.'list')

TERMINATION OF REV_1

Rev_1 certainly terminates if the apply actors inside it terminate. Now let A < B iff A.'list' <' B.'list' where <' is the usual ordering on binary trees. The trees are finite, so the set is well-founded under <', so the set of permissible A is well-founded under <. For the two apply actors in Rev_1:

1. q.'list' = (A.'list').'left', so q.'list' <' A.'list', so $A_1 < A$
2. Similarly $A_2 < A$

So ∀i ∈ I: $A_i < A$, and termination is assured.

For the Reverse program itself:

let Tr be a fin.bin.tree.

We have c refers to Rev_1 (proced)
and, by two applications of the append rule,

a.'r1' = c ∧ a.'list' = Tr

so a.'r1' refers to Rev_1 ∧ a.'list' is a fin. bin.tree
So, by the rule for apply:

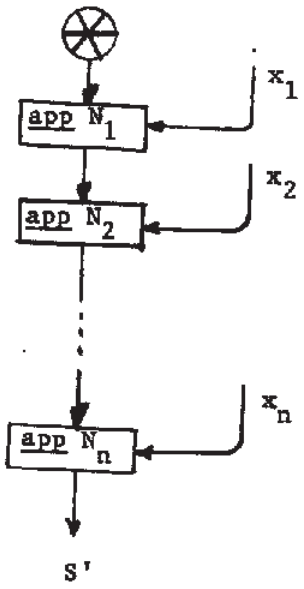
Tr' = Reverse(a.'list')
Tr' = Reverse(Tr)

THE EIGHT QUEENS' PROBLEM

Our final example concerns a program for the Eight Queens' problem, discussed by Dijkstra [6]. We discuss a program somewhat modified from the version of Dennis [3]; the present version creates a data structure containing all the correct configurations, instead of printing them all. This allows us to defer for the present any consideration of I/O, and also has the effect of remedying the troublesome absence of output links from some of the procedure applications.

The program is defined in the diagrams on pages 21-26.

In this program we frequently construct new structures by segments of the form shown in the figure, where the $N_i (1 \le i \le n)$ are any set of distinct names, and the x_n $(1 \le i \le n)$ are any set of values. The final result, S', has the following property:



$$(\forall i: 1 \le i \le n: S' \text{ has } N_i \wedge S'.N_i = x_i)$$

$$\wedge\ (\forall M: S' \text{ has } M \Leftrightarrow (\exists i: 1 \le i \le n: M = N_i))$$

That is, the only components of S' and those with selectors from the set N_i, and for one of these the value is the corresponding x_i. The proof, by induction on n using the definition of the append actor, is simple. We shall write

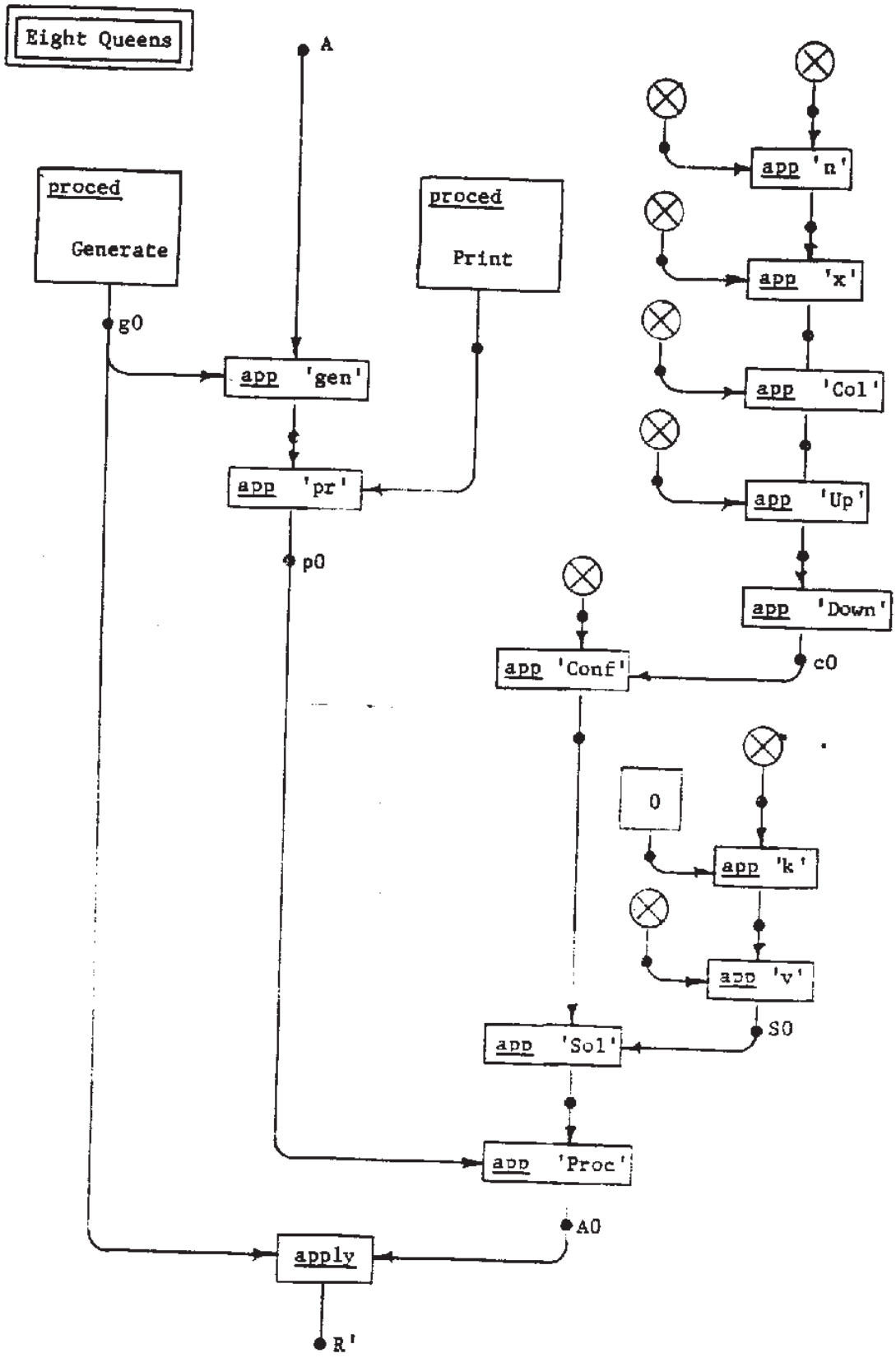
$$S' = \text{Structure } (N_1: x_1, N_2: x_2, \ldots, N_n: x_n)$$

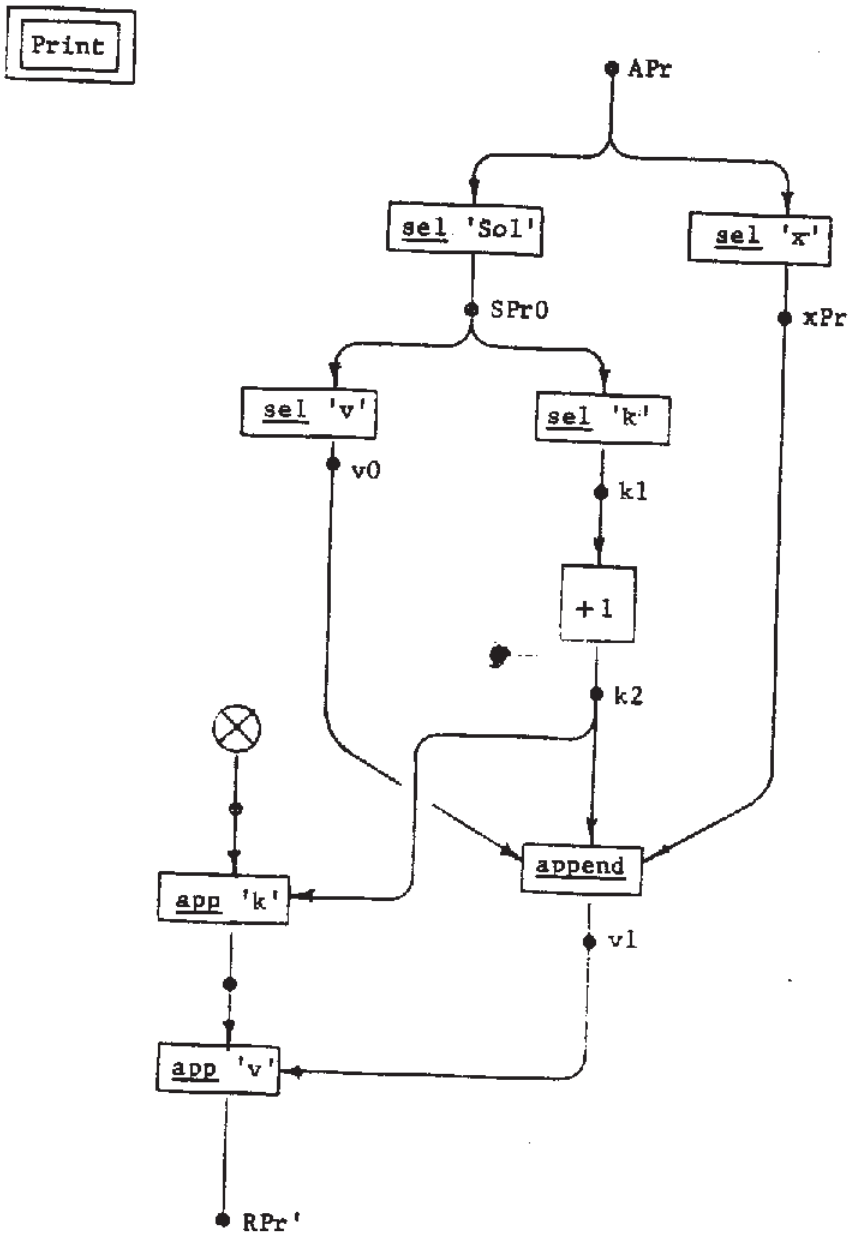
to indicate that S' is the result of a program piece of this form.

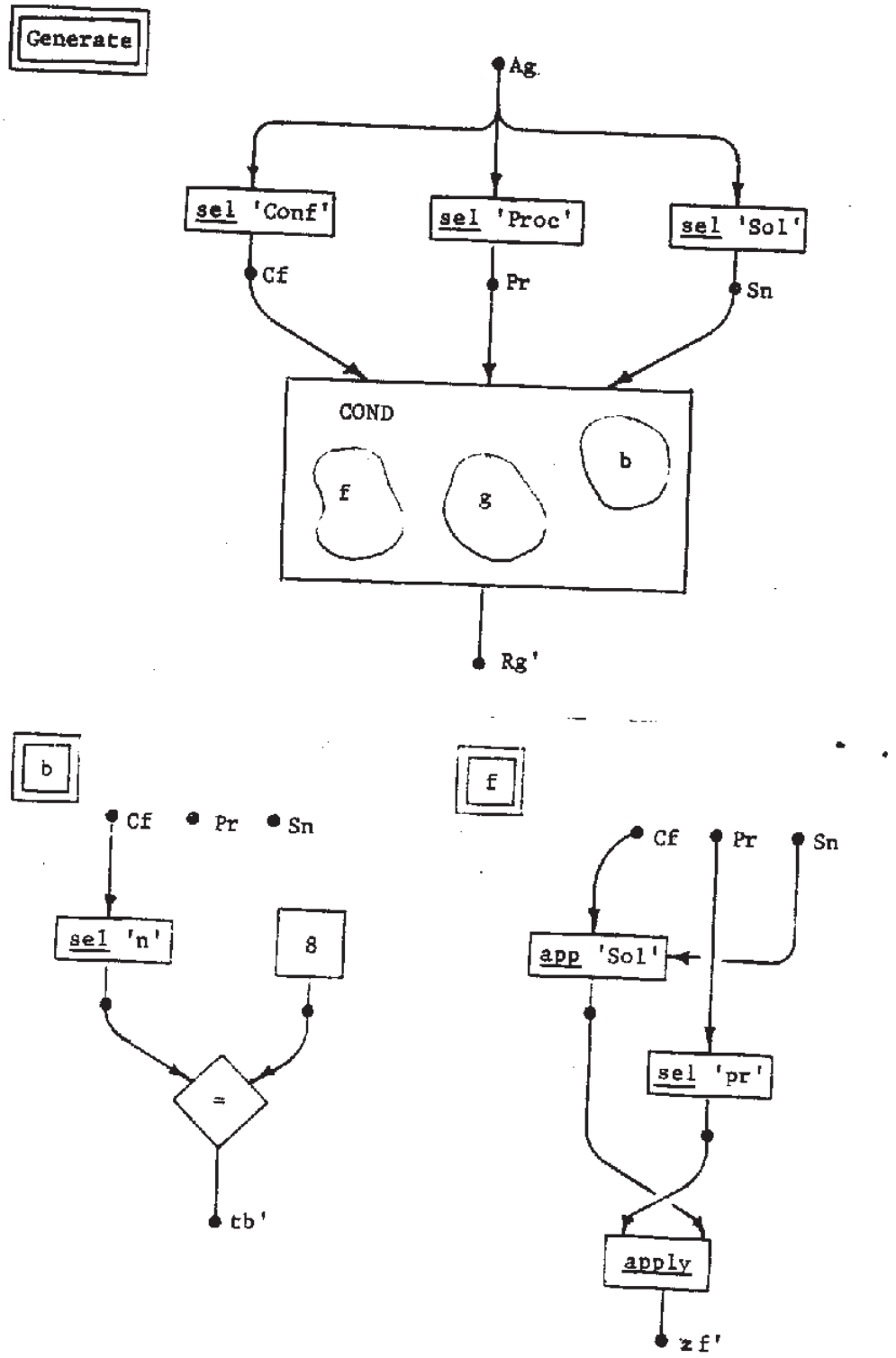
We shall now prove that this program is correct. The proof is more complicated than might initially be imagined, so in order to emphasize the main points of the argument we have relegated the detailed proofs to an appendix.

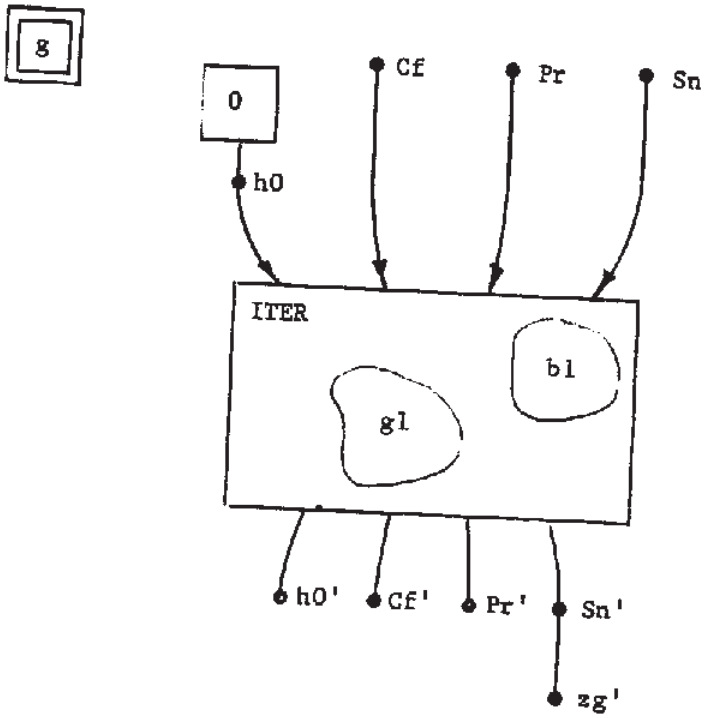
We must prove four things about the program:

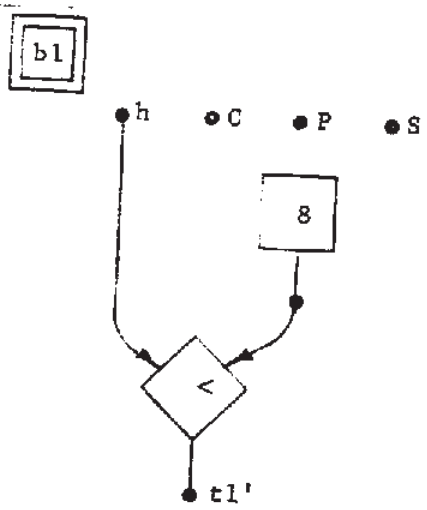
1. that it terminates;
2. that only correct solutions occur in the final solution data structure;
3. that all such correct solutions so occur;
4. that no solution occurs more than once.

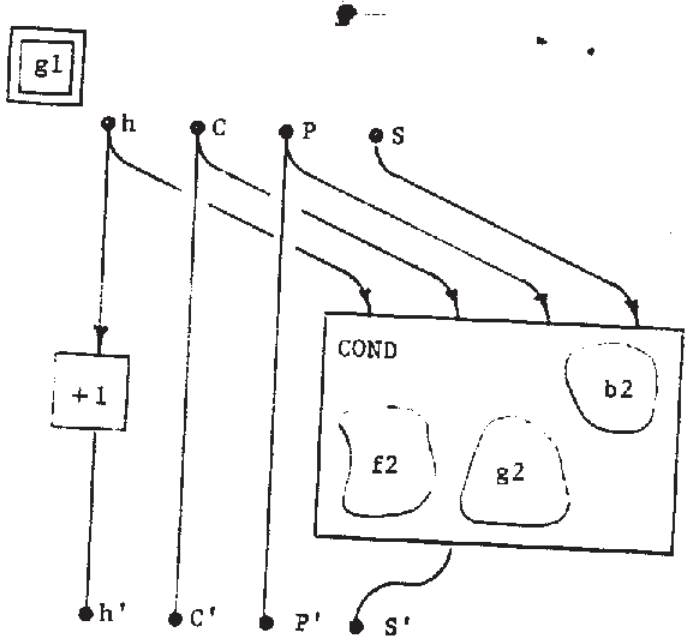

Eight Queens
A
proced
Generate
proced
Print
g0
app 'gen'
app 'pr'
p0
app 'n'
app 'x'
app 'Col'
app 'Up'
app 'Down'
c0
app 'Conf'
0
app 'k'
app 'v'
S0
app 'Sol'
app 'Proc'
A0
apply
R'

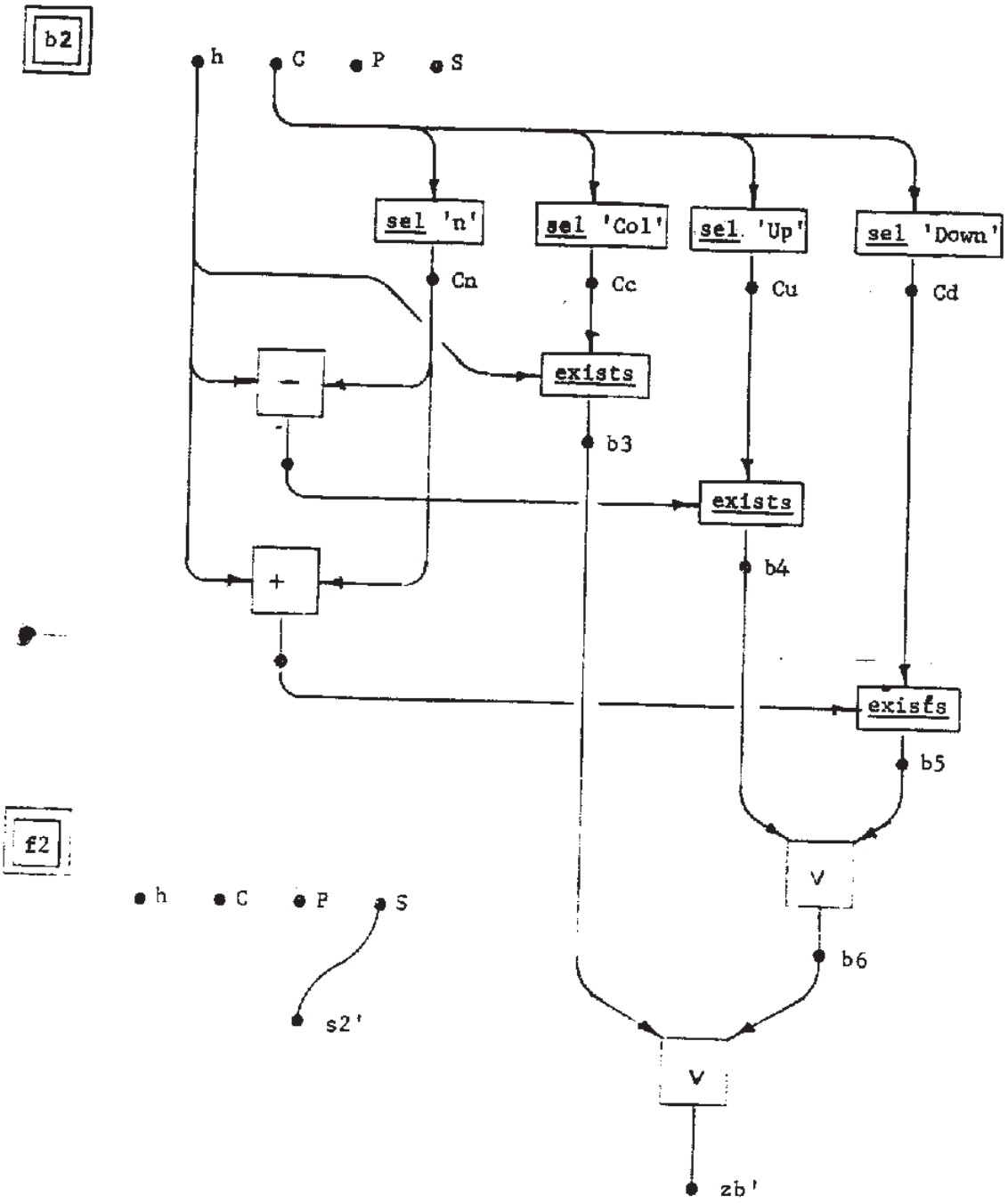

Print
APr
sel 'Sol'
sel 'x'
SPr0
xPr
sel 'v'
sel 'k'
v0
k1
+1
k2
append
app 'k'
v1
app 'v'
RPr'

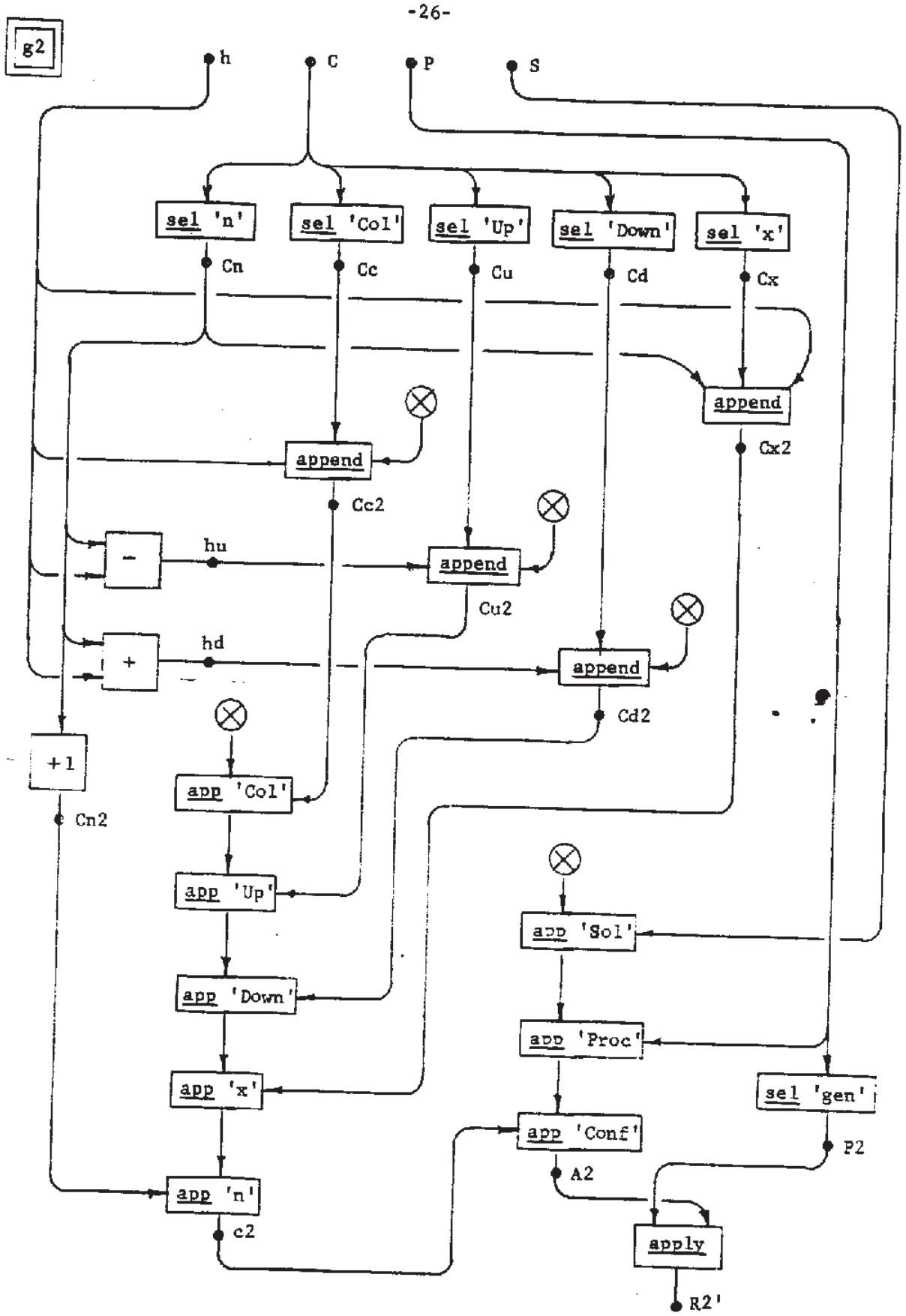

Generate
Ag
sel 'Conf'
sel 'Proc'
sel 'Sol'
Cf
Pr
Sn
COND
f
g
b
Rg'
b
Cf
Pr
Sn
sel 'n'
8
=
tb'
f
Cf
Pr
Sn
app 'Sol'
sel 'pr'
apply
zf'


g
0
Cf
Pr
Sn
h0
ITER
b1
g1
h0'
Cf'
Pr'
Sn'
zg'


b1
h
C
P
S
g
<
t1'


g1
h
C
P
S
+1
COND
b2
f2
g2
h'
C'
P'
S'


b2
h
C
P
S
sel 'n'
sel 'Col'
sel 'Up'
sel 'Down'
Cn
Cc
Cu
Cd
-
exists
b3
exists
b4
+
exists
b5
V
b6
V
zb'
f2
h
C
P
S
s2'


g2
h
C
P
S
sel 'n'
sel 'Col'
sel 'Up'
sel 'Down'
sel 'x'
Cn
Cc
Cu
Cd
Cx
append
Cx2
append
Cc2
-
hu
append
Cu2
+
hd
append
Cd2
+1
Cn2
app 'Col'
app 'Up'
app 'Down'
app 'x'
app 'n'
c2
app 'Sol'
app 'Proc'
app 'Conf'
A2
sel 'gen'
P2
apply
R2'

Our solution structure, S, has two components, S.'k' and S.'v'. S.'k' is a non-negative integer, and S.'v' has components

$$(S.'v').1,\ (S.'v').2,\ \ldots,\ (S.'v').(S.'k')$$

each of which is a correct solution to the problem. To express the property that S is partially valid (i.e. that it contains no mistakes but is not necessarily complete) we define the predicate SPValid(S) as follows:

Defn. 1. Let k, v = S.'k', S.'v'.

Then $\text{SPValid}(S) \equiv k \geq 0 \wedge (\forall j: 1 \leq j \leq k: V \text{ has } j \wedge \text{Correct}(V.j)) \wedge (\forall i,j: 1 \leq i,j \leq k: i \neq j \Rightarrow V.i \neq V.j)$

It will be noted that SPValid is defined in terms of Correct, a predicate on individual solutions, which will be defined below. A proof that SPValid holds of the final result would satisfy requirements 2 and 4 above.

We define the binary predicate contains to express the fact that a particular solution occurs in the solution structure.

Defn. 2. $S \text{ contains } X \equiv \exists i: 1 \leq i \leq S.'k': (S.'v').i = X$

We can now prove the property we require of the procedure Print. For the nomenclature we refer the reader to this procedure's dataflow diagram.

Lemma 1. Let S,X = APr.'sol', APr.'x'

(1) $\text{SPValid}(S) \wedge \text{Correct}(X) \wedge \sim(S \text{ contains } X) \vdash\!\!\!- \text{SPValid}(RPr') \wedge \forall Y: (RPr' \text{ contains } Y \Leftrightarrow S \text{ contains } Y \vee Y = X)$

(2) Print terminates.

This states that the result RPr' differs from S only in that the individual solution X has been added; if S is valid then so is RPr', provided the addition of X does not cause a repetition. The proof appears in full in the appendix.

We turn now to the definition of Correct(x), which is to express the fact that X is a valid individual solution to the problem. We assume that X is a structure with components

$$X.0,\ X.1,\ \ldots,\ X.7$$

each of which is an integer such that X.i gives the index number of the column occupied by the queen in row i. The solution is correct if no two queens are on the same column or diagonal (the requirement that no two be on the same row is of course enforced by our choice of representation --see Dijkstra op. cit.).

It is convenient to formalize this as a property of a partial solution. PCorrect(X) expresses the fact that the first few rows of X contain queens which do not clash in any of the forbidden ways:

Defn. 3. PCorrect(X) ≡

$$\exists n: \{0 \le n \le 8 \wedge (\forall i: X \text{ has } i \Leftrightarrow 0 \le i < n) \wedge (\forall i: 0 \le i < n: 0 \le X.i \le 7) \wedge (\forall i,j: 0 \le i,j < n: i \ne j \Rightarrow (X.i \ne X.j \wedge X.i + i \ne X.j + j \wedge X.i - i \ne X.j - j))\}$$

Note that for an upward diagonal X.i - i is constant, and for a downward diagonal X.i + i is constant. The definition of Correct (X) is then simply as follows:

Defn. 4. Correct(X) ≡ PCorrect(X) ∧ (∀i: X has i ⇔ 0 ≤ i < 8)

It is convenient for the program to construct each solution as part of a more complicated configuration. If Config is such a structure, then Config.'x' is the solution under construction, and Config.'n' is an integer giving the number of queens so far in the solution. Config also has three more components, all of which are themselves structures:

Config.'Col', Config.'Up' and Config.'Down'.

The component

(Config.'Col').i (0 ≤ i ≤ 7)

exists only if column i is occupied by one of the queens in the solution so far;

(Config.'Up').i (-7 ≤ i ≤ +7)

exists only if the upward diagonal i is occupied (i.e. if (Config.'x').j - j = i for some j between 0 and Config.'N' - 1); and

(Config.'Down').i (0 ≤ i ≤ 14)

exists only if downward diagonal i is occupied. These extra components allow an easier decision as to whether a new queen may be placed on a particular square.

The condition for correctness and consistency of all these components is given by the predicate PCConf(C) on a partial configuration, defined as follows:

Defn. 5. Let X, N, COL, UP, DOWN
= Config.'x', Config. 'n', Config.'Col', Config.'Up', Config.'Down'

Then PCConf(Config) ≡ PCorrect(X) ∧ (∀i: X has i ⇔ 0 ≤ i < N)
∧ (∀k: 0 ≤ k ≤ 7: COL has k ⇔ (∃i: 0 ≤ i < N: X.i = k))
∧ (∀k: -7 ≤ k ≤ 7: UP has k ⇔ (∃i: 0 ≤ i < N: X.i - i = k))
∧ (∀k: 0 ≤ k ≤ 14: DOWN has k ⇔ (∃i: 0 ≤ i < N: X.i + i = k))

The condition that the configuration is complete is given by the predicate CConf (Config), defined as follows:

Defn. 6. CConf(Config) ≡ PCConf(Config) ∧ (Config.'n' = 8)

Lemma 2. CConf(Config) ⇒ Correct(Config.'x')

This result is an immediate corollary of Defns. 6, 5 and 4.

The lemma which justifies the introduction of the extra components of a configuration (on the grounds that they make it easier to test whether the addition of a new queen would preserve correctness) is as follows:

Lemma 3. Let N, X, COL, UP, DOWN
= Config.'n', Config.'x', Config.'Col', Config.'Up', Config.'Down'

Then 0 ≤ h ≤ 7 ∧ PCConf(Config) ∧ N < 8 ⊢——
~ (COL has h) ∧ ~ (UP has h-N) ∧ ~ (DOWN has h + N)
⇔ PCorrect (append N:h to X)

The proof of this lemma is given in the appendix.

When adding a new queen it is of course not sufficient merely to append a new component to the solution itself: we must also take care to make all the components of the new configuration consistent. To express this we define a function

Add New Queen (Config, h)

which produces a new configuration in which a queen has been added in column h of the next available row. It is defined as follows:

Defn. 7. Let N, X, COL, UP, DOWN
= Config.'n', Config.'x', Config.'Col', Config.'Up', Config.'Down'
Then AddNewQueen (Config, h) = Structure ('n': N + 1,
'x': append N:h to X,
'Col': append h: nil to COL,
'Up': append h-N: nil to UP,
'Down': append n+N: nil to DOWN)

The following lemma guarantees the correctness of the result when this function is appropriately applied.

Lemma 4. Let N, COL, UP, DOWN
= Config.'n', Config.'Col', Config.'Up', Config.'Down'
Then $0 \leq h \leq 7 \wedge$ PCConf (Config) $\wedge N < 8$
$\wedge \sim$(COL has h) $\wedge \sim$(UP has h-N) $\wedge \sim$(DOWN has h+N) $\vdash$
PCConf(AddNewQueen (Config,h))

This lemma is also proved in the appendix.

The partial solution in the new configuration produced by AddNewQueen is an extension of that in the old one. It is convenient to generalize this notion, so we define extends as follows.

Defn. 8. X extends Y $\equiv$
$\forall i$: Y has i = X has i $\wedge$ X.i = Y.i

Obviously this relation is reflexive and transitive. Its connection with AddNewQueen is given by the following lemma, also proved in the appendix.

Lemma 5. PCConf(Config) $\vdash$
(AddNewQueen(Config, h)).'x' extends Config.'x'

Defn. 8 states that if X extends Y then X and Y have all the elements of Y in common. By removing one at a time the extra elements in X, we may generate a whole sequence of partial solutions until we reach Y. Moreover, if X is partially correct and Y is such that we may reach it from X by always removing the element X.n of highest numbered n, then all the intermediate solutions will be partially correct. So, in particular, we have

Lemma 6. X extends Y ∧ Correct(X)
∧ [∃n: 0 ≤ n ≤ 8 ∧ (∀i: Y has i ⇔ 0 ≤ i < n)]
⇒ PCorrect(Y)

This result gives us a strategy. To generate all the complete correct solutions which are extensions of a given partially correct solution we may reject, whenever we add a queen, all those resulting solutions which are not partially correct, since by Lemma 6 no extensions of such solutions can be correct. This strategy is carried out by the procedure Generate, which is a recursive procedure and must therefore be proved by induction. We prove that Generate has the following property:

Lemma 7. Let S, C, P = Ag.'Sol', Ag.'Conf', Ag.'Proc'
Then (1) SPValid(S) ∧ PCConf(C) ∧ ProcOK(P)
∧ (∀X: Correct(X) ∧ X extends C.'x' ⇒ ~(S contains X))
⇒ SPValid(Rg') ∧ (∀Y: Rg' contains Y ⇔ S contains Y
∨ (Correct(Y) ∧ Y extends C.'x'))

(2) Generate terminates, at least whenever PCConf(C).

Here the identifiers are as defined by our labelling of the links in the dataflow diagrams for Generate. All the predicates have already been defined except for ProcOK: This is to express the fact that the components of Ag.'Proc' refer to the appropriate procedures, and is defined as follows:

Defn. 9. ProcOK(P) ≡ P.'pr' refers to Print
∧ P.'gen' refers to Generate

The inductive proof of Lemma 7 is in the appendix. The Lemma states that a particular solution occurs in the final result either if it already occurred in Ag.'Sol' or if it is a correct extension of (Ag.'Conf').'x'. Since all correct solutions are extensions of the empty partial solution (that is, the empty board), we can generate the complete set by starting with the empty solution structure and the empty partial configuration. We are, therefore, able to construct the main program for this problem. However, one small difficulty arises. The Eight Queens' problem is one which has no parameters: nevertheless, dataflow programs require the appearance of a token on an input link to trigger them off. So, since we do not wish our program to be continually generating sets of solutions to the problem, we somewhat artifically choose one of the links (A in the diagram) to be the input.

We may prove the following about this main program:

Theorem (1) A = nil

SPValid(R') ∧ (∀Y: R' contains Y ⇔ Correct(Y))

(2) The main program terminates.

This theorem (proved in the appendix) states the condition of correctness for the program: the result contains all the correct solutions and no others, there are no duplications, and the program terminates. We have thus fulfilled our initial goal.

Appendix -- Proofs of the Theorem and Lemmata

Lemma 1. Let S,X = APr.'Sol', APr.'X'

(1) SPValid(S) ∧ Correct(X) ∧ ~(S contains X) ⊢
SPValid(RPr') ∧ ∀Y: (RPr' contains Y ⇔ S contains Y ∨ Y = X)

(2) Print terminates.

Proof: (1) SPr0 = APr.'Sol' = S (Select)
and xPr = APr.'x' = X (Select)
So v0 = SPr0.'v' = S.'v' (Select)
and k1 = SPr0.'k' = S.'k' (Select)
So k2 = k1 + 1 = S.'k' + 1 (+1)
So v1 = append k2: x to v0
Now RPr' = Structure ('v': v1, 'k': k2)
So RPr'.'k' = k2 and RPr'.'v' = v1 (Structure)
And v1.k2 = X (Append)

We must prove SPValid(RPr'), for which we separately consider each clause in the definition of SPValid.

(1) S.'k' ≥ 0 (Assumption, since SPValid(S))
So k1 ≥ 0 (k1 = S.'k')
So k2 ≥ 0 (k2 = k1 + 1)

(2) ∀j: 1 ≤ j ≤ k1: v0 has j ∧ Correct(v0.j)
(SPValid(S), k1 = S.'k', v0 = S.'v')
So ∀j: 1 ≤ j < k2: v0 has j ∧ Correct(v0.j) (k2 = k1 + 1)
But v1 = append k2: X to v0
So ∀j: 1 ≤ j < k2: v1 has j ∧ Correct(v1.j)
(v1.i = v0.i when i ≠ k2 -- Append axiom)
But v1 has k2 ∧ v1.k2 = X (Append axiom)
and Correct (X) (Assumption)
So ∀j: 1 ≤ j ≤ k2: v1 has j ∧ Correct(v1.j)

(3) ∀i,j: 1 ≤ i,j ≤ k1: i ≠ j ⇒ v0.i ≠ v0.j (SPValid(S))
so ∀i,j: 1 ≤ i,j < k2: i ≠ j ⇒ v1.i ≠ v1.j (Append axiom, as above)
But v1.k2 = X (Append)
And ~ S contains X (Assumption)

i.e. $\forall i: 1 \le i \le k1: v0.i = X$ (Defn. 2)

i.e. $\forall i: 1 \le i < k2: v1.i \neq v1.k2$ (Append axiom, as above)

So $\forall i,j: 1 \le i,j \le k2: i \neq j \Rightarrow v1.i \neq v1.j$

So, since RPr'.'k' = k2 and RPr'.'v' = v1, we have shown SPValid(RPr')

We must finally show that

$\forall Y$(RPr' contains Y $\Leftrightarrow$ S contains Y $\vee$ Y = X)

Now RPr' contains Y $\Leftrightarrow \exists i: 1 \le i \le$ RPr'.'k': (RPr'.'v').i = Y

(defn. of contains)

i.e. RPr' contains Y $\Leftrightarrow \exists i: 1 \le i \le k2$: v1.i = Y

(RPr'.'k' = k2, RPr'.'v' = v1.)

So RPr' contains Y $\Leftrightarrow$ ($\exists i: 1 \le i < k2$: v1.i = Y) $\vee$ (v1.k2 = Y)

(property of $\exists$)

But v1.k2 = X (append)

And v1.i = v0.i when i $\neq$ k2 (append)

And k2 = k1 + 1

So RPr' contains Y $\Leftrightarrow$ ($\exists i: 1 \le i \le k1$: v0.i = Y) $\vee$ X = Y

Now k1 = S.'k' and v0 = S.'v' (Select)

So $\forall$Y (RPr' contains Y $\Leftrightarrow$ (S contains Y) $\vee$ (X = Y))

(2) Trivial, since the program for Print is an acyclic composition of primitive nodes.

Lemma 2. (As stated in the text, Lemma 2 is an immediate corollary of defns. 6, 5, and 4.)

Lemma 3. Let N, X, COL, UP, DOWN

= Config.'n', Config.'x', Config.'Col', Config.'Up', Config.'Down'

Then $0 \le h \le 7 \wedge$ PCConf(Config) $\wedge$ N < 8 $\vdash$

$\sim$ (COL has h) $\wedge \sim$(UP has h-N) $\wedge \sim$(DOWN has h+N)

$\Leftrightarrow$ PCorrect (append N:h to X)

Proof:

($\Rightarrow$) Let X' = append N:h to X

Let m = N + 1

We claim that m satisfies the existential quantifier in the definition of PCorrect(X') (Defn. 3). We check each clause of the quantitifed formula in turn.

(a) PCConf(Config) (assumption)

so PCorrect(X) (Defn. 5)

so $\exists n: \{0 \le n \le 8 \wedge (\forall i:$ X has $i \Leftrightarrow 0 \le i < n)\}$ (Defn. 3)

But $(\forall i:$ X has $i \Leftrightarrow 0 \le i < N)$ (Defn. 5)

so it is N that satisfies the existential quantifier in defn. 3 for PCorrect

So $0 \le N \le 8$

but $N < 8$ (assumption)

So $0 \le N < 8$

So $0 \le m \le 8$, as required $(m = N + 1)$

(b) X' = append N:h to X (defn)

So $\forall i:$ X' has $i \Leftrightarrow$ (X has $i \vee i = N$) (from append axiom)

But $\forall i:$ X has $i \Leftrightarrow 0 \le i < N$ (above)

So $\forall i:$ X' has $i \Leftrightarrow 0 \le i \le N$

So $\forall i:$ X' has $i \Leftrightarrow 0 \le i < m$, as required $(m = N + 1)$

(c) [As we warned earlier in the main text we shall normally omit the proofs of the existence of all the components we select from structures: they may easily be supplied by the reader if desired.]

X' = append N:h to X (defn)

So $\forall i: 0 \le i < N: X'.i = X.i$ (from append axiom)

But N satisfies the existential quantifier in the definition of PCorrect(X) (above)

So $(\forall i: 0 \le i < N: 0 \le X.i \le 7)$ (defn 3)

So $(\forall i: 0 \le i < N: 0 \le X'.i \le 7)$

But $X'.N = h$ (append axiom)

And $0 \le h \le 7$ (Assumption)

$(\forall i: 0 \le i \le N: 0 \le X'.i \le 7)$

So $(\forall i: 0 \le i < m: 0 \le X'.i \le 7)$ as required $(m = N + 1)$

Note that so far we have used none of the clauses on the left side of the $\Leftrightarrow$ in the statement of the lemma.

(d) N satisfies the quantifier in PCorrect(X) (above)

So $\forall i,j: 0 \le i,j < N:\ i \neq j \Rightarrow (X.i \neq X.j \wedge X.i+i \neq X.j+j \wedge X.i - i \neq X.j - j)$

But $\forall i: 0 \le i < N: X'.i = X.i$ (above)

So $\forall i.j: 0 \le i,j < N: i \neq j \Rightarrow (X'.i \neq X'.j \wedge X'.i + i \neq X'.j + j \wedge X'.i - i \neq X'.j - j)$

Now PCConf(Config) (assumption)

So $(\forall k: 0 \le k \le 7:$ COL has k $\Leftrightarrow (\exists i: 0 \le i < N: X.i = k))$ (defn 5)

But $0 \le h \le 7$ (assumption)

So COL has h $\Leftrightarrow (\exists i: 0 \le i < N: X.i = h)$

But $\sim$ (COL has h) (assumption for this half of proof)

So $\nexists i: 0 \le i < N:\ X.i = h$

So $\nexists i: 0 \le i < N: X'.i = h$ (since $\forall i: 0 \le i < N: X.i = X'.i$)

Similarly,

$\nexists i: 0 \le i < N: X'.i + i = h + N$ (since $\sim$(DOWN has h + N))

and

$\nexists i: 0 \le i < N: X'.i - i = h - N$ (since $\sim$(UP has h - N))

But $X'.N = h$ (above)

So $\forall i: 0 \le i < N: (X'.i \neq X'.N \wedge X'.i + i \neq X'.N + N \wedge X'.i - i \neq X'.N - N)$

And $\forall i,j: 0 \le i,j < N: i \neq j \Rightarrow (X'.i \neq X'.j \wedge X'.i + i \neq X'.j + j \wedge X'.i - i \neq X'.j - j)$

So, combining these last two formulae:

$\forall i,j: 0 \le i,j \le N: i \neq j \Rightarrow (X'.i \neq X'.j \wedge X'.i + i \neq X'.j + j \wedge X'.i - i \neq X'.j - j)$

So, since m = N + 1

$\forall i,j: 0 \le i,j < m: i \neq j \Rightarrow (X'.i \neq X'.j \wedge X'.i + i \neq X'.j + j \wedge X'.i - i \neq X'.j - j)$

as required.

This completes the verification that m satisfies the existential quantifier in the definition of PCorrect(X'), so PCorrect(X') is proved.

($\Leftarrow$) We assume that the formula on the left side of the $\Rightarrow$ in the statement of the lemma is false, and prove that the formula on the right is then also false.

So $\sim(\sim(\text{COL } \underline{\text{has}}\ h) \wedge \sim(\text{UP } \underline{\text{has}}\ h-n) \wedge \sim(\text{DOWN } \underline{\text{has}}\ h+N))$

So $(\text{COL } \underline{\text{has}}\ h) \vee (\text{UP } \underline{\text{has}}\ h-N) \vee (\text{DOWN } \underline{\text{has}}\ h+N)$

(a) Suppose COL has h

Now COL has h $\Leftrightarrow$ $(\exists i: 0 \le i < N: X.i = h)$ (proved above)

So $\exists i: 0 \le i < N: X.i = h$

So $\exists i: 0 \le i < N: X'.i = h$ (since $\forall i: 0 \le i < N: X.i = X'.i$)

But $X'.N = h$ (above)

So $\exists i: 0 \le i < N: X'.i = X'.N$

So $\exists i,j: 0 \le i,j \le N: i \ne j \wedge X.i = X.j$ (i.e. when $j = N$)

So $\sim(\forall i,j: 0 \le i,j < m: i \ne j \Rightarrow X'.i \ne X'.j)$

So m does not satisfy the existential quantifier in the definition of PCorrect(X') (defn 3)

But $\forall i$: X' has $i \Leftrightarrow 0 \le i < m$ (part $\Rightarrow$(b) above)

And for any n satisfying the quantifier in PCorrect(X')

$\forall i$: X' has $i \Leftrightarrow 0 \le i < n$ (defn 3)

So if m does not satisfy the quantifier no other value does.

So $\sim$ PCorrect(X')

(b) and (c). The same result may be proved assuming, respectively, (UP has h - N) and (DOWN has h + N).

So, since at least one of the assumptions (a) (b) and (c) must hold if

$$(\text{COL } \underline{\text{has}}\ h) \vee (\text{UP } \underline{\text{has}}\ h-N) \vee (\text{DOWN } \underline{\text{has}}\ h+N)$$

we have shown that in this case

$$\sim \text{PCorrect}(X')$$

Lemma 4. Let N, COL, UP, DOWN
= Config.'n', Config.'Col', Config.'Up', Config.'Down'
Then $0 \le h \le 7 \wedge$ PCConf(Config) $\wedge N < 8$
$\wedge \sim$(COL has h) $\wedge \sim$(UP has h - N) $\wedge \sim$(DOWN has h + N) $\vdash$
PCConf(AddNewQueen (Config, h))

Proof: Let X = Config.'x'
Let Config' = AddNewQueen (Config, h)
Let X', N', COL', UP', DOWN'
= Config'.'x', Config'.'n', Config'.'Col', Config'.'Up', Config'.'Down'
We are proving
PCConf(Config')
We proceed to check each of the clauses of the definition of PCConf (defn 3)

(a) Config' = Structure('n': N+1,
'x': append N:h to X
'Col': append h:nil to COL
'Up': append h - N: nil to UP
'Down': append h + N: nil to DOWN
(defn 7)

So N' = N + 1
X' = append N:h to X
COL' = append h:nil to COL
UP' = append h - N: nil to UP
DOWN': append h + N: nil to DOWN
(prop. of Structure)

Now X' = append N:h to X (above)
and $0 \le h \le 7 \wedge \sim$(UP has h - N) $\wedge \sim$(DOWN has h + N) (assumption)
So PCorrect(X') (Lemma 3)

(b) X' = append N:h to X
So $\forall i$: X' has i $\Leftrightarrow i = N \vee$ X has i (from append axiom)
But PCConf(Config) (assumption)
So $\forall i$: X has i $\Leftrightarrow 0 \le i < N$ (defn 5)
So $\forall i$: X' has i $\Leftrightarrow 0 \le i < N \vee i = N$
So $\forall i$: X' has i $\Leftrightarrow 0 \le i < N'$ (since N' = N + 1)

(c) PCConf(Config) (assumption)
So $(\forall k: 0 \le k \le 7:$ COL has k $\Leftrightarrow (\exists i: 0 \le i < N: X.i = k))$ (defn 5)

But X' = append N:h to X

So $\forall i: 0 \le i < N:$ X'.i = X.i

So $(\forall k: 0 \le k \le 7:$ COL has k $\Leftrightarrow (\exists i: 0 \le i < N:$ X'.i = k))

Now COL' = append h:nil to COL

So $\forall j: j \neq h:$ COL' has j $\Leftrightarrow$ COL has j (append axiom)

So $\forall k: 0 \le k \le 7 \wedge k \neq h:$ COL' has k $\Leftrightarrow (\exists i: 0 \le i < N:$ X'.i = k)

Now X'.N = h (append axiom)

So $\forall k: 0 \le k \le 7 \wedge k \neq h: (\exists i: 0 \le i < N:$ X'.i = k) $\Leftrightarrow$

$(\exists i: 0 \le i \le N:$ X'.i = k)

So $\forall k: 0 \le k \le 7 \wedge k \neq 7:$ COL' has k $\Leftrightarrow \exists i: 0 \le i \le N:$ X'.i = k

But COL' has h (since COL' = append h:nil to COL)

And $\exists i: 0 \le i \le N:$ X'.i = h (since X'.N = h)

So $\forall k: 0 \le k \le 7:$ COL' has k $\Leftrightarrow \exists i: 0 \le i \le N:$ X'.i = k

But N' = N + 1

So $\forall k: 0 \le k \le 7:$ COL' has k $\Leftrightarrow \exists i: 0 \le i < N':$ X'.i = k as required.

(d) (e) The proofs of the remaining two clauses are very similar and are left to the reader.

Lemma 5. PCConf(Config) $\vdash$

(AddNewQueen(Config, h)).'x' extends Config.'x'

Proof: Let Config' = AddNewQueen(Config, h)

Let X, X', N = Config.'x', Config'.'x', Config.'n'

Now PCConf(Config) (assumption)

So $\forall i:$ X has i $\Leftrightarrow 0 \le i < N$ (defn 5)

So $\forall i:$ X has i $\Rightarrow$ i $\neq$ N

But X' = append N:h to X

So $\forall i:$ X has i $\wedge$ i $\neq$ N $\Rightarrow$ X' has i $\wedge$ X'.i = X.i (append axiom)

So $\forall i:$ X has i $\Rightarrow$ X' has i $\wedge$ X'.i = X.i

i.e. X' extends X, as required. (defn 8)

Lemma 6. X extends Y $\wedge$ Correct(X)

$\wedge \{\exists n: 0 \le n \le 8 \wedge (\forall i:$ y has i $\Leftrightarrow 0 \le i < n)\}$

$\Rightarrow$ PCorrect(Y)

We verify each clause of the definition of PCorrect(Y) under the assumptions of the lemma.

(a) $\exists n: 0 \le n \le 8 \wedge (\forall i:$ Y has $i \Rightarrow 0 \le i \le n)$ (assumption)

This is the first part of the defn. of PCorrect(Y)

(b) Correct(X) (assumption)

So PCorrect(X) (defn 4)

So $\exists n: 0 \le n \le 8 \wedge (\forall i:$ X has $i \Rightarrow 0 \le i < n)$ (defn 3)

Let this n = N, and let the similar n in the statement of the lemma be renamed M.

i.e. $\forall i:$ Y has $i \Rightarrow 0 \le i < M$

and $\forall i:$ X has $i \Rightarrow 0 \le i < N$

But X extends Y (assumption)

So $\forall i:$ Y has $i \Rightarrow$ X has $i \wedge X.i = Y.i$ (defn 8)

So $M \le N$

Now $(\forall i: 0 \le i < N: 0 \le x.i \le 7)$

$\wedge (\forall i,j: 0 \le i,j < N: i \ne j \Rightarrow (X.i \ne X.j$

$\wedge X.i + i \ne X.j + j$

$\wedge X.i - i \ne X.j - j))$ (defn 3)

The quantifier structure of this is such that its validity is not impaired by reducing the size of the universe of quantification.

So $(\forall i: 0 \le i < M: 0 \le X.i \le 7)$

$\wedge (\forall i,j: 0 \le i,j < M: i \ne j \Rightarrow (X.i \ne X.j$

$\wedge X.i + i \ne X.j + j$

$\wedge X.i - i \ne X.j - j))$

But $\forall i: 0 \le i < M \Rightarrow$ Y has i (above)

And $\forall i:$ Y has $i \Rightarrow X.i = Y.i$ (defn 8)

So $(\forall i: 0 \le i < M: 0 \le Y.i \le 7)$

$\wedge (\forall i,j: 0 \le i,j < M: i \ne j \Rightarrow (Y.i \ne Y.j$

$\wedge Y.i + i \ne Y.j + j$

$\wedge Y.i - i \ne Y.j - j))$

This is the remainder of the definition of PCorrect(Y)

Lemma 7. Let S, C, P = Ag.'Sol', Ag.'Conf', Ag.'Proc'.

Then (1) SPValid(S) $\wedge$ PCConf(C) $\wedge$ ProcOK(P)

$\wedge (\forall X:$ Correct(X) $\wedge$ X extends C.'x' $\Rightarrow \sim$(S contains X))

$\Rightarrow$ SPValid(Rg') $\wedge (\forall Y:$ Rg' contains Y $\Leftrightarrow$ S contains Y

$\vee$ (Correct(Y) $\wedge$ Y extends C.'x'))

(2) Generate terminates, at least whenever PCConf(C).

Proof: (1) This part must be proved by induction, as Generate is a recursive procedure. We take the following as our inductive hypothesis.

Hypothesis: Let {1, 2} index the two apply actors in the program Generate.

Let p_i, A_i, R'_i be the corresponding links (procedure, argument and result, respectively) for apply actor i.

Let S_i, P_i, C_i = A_i.'Sol', A_i.'Proc', A_i.'Conf'.

Then

$$\forall i: 1 \le i \le 2: p_i \text{ refersto Generate} \Rightarrow$$
$$[\text{SPValid}(S_i) \wedge \text{PCConf}(C_i) \wedge \text{ProcOK}(P_i)$$
$$\wedge (\forall X: \text{Correct}(X) \wedge X \text{ extends } C_i.'x' \Rightarrow \sim(S_i \text{ contains } X))$$
$$\Rightarrow \text{SPValid}(R_i') \wedge (\forall Y: R_i' \text{ contains } Y \Rightarrow S_i \text{ contains } Y$$
$$\vee (\text{Correct}(Y) \wedge Y \text{ extends } C.'x'))]$$

The reader is referred to the diagrams of the Generate program for the binding of the various identifiers used in the proof.

Cf, Pr, Sn = Ag.'Conf', Ag.'Proc', Ag.'Sol' (Select)
= C, P, S (defn)

and Rg' = if b(C,P,S) then f(C,P,S) else g(C,P,S) (conditional)

Now b(C,P,S) = (C.'n' = 8) (program for b)

So there are two cases in the proof that Rg' has the desired properties.

Case 1: C.'n' = 8

In this case Rg' = zf' (see the program for f)

Now Al = append 'Sol':S to C

So Al.'Sol' = S (append axiom)

and SPValid(S) (assumption)

and PCConf(C) (assumption)

But C.'n' = 8 (this case)

So CConf(C) (defn 6)

So Correct(C.'x') (Lemma 2)

But Al.'x' = C.'x' (append axiom, since 'x' ≠ 'Sol')

Now $\forall X$: Correct(X) $\wedge$ X extends C.'x' $\Rightarrow \sim$(S contains X) (assumption)

But C.'x' extends C.'x' (extends is reflexive)

and Correct(C.'x') (above)

So ~(S contains C.'x')

And A1.'x' = C.'x'

Now let S_1, X_1 = A1.'Sol', A1.'x'

Then SPValid(S_1) ∧ Correct(X_1) ∧ ~(S_1 contains X_1)

Now ProcOK(P) (assumption)

So P.'pr' refers to Print (defn 9)

And P1 = P.'pr'

So SPValid(R1') ∧ ∀Y: (R1' contains Y ⇔ S_1 contains Y ∨ Y = X_1) (Lemma 1)

Now assume Y = X_1

Then Correct(Y) (since Correct(X_1))

and Y extends X_1 (extends is reflexive)

Conversely, assume Correct(Y) ∧ Y extends X_1

So Correct(Y)

So (∀i: Y has i ⇔ 0 ≤ i < 8) (defn 4)

But Correct(X_1) (assumption)

So (∀i: X_1 has i ⇔ 0 ≤ i < 8) (defn 4)

But Y extends X_1

So ∀i: X_1 has i ⇒ Y has i ∧ X_1.i = Y.i (defn 8)

So ∀i: X_1 has i ⇔ Y has i

and ∀i: X_1 has i: X_1.i = Y.i

i.e. Y = X_1 (equality of structures)

So Y = X_1 ⇔ Correct(Y) ∧ Y extends X_1

So, substituting in the result of applying Lemma 1 above,

SPValid(R1') ∧ (∀Y: R1' contains Y ⇔ S_1 contains Y ∨ (Correct(Y) ∧ Y extends X_1))

as required. (since X_1 = C.'x')

Case 2: C.'n' ≠ 8

PCConf(C) (assumption)

So 0 ≤ C.'n' ≤ 8 (by the same argument as at the start of the proof of Lemma 3)

But C.'n' ≠ 8 (this case)

So 0 ≤ C.'n' < 8

In this case Rg' = Zg'. (See the program for g.) Zg' is part of the output of an iteration, and we prove its properties in the usual way.

We take the following predicate as our invariant for the iteration:

P(h, C, P. S) ≡

$0 \le h \le 8 \wedge C = Cf \wedge P = Pr \wedge$ SPValid(S)

$\wedge$ [$\forall$Y: S contains Y $\Leftrightarrow$ (Sn contains Y

$\vee$ (Correct(Y) $\wedge$ Y extends C.'x' $\wedge$ Y.N < h)]

where N = Cf.'n'

We must first show that P holds for the input values to the iteration: i.e. that $P(h_o, Cf, Pr, Sn)$ holds.

(a)	$h_o = 0$, so $0 \le h_o \le 8$	
(b)(c)	Cf = Cf, and Pr = Pr	
(d)	SPValid(Sn)	(assumption for Generate)
(e)	Suppose for some Y, Correct(Y)	
Then	PCorrect(Y)	(defn 4)
And	($\forall$i: Y has i $\Leftrightarrow$ $0 \le i < 8$)	(defn 4)
So	$\forall i: 0 \le i < 8: 0 \le Y.i \le 7$	(defn 3, putting n = 8)
In particular	$0 \le Y.N$	(Since $0 \le N < 8$, this case)
So, since $h_o = 0$,		(above)
	Correct(Y) $\Rightarrow$ $Y.N \not< h_o$	
i.e. $\nexists$Y:	Correct(Y) $\wedge$ $Y.N < h_o \wedge$ Y extends Cf.'x'	
So	($\forall$Y: Sn contains Y $\Leftrightarrow$ (Sn contains Y	
	$\vee$ (Correct(Y) $\wedge$ Y extends Cf.'x' $\wedge$ $Y.N < h_o$)))	
So	$P(h_o, Cf, Pr, Sn)$ holds.	

We must now show that P is preserved by the body g1, whenever b1(h, C, P, S) is true. That is to say, we must prove for the program g1:

$P(h, C, P, S) \wedge b1(h, C, P, S) \vdash P(h', C', P', S')$

We prove each clause of the required result in turn.

(a)	b1(h, C, P, S)	(assumption)
So	$h < 8$	(program for b1)
And	P(h, C, P, S)	(assumption)
So	$0 \le h \le 8$	(defn of P)
So	$0 \le h < 8$	
But	$h' = h + 1$	(+1 actor)
So	$0 \le h' \le 8$ as required.	

(b) C' = C (program for g')
But C = Cf (assumption, since P(h, C, P, S))
So C' = Cf.

(c) Similarly, P' = Pr.

(d) S' = if b2(h, C, P, S) then f2(h, C, P, S) else g2(h, C, P, S) (Conditional)
So, considering the program for b2, we let
N, COL, UP, DOWN = C.'n', C.'Col', C.'Up', C.'Down'
So Cn, Cc, Cu, Cd = N, COL, UP, DOWN (Axioms for select)
And b3 = COL has h (Axiom for exists)
Similarly b4, b5 = UP has h - N, DOWN has h + N
And Zb' = b3 ∨ b4 ∨ b5 (Axiom for ∨)
= (COL has h) ∨ (UP has h-N) ∨ (DOWN has h + N)
= ~(~(COL has h) ∧ ~(UP has h - N) ∧ ~(DOWN has h + N))

For S' we must consider two cases.

Case 1. Zb' is true.
Then S' = f2(h, C, P, S)
= S (program for f2)
But SPValid(S)
So SPValid(S') as required

Case 2. Zb' is false
i.e. ~(COL has h) ∧ ~(UP has h - N) ∧ ~(DOWN has h + N)
Then S' = g2(h, C, P, S)
= S1' (program for g2)
It may readily be verified (the task is left to the reader) that
c2 = AddNewQueen (c, h)
Now 0 ≤ h < 8 (above)
i.e. 0 ≤ h ≤ 7
And PCConf(C) (assumption for Generate)
And N < 8 (above: this case of outer conditional)
And ~(COL has h) ∧ ~(UP has h - N) ∧ ~(DOWN has h + N) (this case of inner conditional)
So PCConf(c2) (Lemma 4)
Now let X2 = c2.'x'
And X1 = Cf.'x'

Now X2 extends C.'x' (Lemma 5)

and C = Cf

So X2 extends X1

So ∀X: X extends X2 ⇒ X extends X1 (extends is transitive)

So ∀X: Correct(X) ∧ X extends X2 ⇒ Correct(X) ∧ X extends X1

But ∀X: Correct(X) ∧ X extends X1 ⇒ ~(Sn contains X)

(assumption for this lemma)

and ∀X: S contains X ⇔ Sn contains X

∨ (Correct(X) ∧ X extends X1 ∧ X.N < h) (since by assumption P(h, C, P, S))

So, since

∀X: Correct(X) ∧ X extends X2 ⇒ ~(Sn contains X)

we have

∀X: Correct(X) ∧ X extends X2 ⇒ (S contains X ⇔ X.N < h)

But if X extends X2,

∀i: X2 has i ⇒ X has i ∧ X.i = X2.i (defn 8)

But X2.N = h (by defn 7)

So ∀X: X extends X2 ⇒ X.N = h

So ∀X: Correct(X) ∧ X extends X2 ⇒ ~(S contains X)

Now p2 = P.'gen' (axiom for select)

And ProcOK(Pr) (assumption for Generate)

And P = Pr (assumption: P(h, C, P, S))

So p2 refers to Generate (defn 9)

And A2 = Structure('Sol':S, 'Proc':P, 'Conf':C2)

So S, P, C2 = A2.'Sol', A2.'Proc', A2.'Conf'. (Structure)

So since SPValid(S) ∧ PCConf(C2) ∧ ProcOK(P)

∧ (∀X: Correct(X) ∧ X extends C2'x' ⇒ ~(S contains X))

we have, by the inductive hypothesis:

SPValid(R2') ∧ (∀Y: R2' contains Y ⇔ S contains Y

∨ (Correct(Y) ∧ Y extends C2.'x'))

In particular, since S' = R2'

SPValid(S') as required.

(e) We must prove

∀Y: S contains Y ⇔ (Sn contains Y

∨ (Correct(Y) ∧ Y extends C'.'x' ∧ Y.N' < h'))

where N = C'.'n'

We know h' = h + 1 and C' = C = Cf.

So, letting N = Cf.'n' as before, we are proving

$$\forall Y\text{: S } \underline{\text{contains}}\ Y \Leftrightarrow (Sn\ \underline{\text{contains}}\ Y \vee (\text{Correct}(Y) \wedge Y \text{ extends } C.'x' \wedge Y.N \leq h))$$

i.e.

$$\forall Y\text{: S } \underline{\text{contains}}\ Y \Leftrightarrow (Sn\ \underline{\text{contains}}\ Y \vee (\text{Correct}(Y) \wedge Y\ \underline{\text{extends}}\ C.'x' \wedge Y.N < h) \vee (\text{Correct}(Y) \wedge Y\ \underline{\text{extends}}\ C.'x' \wedge Y.N = h)$$

Again, as in part (d), we must separately consider two cases according as zb' is true or false.

Case 1. zb' is true

i.e. (using the same notation as before):

$$\sim(\sim(COL\ \underline{\text{has}}\ h) \wedge \sim(UP\ \underline{\text{has}}\ h - N) \wedge \sim(DOWN\ \underline{\text{has}}\ h + N))$$

Let $X1 = Cf.'x'$

Now consider $H = AddNewQueen(C, h)$

Let $Xh = H.'x'$

Then $Xh = \underline{\text{append}}\ N{:}h\ \underline{\text{to}}\ X1$ (defn 7)

So $\sim PCorrect(Xh)$ (Lemma 3)

Now $PCConf(C)$ (assumption)

So $(\forall i: X1\ \underline{\text{has}}\ i \Leftrightarrow 0 \leq i < N)$ (defn 5)

So, by the axion for append:

$$(\forall i: Xh\ \underline{\text{has}}\ i \Leftrightarrow (0 \leq i < N \vee i = N))$$

And $0 \leq N < 8$ (for this case of outer conditional)

So if $M = N + 1$

$$0 \leq M \leq 8 \wedge (\forall i: Xh\ \underline{\text{has}}\ i \Leftrightarrow 0 \leq i < M)$$

But since, by Lemma 6

$$X\ \underline{\text{extends}}\ Xh \wedge \text{Correct}(X) \wedge \{\exists m: 0 \leq m \leq 8 \wedge (\forall i: Xh\ \underline{\text{has}}\ i \Leftrightarrow 0 \leq i < m)\} \Rightarrow PCorrect(Xh)$$

and we know $\sim PCorrect(Xh)$ (above)

We have $\nexists X: X\ \underline{\text{extends}}\ Xh \wedge \text{Correct}(X)$

Now X extends $Xh \equiv$

$$\forall i: Xh\ \underline{\text{has}}\ i \Rightarrow X\ \underline{\text{has}}\ i \wedge X.i = Xh.i$$ (defn 8)

But $Xh = \underline{\text{append}}\ N{:}h\ \underline{\text{to}}\ X1$ (above)

So $\forall i: Xh\ \underline{\text{has}}\ i \Leftrightarrow i = N \vee X1\ \underline{\text{has}}\ i$ (append axiom)

So $X\ \underline{\text{extends}}\ Xh$

$$\Leftrightarrow X\ \underline{\text{has}}\ N \wedge X.N = h \wedge \{\forall i: X1\ \underline{\text{has}}\ i \Rightarrow X\ \underline{\text{has}}\ i \wedge X.i = X1.i\}$$

i.e. X extends Xh ⇔ X extends X1 ∧ X has N ∧ X.N = h

So ∄X: X extends X1 ∧ Correct(X) ∧ X.N = h

So Sn contains Y ∨ (Correct(Y) ∧ Y extends X1 ∧ Y.N ≤ h)

⇔ Sn contains Y ∨ (Correct(Y) ∧ Y extends X1 ∧ Y.N < h)

But ∀Y: S contains Y ⇔ Sn contains Y

∨ (Correct(Y) ∧ Y extends X1 ∧ X.N < h)

(since P(h, C, P, S))

So, since S' = S for this case

∀Y: S' contains Y ⇔ Sn contains Y

∨ (Correct(Y) ∧ Y extends X1 ∧ X.N ≤ h) as required.

Case 2. zb' is false

For this case we have already proved (case (d) above)

∀Y: S' contains Y ⇔ S contains Y

∨ (Correct(Y) ∧ Y extends C2.'x')

Let C2.'x' = X2

Then X2 = append N:h to X1 (above)

(where X1 = C.'x', as before)

So, by the argument used in case 1

Y extends X2 ⇔ Y extends X1 ∧ Y.N = h

So ∀Y: S' contains Y ⇔ S contains Y ∨ (Correct(Y) ∧ Y extends X1 ∧ Y.N = h)

But, since we are assuming P(h, C, P, S),

∀Y: S contains Y ⇔ Sn contains Y ∨ (Correct(Y) ∧ Y extends X1 ∧ Y.N < h)

So ∀Y: S' contains Y ⇔ Sn contains Y

∨ (Correct(Y) ∧ Y extends X1 ∧ Y.N ≤ h) as required

So we have proved

P(h0, Cf, Pr, Sn)

and P(h, C, P, S) ∧ bl(h, C, P, S) P(h', C', P', S')

So by the iteration induction rule, we have

P(h0', Cf', Pr', Sn') ∧ ~bl (h0', Cf', Pr', Sn')

i.e.

0 ≤ h0' ≤ 8 ∧ Cf' = Cf ∧ Pr' = Pr ∧ SPValid(Sn')

∧ [∀Y: Sn' contains Y ⇔ Sn contains Y

∨ (Correct(Y) ∧ Y extends Cf'.'x' ∧ Y.N < h0')]

∧ h0' ≥ 8

where N = Cf.'n'.

So h0' = 8

Now if Correct(Y)

we have, from definitions 4 and 3:

$$\forall i: 0 \le i < 8: 0 \le Y.i \le 7$$

And we know $0 \le N < 8$ (this case of outer conditional)

So Correct(Y) $\Rightarrow$ Y.N < 8

So Correct(Y) $\wedge$ Y.N < h0' $\Leftrightarrow$ Correct(Y)

So, since zg' = Sn', we have in particular:

SPValid(zg') $\wedge$ [$\forall$Y: zg' contains Y

$\Leftrightarrow$ (Sn contains Y $\vee$ (Correct(Y) $\wedge$ Y extends Cf.'x'))]

This concludes the proof of Lemma 7 part 1.

Proof of Part 2: This is straightforward: for the recursion we can order the set of possible arguments for Generate according to:

$$8 - (Ag.'Conf').'n'$$

Since PCorrect(Ag.'Conf') is an assumption, we have

$$0 \le (Ag.'Conf').'n' \le 8.$$

So the ordering expression is always non-negative, and the set is well-founded. Moreover, for the only recursive call, it has already been verified that

$$(A2.'Conf').'n' = (Ag.'Conf').'n' + 1$$

So the argument is less than the original argument in the ordering, as required for termination.

Similarly, for the iteration, we order the set of possible input and output values according to

$$8 - h$$

Since $0 \le h \le 8$, this also is always non-negative, and the set is, therefore, well-founded. And, since

$$h' = h + 1$$

the input tuple is mapped into a lesser one by the body, as required for termination.

Theorem (1) A = nil $\vdash$

SPValid(R') $\wedge$ ($\forall$Y: R' contains Y $\Leftrightarrow$ Correct(Y))

(2) The main program terminates.

Proof: (1) (See the main program diagram for the meaning of the identifiers used.)

We wish to show that A0 satisfies all the assumptions of the Generate theorem. Now,

A0 = Structure('Conf': C0, 'Sol': S0, 'Proc': P0)

So C0, S0, P0 = A0.'Conf', A0.'Sol', A0.'Proc'

(a) S0 = Structure('k':0, 'V':nil)

So S0.'k' = 0

So S0.'k' ≥ 0

And (∀j: 1 ≤ j ≤ S0.'k': χ)

where χ is any formula, as {j|1 ≤ j ≤ 0} is empty.

So, in particular,

SPValid(S0) as required.

(b) C0 = Structure('n':0, 'x':nil, 'Col':nil,
'Up':nil, 'Down':nil)

We must show PCConf(C0)

(i) We claim that 0 satisfies the existential quantifier in PCorrect(nil) (defn. 3)

(1) 0 ≤ 0 ≤ 8

(2) ∄i: nil has i (nil axiom)

and ∄i: 0 ≤ i < 0

So ∀i: (nil has i ⇔ 0 ≤ i < 0)

(3)-(4) The remaining clauses are trivially true, as the universes of quantification are empty.

So PCorrect(C0.'x')

(ii) C0.'n' = 0

So ∄i: 0 ≤ i < C0.'n'

But ∄i: (C0.'x') has i ((C0.'x') = nil)

So ∀i: (C0.'x') has i ⇔ 0 ≤ i < C0.'n'

(iii) C0.'Col' = nil

So ∀k: ~((C0.'Col') has k) (nil axiom)

Also ∄i: 0 ≤ i < C0.'n' (since C0.'n' = 0)

So ∀k: 0 ≤ k ≤ 7: (C0.'Col') has k ⇔ ∃i: 0 ≤ i < C0.'n':
((C0.'x').i) = k

(iv)(v) Similarly for the other two clauses.

So PCConf(C0) as required.

(c) We can easily see that P0.'pr' refers to Print and P0.'gen' = g0, and g0 refers to Generate

so ProcOK(P0) as required.

(d) S0.'k' = 0 (above)

So $\nexists i: 1 \leq i < S0.'k'$

So $\forall X: \sim(S0 \text{ contains } X)$ (defn. 2)

So $\forall X: \text{Correct}(X) \wedge X \text{ extends } C0.'x' \Rightarrow \sim(S0 \text{ contains } X)$

as required.

So A0 satisfies all the assumptions of Lemma 7, and g0 refers to Generate

So, by Lemma 7

$$\text{SPValid}(R') \wedge (\forall Y: R' \text{ contains } Y \Leftrightarrow S0 \text{ contains } Y \vee (\text{Correct}(Y) \wedge Y \text{ extends } C0.'x'))$$

Now $\forall Y: \sim(S0 \text{ contains } Y)$ (above)

Also, since $\forall i: \sim((C0.'x') \text{ has } i)$ (since C0.'x' = nil)

$\forall Y: Y \text{ extends } C0.'x'$ (defn. 8)

So

$$\text{SPValid}(R') \wedge (\forall Y: R' \text{ contains } Y \Leftrightarrow \text{Correct}(Y))$$ as required

Part (2): The proof of termination is trivial, since we have proved that Generate terminates, and the main program is therefore an acyclic composition of actors each of which terminates.

References

[1] R. W. Floyd. Assigning meanings to programs. Proc. Amer. Math. Society Symposium in Applied Mathematics 19 (1967),19-31.

[2] C. A. R. Hoare. An axiomatic basis for computer programming. Comm. of the ACM 12 (1969), 576-580, 583.

[3] J. B. Dennis. First Version of a Data Flow Procedure Language. Computation Structures Group Memo 93, Project MAC, M.I.T., 1973.

[4] J. B. Dennis and J. B. Fosseen. Introduction to Data Flow Schemas. Computation Structures Group Memo 81, Project MAC, M.I.T., 1973.

[5] G. Kahn. A Preliminary Theory for Parallel Programs. Internal Memo, IRIA (Laboria), 1973.

[6] E. W. Dijkstra. Notes on structured programming. Section 1 of Structured Programming by O.-J. Dahl, E. W. Dijkstra and C. A. Hoare, Academic Press, London and New York, 1972.