THE EQUIVALENCE OF MODELS OF TASKING[+]

by Daniel M. Berry
Brown University

Abstract. A technique for proving the equivalence of implementations of multi-tasking programming languages is developed and applied to proving the equivalence of the contour model and a multi-tasking version of the copy rule.

0. Introduction

In multi-tasking programming languages, e.g. Algol 68 [vWN 69] and PL/1 [Wlk 69], the inherent nondeterminism associated with tasking accentuates the difficulty of precisely describing computations of programs in the language and of proving the correctness (i.e., equivalence to definition) of implementations of the language.

The purposes of this paper are 1) to illustrate a method of defining (implementations of) multi-tasking languages and 2) to show the development of a proof technique which is based on the definition method and which is appropriate for proving equivalence of implementations of multi-tasking languages*.

As the definition method we make use of nondeterministic information structure models (NDISMs) [Weg 70,71a,b,72]. An NDISM treats a computation as a sequence of successive snapshots; because of the nondetermism, there may be a choice of several possible successor snapshots for a given snapshot. One defines a language and/or an implementation by giving an NDISM which models the computations in the language and/or implementation.

To illustrate the proof technique, we define and prove equivalent two NDISMs which model implementations of a fictitious generalized block structured language GBSL. One of these models is an extension of the copy rule which has been used to define several block structured languages, e.g. PL/1 [Wlk 69], Algol 60 [BG 70, Nau 63] and Algol 68 [vWn 69], and the other is Johnston's contour model [Joh 71] which is closer to an actual implementation.**

Because of the sheer size of the definition of a complete language, a complete implementation, and thus a complete proof of equivalence, we find it necessary to find a suitable level of semi- to informal definition and proof. While the definitions and proofs given as examples may of necessity be less than completely formal, it is necessary that the basis on which the definitions and proofs are made be firmly rooted in formal results. Therefore, we shall formally develop the notions of nondetermistic information structure model, computations, and equivalence of NDISMs, and we shall prove the validity of the proof techniques we use.

In an attempt to be somewhat coherent, somewhat narrative, and somewhat precise, the plan for the sections of the paper is as follows:

+This work was supported in part by NSF Grant GJ-28074 and by General Electric R & DC, Schenectady, N.Y.

*Similar work has been done by the IBM Vienna Group with their definition of the multi-tasking language PL/1 [Wlk 69] and in their development and application of the Lucas Twin Machine proof technique [Luc 68, HJ 70, JL 71].

**Thus, this paper extends the result of a previous paper by this author [Bry 71c] which shows that the single task contour model is equivalent to the single task copy rule.

1. Define NDISMs in general
2. Give assumption for NDISMs to be used for defining programming languages
3. Informally describe syntax of GSBL
4. Give and describe example program in GSBL to be used to refer to in sections 5 and 6
5. Informally define contour model and give sequence of snapshots of example program
6. Informally define copy rule and give sequence of snapshots of example program
7. Define various forms of equivalence for NDISMs for programming languages and show validity of proof techniques
8. Informally prove contour model equivalent to copy rule.

Due to page limitations, brevity was required. Since the contour model and the copy rule have been written about elsewhere, brevity was obtained at the expense of the definition of the models in sections 5 and 6. The reader who finds these sections too brief is urged to consult the following more leisurely, narrative and pedagogic introductions (beware: there are a few minor changes in the present versions of these models):

contour model: Joh 71, Bry 71a,b
copy rule: Weg 71b, Bry 71c, Weg 72

1. Nondeterministic information structure models

We will define a programming language by giving a nondetermistic information structure model (NDISM) for it. A computation of a program in the language will be described as a sequence of snapshots (instantaneous descriptions, core dumps) taken between successive instruction executions. Each computation starts off with an initial snapshot S_0 and proceeds through successive snapshots $S_1,S_2,\ldots$. Each snapshot is obtained from the previous by execution of some instruction by some task, that is, by the application of some transformation. Since it is nondeterministic as to which task executes next, the transformations are nondeterministic; that is, they map a snapshot to a set of snapshots. Although we make use of such programming language notions as programs and tasks in intuitive discussions, for the sake of generality we would like the definitions of NDISMs and related concepts to be as independent as possible of these specifics of programming languages.

Therefore, we define an NDISM as a threetuple, (I,I_0,F), where I is a countable set of all possible snapshots, I_0 is the subset of I which is the set of all possible initial snapshots, and F is a finite set of transformations which map a snapshot to a set of snapshots.

Definition 1. $M = (I,I_0,F)$ is a nondeterministic information structure model (NDISM) if and only if

1) I is a countable set of objects called snapshots.
2) $I_0 \subseteq I$ is the set of objects called initial snapshots.
3) F is a finite set of transformations each of the form $f:I\to\mathcal{P}(I)$, where $\mathcal{P}(I)$ denotes the set of all subsets of I∎

A transformation is applicable to a given snapshot if the transformation maps the snapshot to a non-null set of snapshots. A snapshot is transformable if there is a transformation applicable to it; otherwise it is intransformable.

Definition 2. Let $M=(I,I_0,F)$ be an NDISM; let $S\varepsilon I$ be a snapshot. Then,

1) If $f\varepsilon F$ is a transformation, then f is applicable to S if and only if $f(S)\neq\emptyset$.

2) S is transformable if and only if for some $f\varepsilon F$, f is applicable to S.

3) S is intransformable if and only if S is not transformable∎

A computation is a sequence of snapshots satisfying certain initial and inductive conditions; i.e., if the sequence is non-empty, then the first snapshot in the sequence, S_0, is an element of I_0; and, for all S_i in the sequence, $S_i\varepsilon f(S_{i-1})$ for some $f\varepsilon F$. However, this is not enough. Suppose $<S_0,S_1,S_2,S_3>$ is a computation. Then clearly the sequences $<S_0>$, $<S_0,S_1>$ and $<S_0,S_1,S_2>$ all satisfy the initial and inductive conditions and thus appear to be computations even though they are all "incomplete" subsequences of a computation. To fix this hole in the definition we add the stipulation that a computation is a sequence which is also not a proper initial subsequence of any other sequence satisfying the initial and inductive conditions.

Definition 3. Let $M=(I,I_0,F)$ be an NDISM. Then the sequence $C=<S_0,S_1,\ldots,S_i,\ldots>$ is a computation in M if and only if

1) for all S_i in C, $S_i\varepsilon I$

2) if $C\neq<>$ (the empty sequence), then $S_0\varepsilon I_0$

3) for all S_i in C with $i\geq 0$, there exists an $f\varepsilon F$ such that $S_i\varepsilon f(S_{i-1})$

4) for all sequences D satisfying (1), (2) and (3) above, C is not a proper initial subsequence of D∎

We say that C is a computation of S_0 in M if the first snapshot of C is S_0. Also, for an NDISM M, we define the function M to give all computations in M of a given S_0.

Definition 4. Let $M=(I,I_0,F)$ be an NDISM; let $S_0\varepsilon I_0$. Then

1) C is a computation of S_0 in M if and only if
   a) $C\neq <>$ is a computation in M.
   b) The first snapshot in C is S_0.

2) $M(S_0)=\{C|C$ is a computation of S_0 in $M\}$∎

Since every snapshot of a computation C except the first is obtained by transforming the previous, it is clear that there is at most one intransformable snapshot in C (there may be none, if C does not halt) and if there is an intransformable snapshot in C, it must be the last one.

THEOREM 1. Let $M=(I,I_0,F)$ be an NDISM; let C be a computation in M. Then

1) There exists at most one snapshot $S\varepsilon I$ in C such that S is intransformable.

2) If $S\varepsilon I$ is a snapshot in C such that S is intransformable, then S is the last snapshot in C∎

We now have the right to speak of the unique intransformable snapshot S of a computation if such a snapshot exists. We call a computation that has one a halting computation and we call the intransformable snapshot the final snapshot of the computation.

Definition 5. Let $M=(I,I_0,F)$ be an NDISM; let C be a computation in M. Then

1) C halts if and only if for some $S\varepsilon I$, S is in C and S is intransformable,

2) final(C) is defined if and only if C halts,

3) final(C)=S if and only if S is in C and S is intransformable.

2. NDISMS for programming languages

So far NDISMs and computations have been defined independently of programming languages. Since the NDISMs given in what follows are for modelling executions of programs in some programming language, we must add assumptions under which an NDISM is an NDISM for a programming language L. In most programming languages we have input and output routines. Therefore, we assume that for each NDISM, M, for executing programs of L there are snapshot component selection functions called input_M and output_M, which select the intput and output lists of a snapshot. We also assume that there is a set of input lists, INPUT, and a set of output lists, OUTPUT, in which all possible input lists and output lists of integers, reals, booleans and character strings may be found.

Furthermore, we assume that there exists a compile function, compile_M, which translates a given program p in L and a given input δ in INPUT to the initial snapshot of M for a computation of the program p with input δ. The input_M and compile_M functions are assumed to be related in the following manner: the input_M of an initial snapshot S_0 is δ if and only if for some program $p\varepsilon L$, S_0 is the result of compiling p and δ.

Assumption 1. Let $M=(I,I_0,F)$ be an NDISM for a programming language L. Then

1) there exist countable sets INPUT and OUTPUT of lists of integers, reals, booleans and character strings. The empty list, <>, is in both INPUT and OUTPUT.

2) there exist functions, $\text{input}_M:I\rightarrow\text{INPUT}$ and $\text{output}_M:I\rightarrow\text{OUTPUT}$.

3) there exists a function, $\text{compile}_M:(L\times\text{INPUT})\rightarrow I_0$.

4) for all $S_0\varepsilon I_0$, $\text{input}_M(S_0)=\delta$ if and only if for some $p\varepsilon L$, $\text{compile}_M(p,\delta)=S_0$∎

Therefore, the set of computations in M of the program p in L with input δ is denoted by $M(\text{compile}_M(p,\delta))$.

Since we are concerned only with NDISMs for programming languages, we shall refer to them simply as NDISMs in the discussions that follow.

3. Syntax of programs in GBSL

We give a brief description of the syntax of programs in a generalized block structured language GBSL, which is a generalization of Algol 60 and which incorporates most of the common features of most block structured languages. GBSL also has facilities for task creation, termination and synchronization. While these tasking features are not entirely standard, they do indicate what can be done.

A program in GBSL is a single block*. A block has a declaration list and a statement list. A statement, which is optionally labelled, is either an assignment statement, a conditional statement, a procedure call, a goto statement, a (nested) block, a task creation statement, a block, resume or terminate statement, or a request or release statement. Some of the standard procedures include input/output procedures.

A procedure declaration is similar to a block. It has a parameter specification list and a body which consists of a single statement (which, of course, may be a block). We observe that blocks and procedures may be arbitrarily nested within each other.

All identifiers used in a program must be declared in some block or be specified as a parameter

*This assumption can be made at no loss of generality. To get external procedures, consider the outer block to be a prelude block in which all external procedures are declared and in which the main block is immediately nested.

of some procedure. We shall use the term "declaration" to mean both declaration and parameter specification. A declaration of an identifier is said to have a scope which includes all statements, blocks and procedures nested inside the declaring block or procedure except for those blocks and procedures in which the identifier is redeclared. The use of an identifier is said to identify the declaration in whose scope the use lies; every legal use of an identifier must identify some declaration.

When an identifier is declared, it is declared to be a variable of a certain data type. These types include integer, real, boolean, character, pointer, label, procedure, task and semaphore, as well as arrays* and structures* consisting of elements of any data type.

An assignment statement has a left-hand side, which may be a simple variable of any data type, a subscripted variable, or a pointer identifier with indirection specified, and a right-hand side which may be a variable of any data type or some arithmetic, boolean, character or pointer expression.

There are two forms that a task creation statement may take:

```
goto ℓ task
p(arg) task
```

The first specifies that a task is to be created so that it begins executing at the statement labelled ℓ. The second specifies that a task be created so that it begins executing in the procedure body of p. Parameters may be passed to the procedure. When the task finishes executing the body, it terminates itself.

Optionally, one may specify that the created task is to be named by a task variable t, as in

```
goto ℓ task t
```

or

```
p(arg) task t
```

A named task may be blocked, resumed (unblocked), or terminated by using its task name in the appropriate statement:

```
block t
resume t
terminate t
```

If a task executes block or terminate, it blocks or

(continued on top right column)

terminates itself.

Task synchronization is accomplished by allowing a task to request the private, uninterrupted use of a semaphore and to release the semaphore when it is done. The operations of request(S) and release(S) are similar to those of Dijkstra's P(S) and V(S) and ESOPE's request(s) and release(s) [Dij 68, Bet 70].

4. Example program

As an example of GBSL syntax we have given a GBSL program which has been very purposely contrived to illustrate most of the features we will be discussing in coming sections. The program,

```
 1   begin integer a; task t;
 2     procedure p(i);
 3       integer i; value i;
 4       a:=i+1
 5     ;
 6     begin integer a;
 7       p(5);
 8       if random=.5 then goto ℓ
 9     end;
10   ℓ:p(5) task t
11   end
```

consists of an outer block declaring integer a, task t, label ℓ (implicitly), and procedure p. It has an inner block which declares another integer a. The procedure p has an integer by-value parameter i and a use of the non-local a declared in the outer block. The program can be executed in the following sequence: the outer block and then the inner block are entered. In line 7, p(5) is called. Then, depending on the value of the random number generator, either we goto ℓ, performing a so-called non-local jump, or else we exit the block through the end. In either case, in line 10, a new task which executes the body of p is created and named t. In the meanwhile the main task goes on to exit the outer block and terminate itself. The created task finishes executing the body of p and then it terminates. With both tasks terminated, the computation halts. This execution sequence is the basis for the sequences of snapshots in sections 5 and 6.

In sections 5 and 6 below, the contour model, CM, and the copy rule, CR, are defined as NDISMs. For each we give the format of the snapshots, the format of the initial snapshot and the compile function, the transformation, and a complete sequence of snapshots for a computation of the example program. Occasionally the descriptions of the models will refer to "before" and "after" snapshots in these computations.

Sections 5 and 6 are given side-by-side in opposite columns of the page so that corresponding parts of the definition and corresponding snapshots are opposite each other. Descriptive text which is common to both definitions runs across both columns (such as this paragraph). This side-by-side treatment should accentuate the equivalence of the two models and should set the reader in the proper frame of mind for the forthcoming proof technique.

5. Contour model (CM)

Snapshots. Each snapshot in CM consists of three components (see snapshots 0 and 1 for examples).

1) The algorithm is a GBSL program with the lines numbered for the purposes of addressing (pointing to) these lines. We require that the algorithm be time-invariant and re-entrant. (Snapshot 0 has an algorithm; since the algorithm is invariant, it is not shown in the rest of the snapshots.)
2) The record of execution contains all cells which are allocated during the execution of the algorithm. These include:

*We shall not be concerned with these types.

6. Copy rule (CR)

Snapshots. Each snapshot in CR consists of five components (see snapshots 0 and 1 for examples).

1) A possibly null set of task stacks, each of which is a task. (This is in the top half of the snapshots.) Each task has four components:
   a) the stack-of-texts is a pushdown stack of modified versions of the GBSL program being executed. The bottom of the stack** contains an unmodified version of this program. In each of these texts the lines are numbered for the purpose of addressing these lines. The numbering is such that modified versions

**When we use the word "stack" we refer to "stack-of-texts" rather than "task stack".

a) a possibly empty set of contours. (These are the rectangles of the snapshots.) Each contour contains all (sub)cells* allocated as the result of one block or procedure entry, in particular:
   1) a declaration array contains one (sub)cell* for each identifier explicitly or implicitly declared in the entered block (in the upper left-hand corner of a contour);
   2) a static link which for a contour c points to a contour d allocated for the block d' in which the block c' corresponding to contour c is nested (the static link is in the lower left-hand corner of the contour).

b) a possibly empty set of processors, each of which is a task (these are the π's in the snapshots). The components of a processor are three:
   1) an instruction pointer, ip, which is a number pointing to the next statement in the algorithm to be executed. (The ip is the arrow coming out of the right-hand side of the π; an ip pointing to line n is denoted by "→n".
   2) an environment pointer, ep, which points to a contour allocated for the block in which the statement pointed to by the ip is nested. Together the ip and the ep of the processor form its site of activity. (The ep is the arrow coming out of the left-hand side of a π.)
   3) a state which is either awake, asleep or terminated. (An awake processor is denoted by a solid π; an asleep processor by a dotted π→ with its ip drawn as an arrow, and a terminated processor by a dotted π•— with a null ip.)

All cells in the record of execution are allocated from free storage and remain allocated as long as they are accessible to some awake processor.

A cell c is accessible to an awake processor if and only if one of the following three is true:
   a) c is an awake processor.
   b) c is pointed to by the ep of an awake processor.
   c) c is pointed to by some pointer p which is stored in a subcell of some cell d which is accessible to an awake processor (the most important such pointer p is the static link of a contour).

A cell of the record of execution is deallocated immediately when it is no longer accessible to any awake processor.

3) The input/output component contains four subcomponents:
   a) an input list selected by input_{CM}.
   b) an input pointer which points to the next item on the input list to be read.
   c) an output list selected by output_{CM}.
   d) an output pointer which points to the last item on the output list that was printed.

Initial snapshots and compile_{CM}

Application of the function compile_{CM} to a program p and an input list δ will produce the initial snapshot for this program in CM. In the initial snapshot the algorithm is the program p with the lines numbered. There is a processor π whose ip

*Technically, a contour is one big cell allocated and deallocated as a unit. The contour's constituent subcells are called cells without fear of confusion.

of the same statement are numbered the same. (The stack-of-texts is shown opening to the right.)

b) an instruction pointer, ip, which is a number pointing to the next statement to be executed by the task stack. The statement actually executed is the one in the top text of the stack-of-texts whose number is ip. (The ip is shown as a horizontal arrow pointing in from the right.)

c) a task name, tn_i, with $i \geq 1$, which uniquely identifies the task stack. (The task name is in the little box on the upper left-hand corner of the task stack.)

d) A state, which is either awake, asleep, or terminated (an awake task stack has solid boundary lines; an asleep task stack has dotted boundary lines and an arrow as an ip; a terminated task stack has dotted boundary lines and a null ip, "•— ").

2) A task name generator, TNG, which is incremented by 1 at each task creation and whose value is used to form the task hame, tn_i, for the created task stack (in the lower left-hand corner).

3) A block entry count generator, BECG, which is incremented by 1 at each block or procedure entry. At block or procedure entry the current value of the BECG is used to form unique names (addresses) by subscripting all identifying occurrences of locally declared identifiers with this value (in the lower left-hand corner).

4) A countably infinite two-dimensional storage component whose locations are addressed by ordered pairs of the form (id,n) where id is an identifier and n is an integer. The unique name id_n addresses location (id n) of the storage. Storage is never destroyed. (This is the array in the bottom half of the snapshots; the identifiers label the columns and the integers label the rows.)

5) The input/output component contains four subcomponents:
   a) An input list selected by input_{CR}.
   b) An input pointer which points to the next item on the input list to be read.
   c) an output list selected by output_{CR}.
   d) an output pointer which points to the last item on the output list that was printed.

Initial snapshot and compile_{CR}

Application of the function compile_{CR} to a program p and an input list δ produces the initial snapshot for the execution of the program with the input list. The initial snapshot consists of a single awake task stack named tn_1, with the text of points to the first statement in line and whose ep is null (grounded arrow). The input list δ is made to be the $\underline{\text{input}}_{CM}$ component. The input pointer is set to point to the first item on the list. The output list and the output pointer are both empty.

the program p on top of the stack, the ip pointing to the first statement, the BECG set to 1, the TNG set to 2 (because tn_1 has been allocated already), and a completely uninitialized storage. The input list δ has been made to be the input_{CR} component of the snapshot. The intput pointer is set to point to the first item on the input list. Both the output list and the output pointer are empty at this time.

Example: Snapshot 0 (the input list is null with this program).

Transformation

The single transformation f of CM is the following five-step procedure:

1) Select one awake processor: if there are none, the computation halts.
2) Fetch the instruction pointed to by the selected processor's ip.
3) Sequence the selected processor's ip to point to the next statement.
4) Execute the fetched instruction.
5) Start all over at (1) with a new snapshot.

Transformation

The single transformation of CR is the following five-step procedure:

1) Select some awake task stack. If there are none, the computation halts.
2) Fetch the statement in the topmost text of the selected task stack's stack which is pointed to by the selected task stack's ip.
3) Sequence the selected task stack's ip to point to the next statement.
4) Execute the fetched statement.
5) Start all over at step (1) with a new snapshot.

Step (1) attempts to reflect the nondeterminism inherent in asynchonous execution of tasks by dispatching for one instruction a nondeterministically selected processor or task stack. Step (4) is broken up into sub-cases, one for each statement and expression type of GBSL.

In the following subsections, we give the details of step (4) of the transformation for the important statement types and expression types. (All uses of the words "processor" or "task stack" will refer to the selected or executing processor or task stack unless otherwise stated.)

Block entry

A contour is allocated with a cell in its declaration array for each identifier declared in the entered block and a cell for the static link. The static link is made to be a copy of the processor's current ep. Finally, the processor's ep is set to point to the new contour.

Block entry

A new text of the program is pushed into the stack. The new text is obtained from the previous top-of-stack text by using the current value of BECG to subscript all identifying instances (in the entered block) of each identifier declared in the block.* Also, the begin delimiter of the block is subscripted with the current value of the BECG. Then the BECG is incremented by 1. Finally, the storage location for each newly created unique name (subscripted declared identifier) is marked declared.

Note that the ip of the processor or task stack has already been advanced to point to the first statement in the block entry.

Examples: #1: before: snapshot 0
after: snapshot 1

#2: before: snapshot 1
after: snapshot 2

Block exit

The processor's ep is made to be a copy of the static link of the contour pointed to by the processor's current ep. (Note that the processor's ip has already been advanced.) If the contour has become inaccessible to all awake processors, it is deallocated. (We discuss this in more detail in the section on retention.)

Examples: #1: before: snapshot 6
after: snapshot 7
(deallocation of contour)

Block exit

The top text on the executing task stack's stack-of-texts is popped. The ip has already been advanced to point to the statement after the block. The storage component is not changed. As we shall see, the fact that storage is not changed now effectively results in retention.

#2: before: snapshot 8
after: snapshot 9
(no deallocation of contour since it is still accessible to right-hand processor)

Accessing environment and designated cell

The accessing environment of a processor is the ordered list of contours such that the first of these is the contour pointed to by the processor's ep and each successive contour on the list is pointed to by the static link of the previous contour on the list. (By the way the contours are drawn, it is the list of contours surrounding the processor.) The cell designated by the use of an identifier is the one in the contour such that it is the first on the list with a cell for that identifier (that is, the innermost cell with that identifier as the label). If there is no such cell the use is in error because the identifier has not been declared.

Referred-to location

There is no complicated procedure for looking up the value of an identifier. By the time an identifier is used, it has been subscripted (if not, the use of the identifier is not within the scope of a declaration; i.e., the use is in error because the identifier has not been declared). The location referred to by the subscripted identifier $\underline{id}_n$ is the one addressed by (id,n).

*Only those uses which identify the declarations in the entered block are subscripted.

Examples: In snapshot 3, the accessing environment of the processor consists of cells for ℓ,p,a and t in the outer contour and of cells for i and $\aleph$ in the right-hand inner contour. The cell for a in the left-hand inner contour is not included in the processor's accessing environment. In snapshot 8 the two processors share access to the cells for ℓ, p,a, and t in the outer contour, but only the right-hand processor includes the cells for i and $\aleph$ in its accessing environment.

Assignments

To execute an assignment statement the value of the right-hand side is stored into the cell designated by the left-hand side. The value of the right-hand side is computed by using the values stored in the cells designated by the identifiers in the right-hand side.

Assignments

When executing an assignment statement the value of the right-hand side is assigned to the location referred to by the left-hand side. The value of the right-hand side is computed using the contents of the locations referred to by the subscripted identifiers in the right-hand side.

Examples: #1: before: snapshot 3
after: snapshot 4

#2: before: snapshot 9
after: snapshot 10
(note: no change in the value of a)

Label declaration and initialization

A label constant identifier is assumed to be declared in the innermost block in which the identifier appears as a statement label.

A goto may be thought of as specifying a new site of activity for the executing processor. Since both an ip and ep are required to completely specify a site of activity, a label must be a data item and must consist of an ip and ep. A label constant is initialized upon entry to the block in which it is declared. Its ip points to the labelled statement in the algorithm and its ep points to the contour allocated for the block.

Example: The label constant ℓ in line 10 is assumed to be declared in the outer block. In snapshot 1, just after entry to the outer block, the cell for ℓ has an ip pointing to line 10 and an ep pointing to the contour in which the cell resides.

In CR, a label value consists of an ip and a stack-of-texts; that is, it contains all the information that a task stack does except for the task name and state.

A label constant, ℓ, is initialized upon entry to the block in which it is assumed to be declared. If the value of the BECG before entry is n, then in location (ℓ,n) is stored an ip pointing to the labelled statement and a copy of the stack-of-texts of the executing task stack just after the entry. This is so that the identifiers declared in the same block that the label is will already be modified in the top text of the label's stack.

Example: Snapshot 1 contains the first appearance of the label value in (ℓ,1).

Gotos

The label value is obtained from the cell designated by the target label. The executing processor's ip and ep are reset to copies of that of the label value.

Gotos

The label value is obtained from the location referred to by the target label. The executing task stack's ip and stack-of-texts are reset to copies of that of the label value.

Example: before: snapshot 5
after: snapshot 7

(Note that the goto has been arranged to make double use of snapshot 7. We assume that the random number that was generated equals .5.)

Procedure declarations and initialization

A procedure value is an (ip,ep) pair since a procedure call is a goto with saving of a return label. Also, the ep points to the environment which includes the cells designated by the nonlocal identifiers in the procedure body.

A procedure identifier is initialized upon entry to the block in which the procedure is declared. The cell for the identifier is initialized with an ip pointing to the entry point of the procedure in the algorithm and an ep pointing to the contour in which the cell resides.

Example: In snapshot 1, just after entry to the outer block in which p is declared, the cell for p in the outer contour has an ip pointing to line 2 and an ep pointing to that contour. Note that the outer contour contains the cell designated by the nonlocal a in the body of p. Thus at call time we shall be able to access the cell for a.

Procedure declaration and initialization

In CR, a procedure value consists of an ip and a single modified text of the program***. Upon entry to a block in which a procedure p is declared, if the value of the BECG is n just before entry, location (p,n) is initialized with an ip pointing to the entry point of the procedure and with a copy of the top text of the stack just after entry to the block. Note that nonlocals in the procedure body are already modified in the text associated with the procedure value. Consequently, we have a means of accessing the proper locations for these nonlocals when the procedure is called.

Example: Snapshot 1 contains the first appearance of the procedure value in (p,1).

***The reason a procedure is not a full stack like a label is that lower levels of the stack are not needed. To see this, observe that there is no way to get out of a cell except by a return or a goto, both of which replace the stack.

Procedure calls

A contour with a cell for each of the parameters and a cell for a return label, ℵ*, in its declaration array is allocated. The cells for the parameters are filled in with the values** of the corresponding actual arguments. The cell for ℵ is filled in with the current ip and ep of the processor. (Remember that the processor's ip has already been sequenced to point to the next statement. Thus the return label's site of activity designates the statement after the call in the calling environment.) The static link of the new contour is set to a copy of the ep of the called procedure. Then the processor resets its ep to point to the new contour and its ip to point to the first executable statement after the entry point.

Procedure calls

The text of the program which is in the procedure value is obtained and pushed into the executing task stack. In this copy, the procedure delimiter and all identifying occurrences of the formal parameter identifiers are subscripted by the current value of the BECG. The values of the actual parameters are stored in the appropriate formal parameter locations in storage. The ending semicolon of the procedure body is replaced by goto $\aleph_i$ where i is the current value of the BECG. A label (an ip and a stack) indicating the return site of activity is stored into (,i). The BECG is incremented by 1 and the ip of the task stack is set to point to the first executable statement in the procedure body (this is obtained from the ip of the procedure value).

Example: before: snapshot 2
after: snapshot 3

Procedure exit

Upon reaching the end of the procedure body, the processor simulates execution of goto ℵ. This results in using the return label stored in the cell for ℵ in the present innermost contour (the one allocated for the call); the processor goes to the statement just after the call in the environment in which the call was made.

Procedure exit

Procedure exit occurs by explicit execution of goto $\aleph_i$ which has been written at the end of the procedure body.

Example: before: snapshot 4
after: snapshot 5

Task creation

A task, as mentioned before, is a processor. Thus creation of a task means that one must allocate and initialize a processor. Initializing a processor means giving it a site of activity. Therefore a task creation statement must provide an (ip,ep) pair of some kind.

Task creation

As observed earlier, a task is an ordered quadruple (stack-of-texts, ip, state, task name). Creating a task involves creating a new task stack with the stack-of-texts and ip obtained from a label or a procedure value.

Call procedure as task

If p (actual params) task is executed, a contour is allocated with cells in its declaration array for each of the formal parameters and the return label, ℵ. The formal parameters are initialized to the values of the corresponding actual parameters and the return label is set to nil. (Thus when a processor tries to execute a return with a nil label, it will terminate.) A new processor is allocated. Its ep is set to point to the new contour, its ip is set to point to the first statement of the body, and its state is set to "awake".

Call procedure as task

If p_i(actual params) task is executed, then the executing task creates a new task stack with a stack of height 2. The bottom text is the original program which may be copied from the executing task's stack. The top text is formed by modifying the text stored with the procedure value in location (p,i). In this text, the procedure delimiter of the entered procedure and the identifying occurrences of the entered procedure's formal parameters are subscripted with the current value of the BECG. The formal parameter storage locations are initialized to the corresponding actual parameter values. The end semicolon is replaced by goto $\aleph_i$ where i is the current value of the BECG. Since we want termination at the end of the body, nil is stored in location (ℵ,i). The ip of the new task stack is set to point to the entry point of the called procedure. Then the BECG is incremented by 1. Next, the state of the new task stack is set to "awake" and the task name is set to tn_i where i is the current value of the TNG. Finally, the TNG is incremented by 1.

Example: before: snapshot 7
after: snapshot 8

Tasked goto

If goto ℓ task is executed, a new processor is allocated. Its ep and ip are initialized to those of the label stored in the cell designated by ℓ. The state of the new processor is set to "awake".
Example: Suppose line 8 read

if random=.5 then goto ℓ task

and in snapshot 5 random returned .5; then we have:

Tasked goto

If goto ℓ_i task is executed, the executing task creates a new task stack with the stack-of-texts and ip being a copy of those of the label value stored in location (ℓ,i). The state is set to "awake" and the task name is formed by subscripting tn by the current value of the TNG. The TNG is then incremented by 1.

*We use the letter aleph, ℵ, to identify the return label so that it will not conflict any programmer-defined variables.

**That is, the parameter passing mode is assumed to be call-by-value. To pass a parameter by name, pass a parameterless procedure by value; to pass a parameter by reference, pass a pointer by value.

Example: Suppose line 8 read

```
        if random=.5 then goto ℓ task.
```

Then in snapshot 5 line 8 would read

```
        if random=.5 then goto ℓ1 task.
```

If the random number generator did indeed return .5, then we have:

before: snapshot 5
after: extra snapshot (at end of set of snapshots) (Assume line 8 reads properly in these snapshots.)

Set task variable

If the task creation statement specifies a task variable to name the created task, as in p(5) task t, then, in addition to the steps done to create the task, a pointer pointing to the new processor is stored in the cell designated by the task variable.

Set task variable

If the task creation statement specifies a task variable to name the created task, as in p_1(5) task t_1, then in addition to the steps done to create the task, the task name of the new task stack, is stored in the location referred to by the task variable.

Example:
before: snapshot 7
after: snapshot 8

Task blocking, termination and resuming

Execution of

```
    block t; terminate t; or resume t;
```

where t is a task variable, results in setting the state of the processor pointed to by the value of t to "asleep", "terminated" or "awake", respectively. This happens only if the processor pointed to by the value of t is not already terminated.

Execution of

```
    block;      or      terminate;
```

changes the status of the executing processor to "asleep" or "terminated", respectively.

A task may also be terminated in the following two manners:

1. It reaches the end of a procedure body and nil is stored in the local cell for $\aleph$.
2. It runs out of instructions to execute, for example by exiting the outer block.

In these cases the executing task changes its status to "terminated".

In any of the above operations, if an asleep or terminated processor should become inaccessible to all awake processors then it is deallocated. (Recall that an awake processor is considered to be accessible to itself.)

Task blocking, termination and resuming

Execution of

```
    block ti; terminate ti; or resume ti;
```

where t_i is a subscripted version of a task variable t, results in setting the state of the task stack whose name is stored in location (t,i) to "asleep", "terminated" or " awake", respectively. This happens only if the target task stack has not already become terminated.

Execution of

```
    block;      or      terminate;
```

changes the status of the executing task stack to "asleep" or "terminated", respectively.

A task stack may also become terminated in the following two manners:

1. It reaches the end of a procedure body and attempts to execute goto$\aleph_i$ with nil stored in ($\aleph$,i).
2. It runs out of instructions to execute, for example by exiting the outer block.

In these cases the executing task stack changes its status to "terminated".

Examples:
termination in manner 1: before: snapshot 10, after: snapshot 11
in manner 2: before: snapshot 8, after: snapshot 9

Task synchronization

A semaphore is a data cell whose purpose is to limit the number of tasks that use the resource that the semaphore represents, e.g. a critical section of code. A semaphore cell consists of two fields, the value field and the queue field.

The value field contains an integer value; if the value is negative, then it is the number of tasks waiting for the use of the resource. If the value is positive, it is the number of tasks that will be allowed to use the resource without waiting. If the value is zero, then the resource is fully utilized and no task is waiting for it.

The queue field contains the head of a doubly linked list of the processors which are waiting for the value of the semaphore to become zero (from a negative value).

The only ways to affect a semaphore's value are to declare it with a certain value or to request or release it.

The declaration

```
        sema s with 1;
```

results in storing 1 in the value field of the cell for s in the contour being allocated.

The effect of executing request(s) may be described by the following code:

```
 request(s) ≡ value of s:=value of s-1;
    if value of s<0 then
       add executing processor to the queue of
       s and block the executing processor;
```

The queue field contains an ordered list of task names.

The only ways to affect a semaphore's value are to declare it with a certain value or to request or release it.

The declaration

```
        sema s with 1;
```

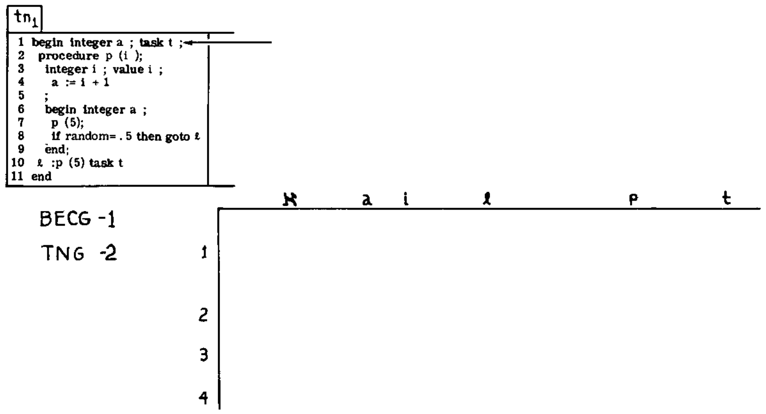
will result in 1 being stored in the value field of location (s,i) where s_i is the modification of s that results from the actions of block entry.

The execution of request(s_i) is described by the following code:

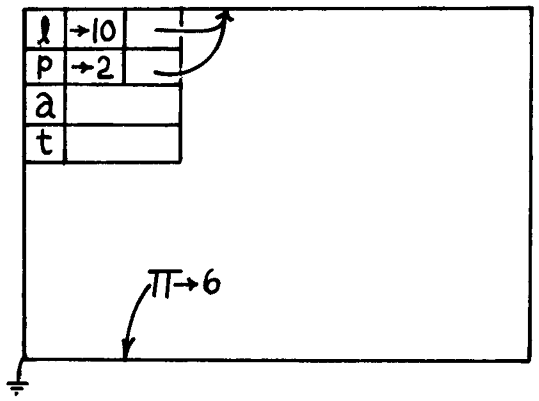
```
 request(si) ≡ value of si:=value of si-1;
    if value of si<0 then add task name of
      executing task to the list in queue of
      si and set state of executing task to
      "asleep";
```

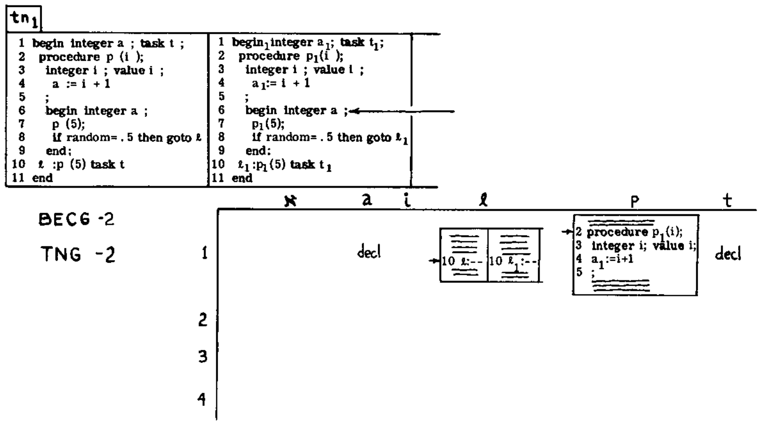
The execution of release(s_i) is described by the

```
1 begin integer a ; task t ;
2  procedure p (i );
3   integer i ; value i ;
4    a := i + 1
5   ;
6   begin integer a ;
7    p (5);
8    if random= . 5 then goto ℓ
9   end;
10  ℓ :p (5) task t
11 end
```

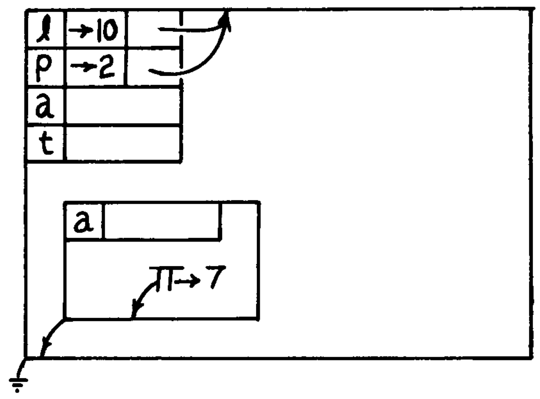


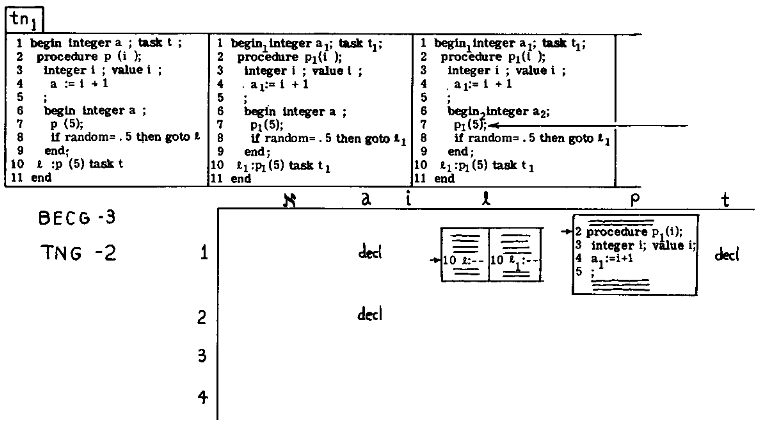
SNAPSHOTS 0



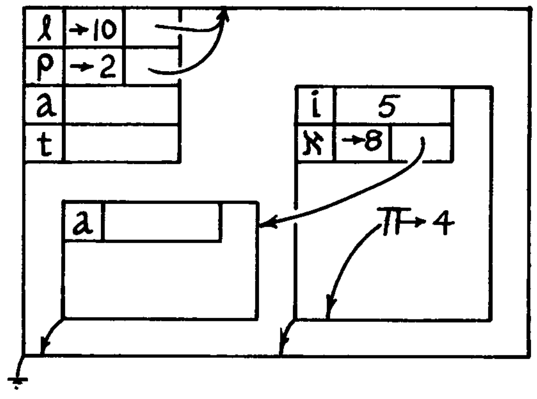


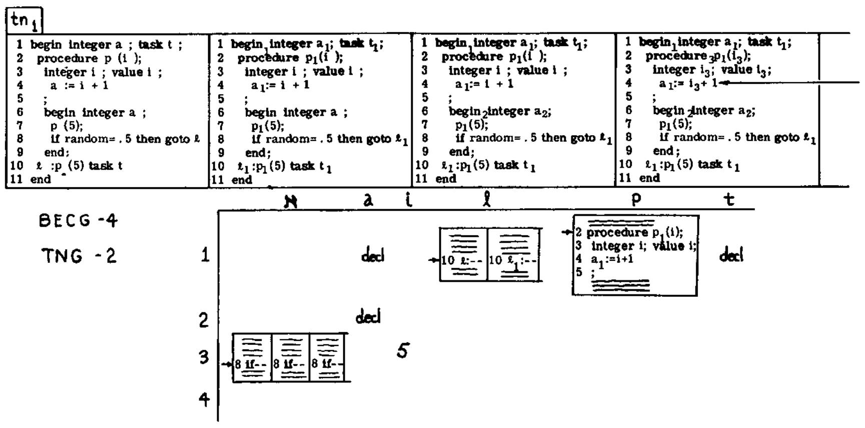
SNAPSHOTS 1



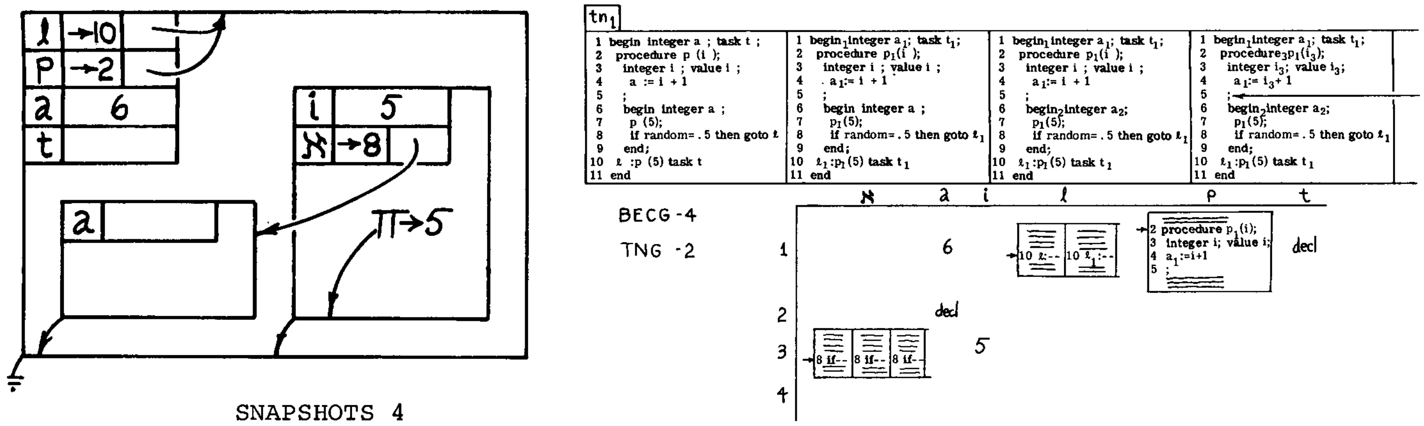


SNAPSHOTS 2





SNAPSHOTS 3



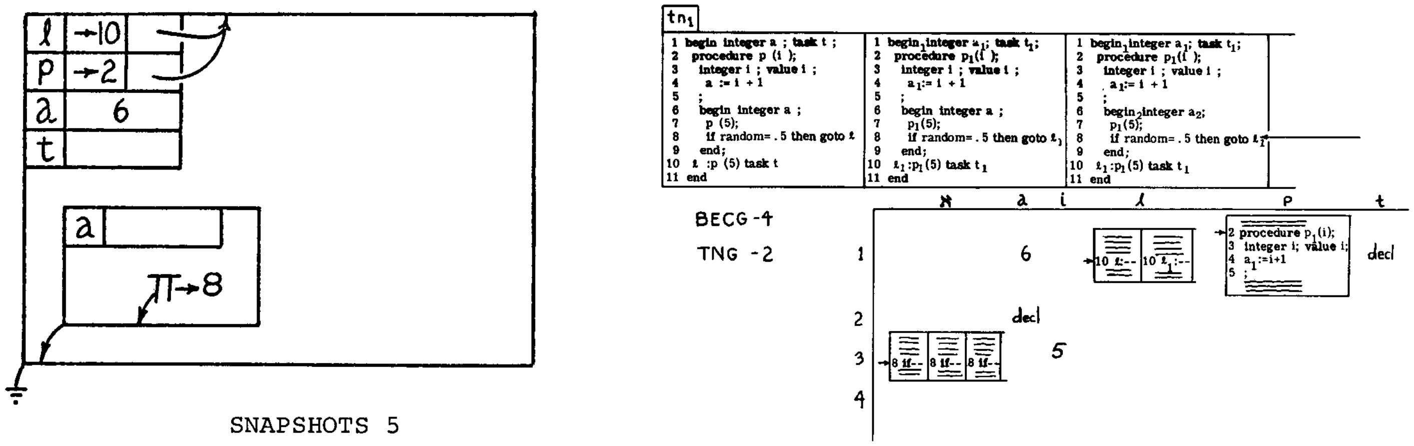
```
1  begin integer a ; task t ;
2   procedure p (i );
3    integer i ; value i ;
4     a := i + 1
5    ;
6    begin integer a ;
7     p (5);
8     if random= .5 then goto ℓ
9    end;
10  ℓ :p (5) task t
11 end
```

```
1  begin₁integer a₁; task t₁;
2   procedure p₁(i );
3    integer i ; value i ;
4     a₁:= i + 1
5    ;
6    begin integer a ;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

```
1  begin₁ integer a₁; task t₁;
2   procedure p₁(i );
3    integer i ; value i ;
4     a₁:= i + 1
5    ;
6    begin₂integer a₂;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

```
1  begin₁integer a₁; task t₁;
2   procedure₃p₁(i₃);
3    integer i₃; value i₃;
4     a₁:= i₃+ 1
5    ;
6    begin₂integer a₂;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

SNAPSHOTS 4

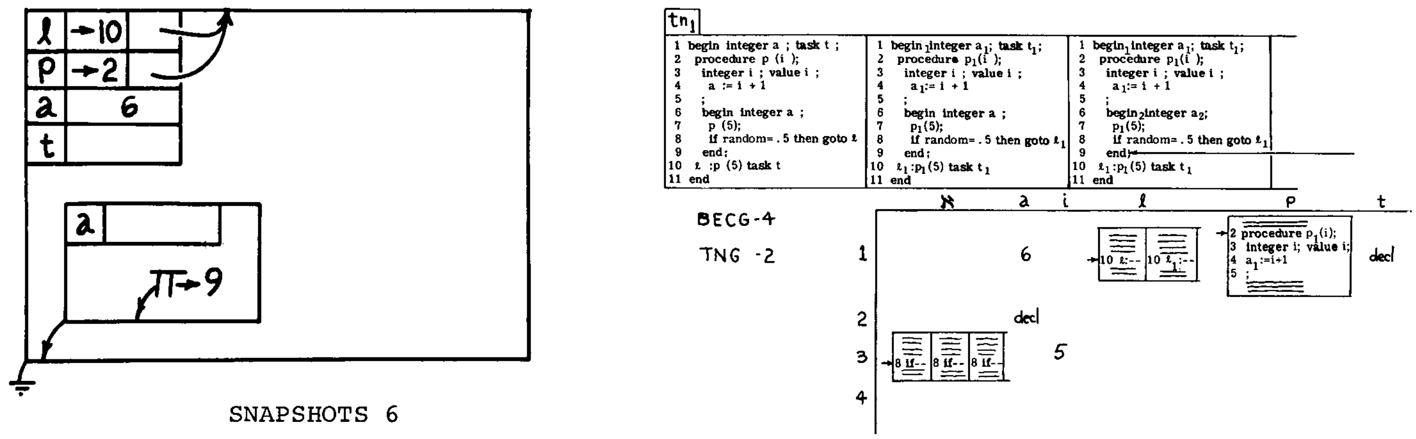


```
1  begin integer a ; task t ;
2   procedure p (i );
3    integer i ; value i ;
4     a := i + 1
5    ;
6    begin integer a ;
7     p (5);
8     if random= .5 then goto ℓ
9    end;
10  ℓ :p (5) task t
11 end
```

```
1  begin₁integer a₁; task t₁;
2   procedure p₁(i );
3    integer i ; value i ;
4     a₁:= i + 1
5    ;
6    begin integer a ;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

```
1  begin₁integer a₁; task t₁;
2   procedure p₁(i );
3    integer i ; value i ;
4     a₁:= i + 1
5    ;
6    begin₂integer a₂;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

SNAPSHOTS 5

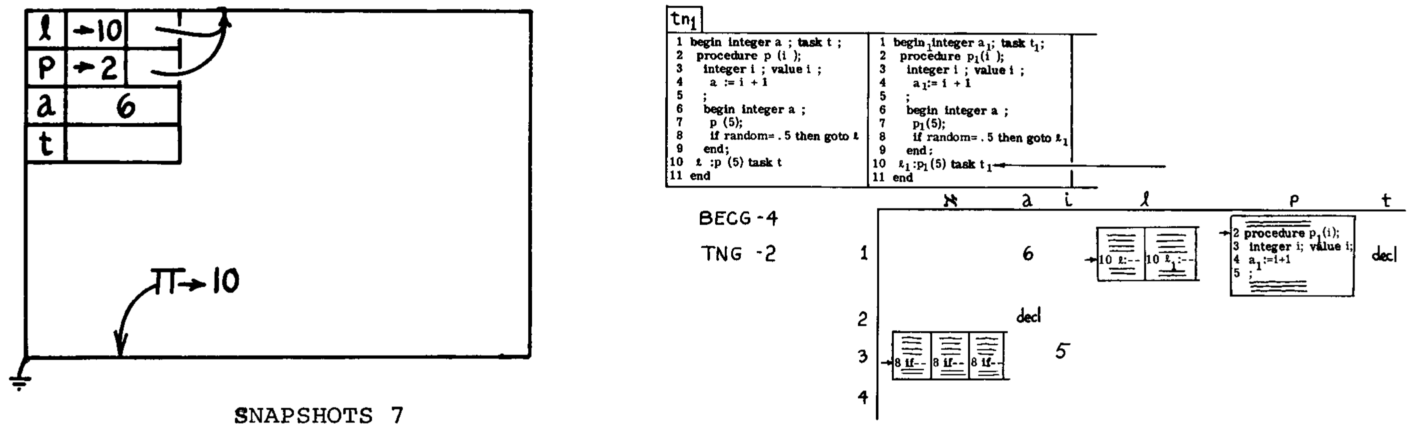


```
1  begin integer a ; task t ;
2   procedure p (i );
3    integer i ; value i ;
4     a := i + 1
5    ;
6    begin integer a ;
7     p (5);
8     if random= .5 then goto ℓ
9    end;
10  ℓ :p (5) task t
11 end
```

```
1  begin₁integer a₁; task t₁;
2   procedure p₁(i );
3    integer i ; value i ;
4     a₁:= i + 1
5    ;
6    begin integer a ;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

```
1  begin₁integer a₁; task t₁;
2   procedure p₁(i );
3    integer i ; value i ;
4     a₁:= i + 1
5    ;
6    begin₂integer a₂;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

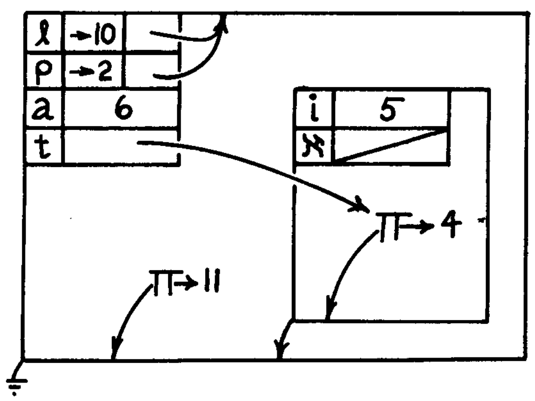
SNAPSHOTS 6



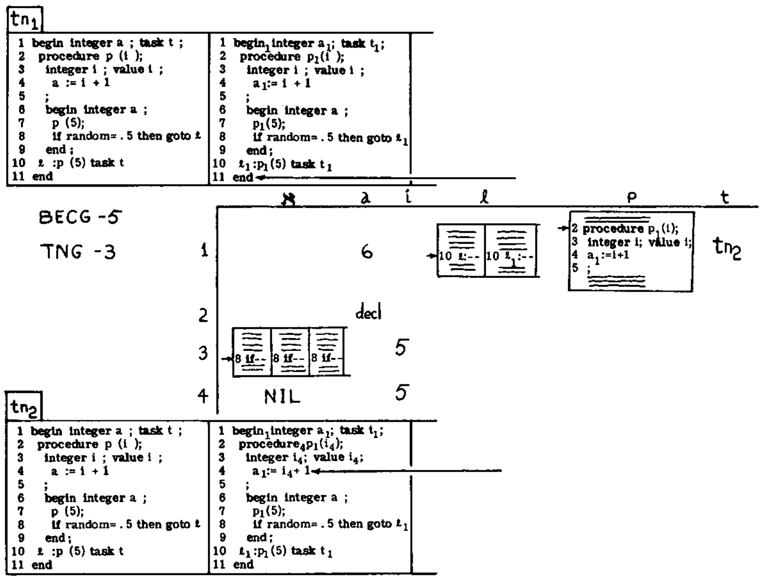
```
1  begin integer a ; task t ;
2   procedure p (i );
3    integer i ; value i ;
4     a := i + 1
5    ;
6    begin integer a ;
7     p (5);
8     if random= .5 then goto ℓ
9    end;
10  ℓ :p (5) task t
11 end
```

```
1  begin₁integer a₁; task t₁;
2   procedure p₁(i );
3    integer i ; value i ;
4     a₁:= i + 1
5    ;
6    begin integer a ;
7     p₁(5);
8     if random= .5 then goto ℓ₁
9    end;
10  ℓ₁:p₁(5) task t₁
11 end
```

SNAPSHOTS 7



SNAPSHOTS 8

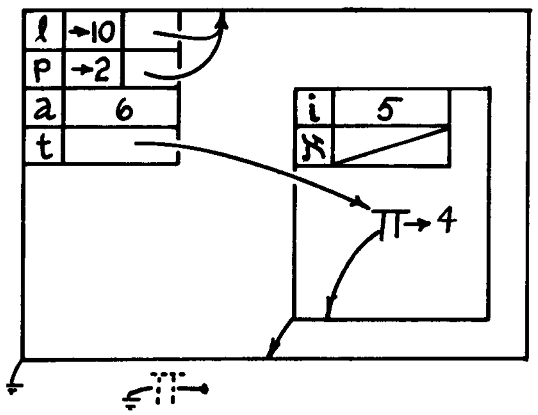


```
1  begin integer a ; task t ;
2  procedure p (i );
3    integer i ; value i ;
4      a := i + 1
5    ;
6    begin integer a ;
7      p (5);
8      if random= .5 then goto ℓ
9    end ;
10 ℓ :p (5) task t
11 end
```

```
1  begin1 integer a1; task t1;
2  procedure p1(i );
3    integer i ; value i ;
4      a1:= i + 1
5    ;
6    begin integer a ;
7      p1(5);
8      if random= .5 then goto ℓ1
9    end;
10 ℓ1:p1(5) task t1
11 end
```

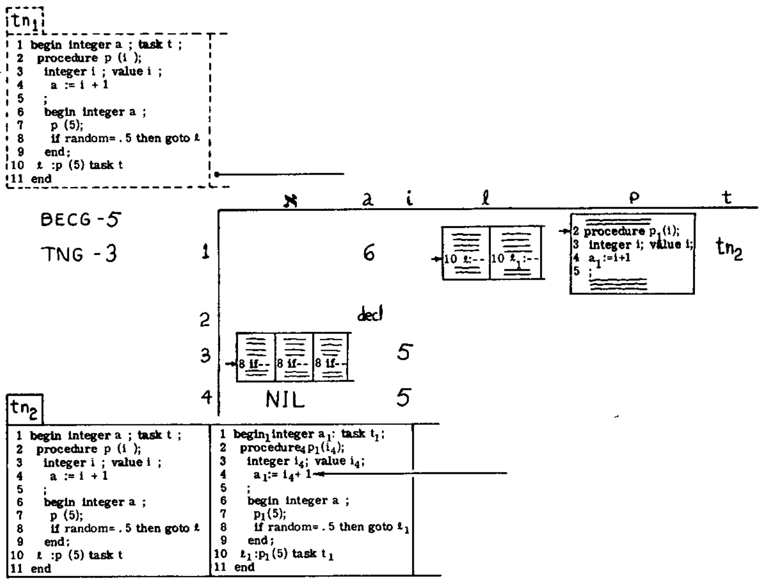
```
1  begin integer a ; task t ;
2  procedure p (i );
3    integer i ; value i ;
4      a := i + 1
5    ;
6    begin integer a ;
7      p (5);
8      if random= .5 then goto ℓ
9    end;
10 ℓ :p (5) task t
11 end
```

```
1  begin1integer a1; task t1;
2  procedure4p1(i4);
3    integer i4; value i4;
4      a1:= i4+ 1
5    ;
6    begin integer a ;
7      p1(5);
8      if random= .5 then goto ℓ1
9    end;
10 ℓ1:p1(5) task t1
11 end
```



The terminated processor should already be deallocated in this snapshot.

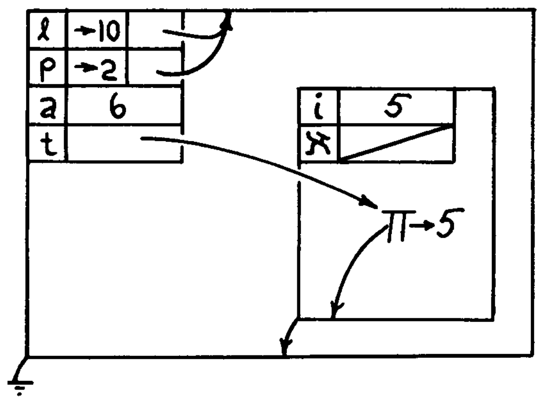
SNAPSHOTS 9



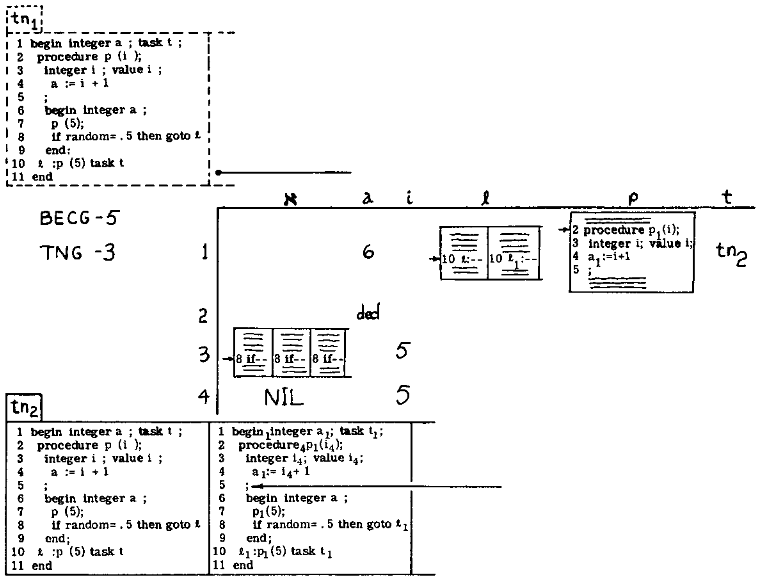
```
1  begin integer a ; task t ;
2  procedure p (i );
3    integer i ; value i ;
4      a := i + 1
5    ;
6    begin integer a ;
7      p (5);
8      if random= .5 then goto ℓ
9    end;
10 ℓ :p (5) task t
11 end
```

```
1  begin integer a ; task t ;
2  procedure p (i );
3    integer i ; value i ;
4      a := i + 1
5    ;
6    begin integer a ;
7      p (5);
8      if random= .5 then goto ℓ
9    end;
10 ℓ :p (5) task t
11 end
```

```
1  begin1integer a1; task t1;
2  procedure4p1(i4);
3    integer i4; value i4;
4      a1:= i4+ 1
5    ;
6    begin integer a ;
7      p1(5);
8      if random= .5 then goto ℓ1
9    end;
10 ℓ1:p1(5) task t1
11 end
```



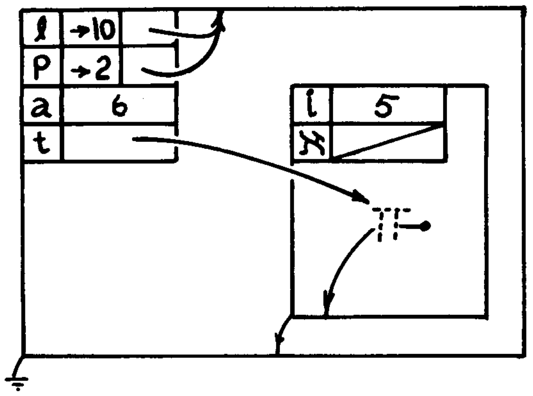
SNAPSHOTS 10



```
1  begin integer a ; task t ;
2  procedure p (i );
3    integer i ; value i ;
4      a := i + 1
5    ;
6    begin integer a ;
7      p (5);
8      if random= .5 then goto ℓ
9    end;
10 ℓ :p (5) task t
11 end
```

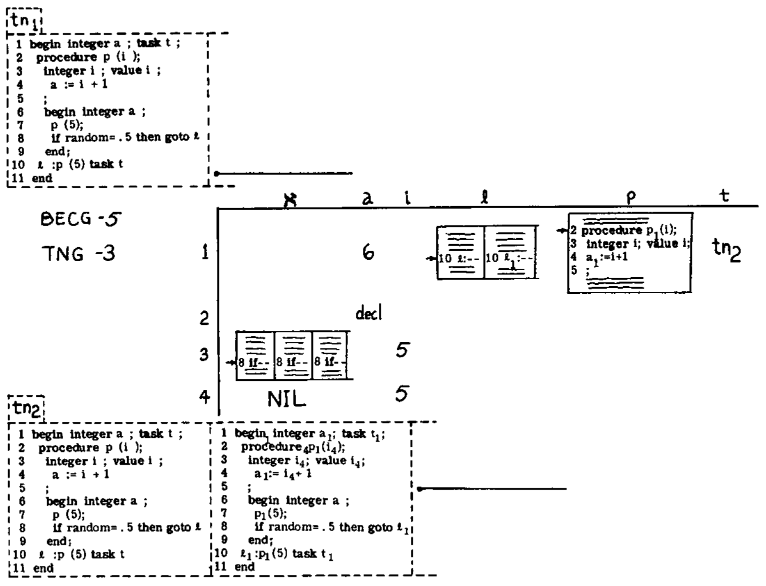
```
1  begin integer a ; task t ;
2  procedure p (i );
3    integer i ; value i ;
4      a := i + 1
5    ;
6    begin integer a ;
7      p (5);
8      if random= .5 then goto ℓ
9    end;
10 ℓ :p (5) task t
11 end
```

```
1  begin1integer a1; task t1;
2  procedure4p1(i4);
3    integer i4; value i4;
4      a1:= i4+ 1
5    ;
6    begin integer a ;
7      p1(5);
8      if random= .5 then goto ℓ1
9    end;
10 ℓ1:p1(5) task t1
11 end
```



The entire record of execution should already be deallocated.

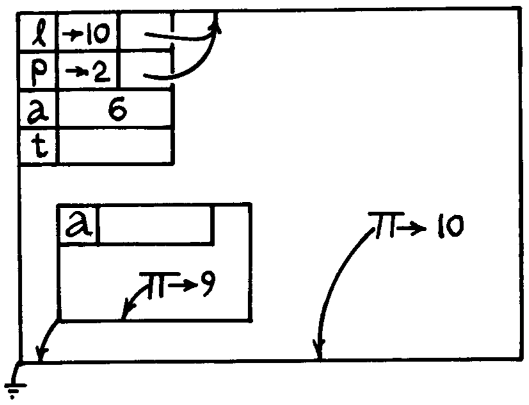
SNAPSHOTS 11
(FINAL)



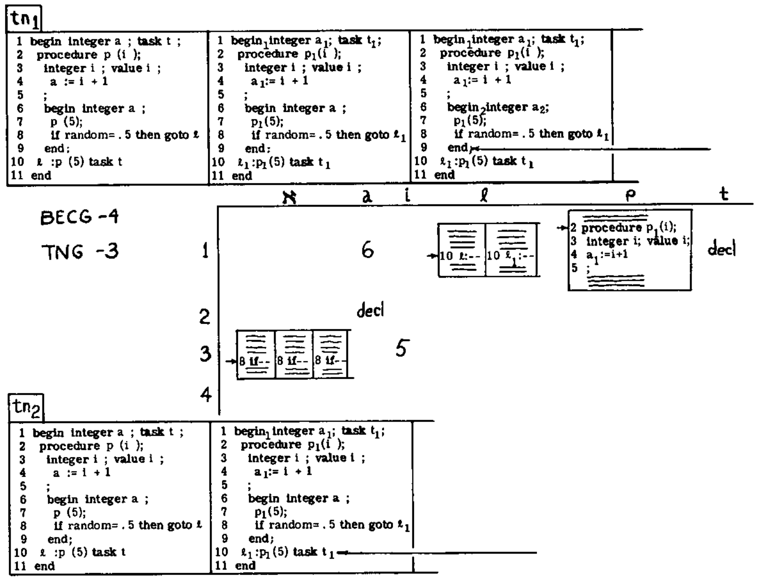
```
1 begin integer a ; task t ;
2 procedure p (i );
3 integer i ; value i ;
4 a := i + 1
5 ;
6 begin integer a ;
7 p (5);
8 if random= . 5 then goto ℓ
9 end;
10 ℓ :p (5) task t
11 end
```

```
1 begin integer a ; task t ;
2 procedure p (i );
3 integer i ; value i ;
4 a := i + 1
5 ;
6 begin integer a ;
7 p (5);
8 if random= . 5 then goto ℓ
9 end;
10 ℓ :p (5) task t
11 end
```

```
1 begin1 integer a1; task t1;
2 procedure4 p1(i4);
3 integer i4; value i4;
4 a1:= i4+ 1
5 ;
6 begin integer a ;
7 p1(5);
8 if random= . 5 then goto ℓ1
9 end;
10 ℓ1 :p1(5) task t1
11 end
```



EXTRA SNAPSHOTS



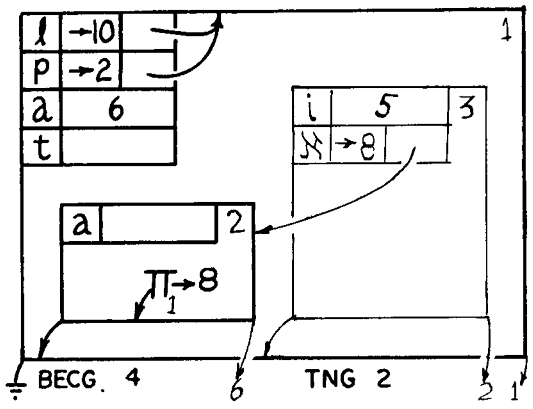
```
1 begin integer a ; task t ;
2 procedure p (i );
3 integer i ; value i ;
4 a := i + 1
5 ;
6 begin integer a ;
7 p (5);
8 if random= . 5 then goto ℓ
9 end;
10 ℓ :p (5) task t
11 end
```

```
1 begin1 integer a1; task t1;
2 procedure p1(i );
3 integer i ; value i ;
4 a1:= i + 1
5 ;
6 begin integer a ;
7 p1(5);
8 if random= . 5 then goto ℓ1
9 end;
10 ℓ1 :p1(5) task t1
11 end
```

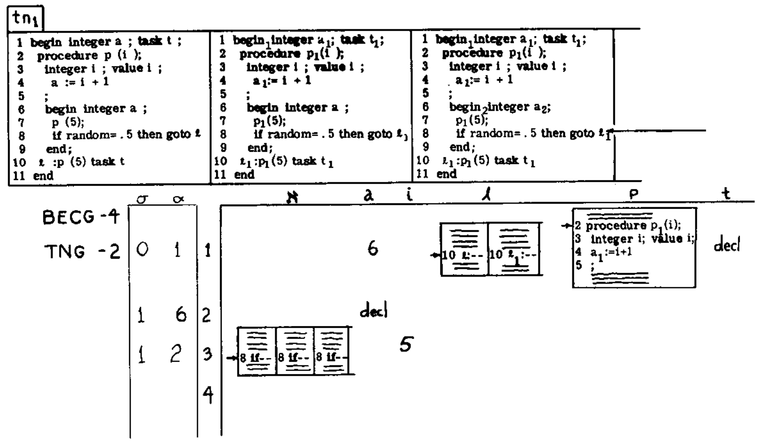
```
1 begin1 integer a1; task t1;
2 procedure p1(i );
3 integer i ; value i ;
4 a1:= i + 1
5 ;
6 begin2 integer a2;
7 p1(5);
8 if random= . 5 then goto ℓ1
9 end;
10 ℓ1 :p1(5) task t1
11 end
```

```
1 begin integer a ; task t ;
2 procedure p (i );
3 integer i ; value i ;
4 a := i + 1
5 ;
6 begin integer a ;
7 p (5);
8 if random= . 5 then goto ℓ
9 end;
10 ℓ :p (5) task t
11 end
```

```
1 begin1 integer a1; task t1;
2 procedure p1(i );
3 integer i ; value i ;
4 a1:= i + 1
5 ;
6 begin integer a ;
7 p1(5);
8 if random= . 5 then goto ℓ1
9 end;
10 ℓ1 :p1(5) task t1
11 end
```



SNAPSHOTS S_5 AND S_5'
FOR SECTION 8



```
1 begin integer a ; task t ;
2 procedure p (i );
3 integer i ; value i ;
4 a := i + 1
5 ;
6 begin integer a ;
7 p (5);
8 if random= . 5 then goto ℓ
9 end;
10 ℓ :p (5) task t
11 end
```

```
1 begin1 integer a1; task t1;
2 procedure p1(i );
3 integer i ; value i ;
4 a1:= i + 1
5 ;
6 begin integer a ;
7 p1(5);
8 if random= . 5 then goto ℓ1
9 end;
10 ℓ1 :p1(5) task t1
11 end
```

```
1 begin1 integer a1; task t1;
2 procedure p1(i );
3 integer i ; value i ;
4 a1:= i + 1
5 ;
6 begin2 integer a2;
7 p1(5);
8 if random= . 5 then goto ℓ1
9 end;
10 ℓ1 :p1(5) task t1
11 end
```

Similarly, we may describe release(s) as

```
release(s) ≡ value of s:=value of s+1;
   if value of s<0 then
      remove any processor from queue of s and
      awaken it;
```

following code:

```
release($s_i$) ≡ value of $s_i$:=value of $s_i$+1;
   if value of $s_i$<0 then remove any $tn_j$ from
      list in queue of $s_i$ and set state of the
      task stack named $tn_j$ to "awake";
```

Both of these operations are indivisible; that is, only one processor or task stack may be manipulating any given semaphore at a time. The effect of saying that request and release are indivisible is to make their many-step executions appear as one step of the computation.

Notice that the code for release specifies that any processor (task stack) in the queue is to be removed from the queue and awakened. This nondeterminism is to allow modelling of varied queueing schemes without actually saying what they are.

Input/Output

Recall that a snapshot contains an input pointer which points to the next item on the input list to be read and an output pointer which points to the last item on the output list that was printed. We assume that integer, real, boolean and character items may be read or printed in one input/output operation. We also assume that in GBSL there are two language-defined input/output procedures, read and print. Read is a parameterless procedure which picks up the item pointed to by the input pointer, resets the input pointer to point to the next item on the input list, and returns as its result the item that was picked up. Print is a single-parameter, resultless procedure which resets the output pointer to point to the next position on the output list and places the value passed as its parameter in that position.

Retention

In CM we have specified that a cell is deallocated when it becomes inaccessible to all awake processors.

This scheme leads to automatic deallocation (when allowed), in the usual manner, upon block exits, gotos to nonlocal labels and procedure exits. On the other hand, it also leads to being able to keep data items such as contours for as long as they are pointed at. Thus we call the deallocation scheme retention. It has a few consequences that are worth noting.

For example, assignment of a label value is a simple matter of copying both the ip and the ep of the label value into the target cell. By virtue of the ep pointing to the innermost contour of the label's environment (and thus by a chain of static links to all contours of the environment), we know that the contours of the environment will be retained for as long as the label value itself is around, even if the blocks corresponding to the retained contours are exited. Clearly, the same can be said for procedure values.

In the example, there is a non-trivial use of retention to insure that a task's accessing environment remains intact. In snapshots 8-9, one processor has left the outer block while the other processor is still executing and using cells from the outer contour. Because of retention, the outer contour is retained on behalf of the second processor. For further discussion on retention see [Joh 71, Bry 71a,b,c, Weg 71a].

Retention

By virtue of the fact that storage once initialized is never erased, we have retention. In particular, the three points noted about retention in the section on retention in CM carry over to CR.

Because the entire stack-of-texts has been stored with the label value, it is always possible to restore the label's entire "accessing environment" even if the current stack has a completely different set of modified programs.

When the text for a procedure value is created, all nonlocals in the body have already been subscripted. Since storage is never thrown out, no matter when and where the procedure is called, we always have enough information to use the current value of the proper location for its nonlocals, even if the stack in which the procedure was declared is long since gone.

Since one task exiting a block (even the outer one) has no effect on storage, as long as there is an awake task stack to use storage, it will be used. In snapshots 8-9 we see an occurrence of this phenomenon.

Missing details

In the above sections, many details of the behavior of CM and CR have been left out in the interest of economy and because they do not come to the heart of the differences between the two models. These include the obvious details of expression evaluation and error handling. We dispense with these with the following assumption.

Assumption 2. The handling of all missing details in the description of CM and CR is assumed to be such that the to-be-proved equivalence of CM and CR is preserved.

7. Equivalence of NDISMs

By now, the two models should look suspiciously "equivalent". We must set up the mechanism for proving this. We will first develop and justify a reasonable definition of equivalence. The definition we finally settle on also suggests a proof technique. This technique will be used in showing that the two models are indeed equivalent.

In giving a definition of equivalence of two NDISMs, several properties of multi-tasking models must be taken into account.

1) Because of nondeterminism in the transformations, there are many computations of a given program.

2) Each of these many computations is to be considered a valid computation.

3) Each of these, if they halt, may produce different output.

A possible definition of equivalence which captures these three points is the well-known "output equivalence" extended to handle nondeterministic computations. This definition requires that for each program in the language, for each computation of that program in one model, we can find a corresponding computation in the other model of the same program (and vice versa) such that the corresponding computations either both halt with the same output or both do not halt. Stated more formally:

Definition 6. Let $M=(I,I_0,F)$ be an NDISM for a programming language L. Let $M'=(I',I_0',F')$ be an NDISM for a programming language L. Then M is output equivalent to M' if and only if for all $p\epsilon L$, for all $\delta\epsilon$INPUT, $S_0=\text{compile}_M(p,\delta)$ and $S_0'=\text{compile}_{M'}(p,\delta)$,

1) For all $C\epsilon M(S_0)$, there exists $C'\epsilon M'(S_0')$ such that

a) C halts if and only if C' halts;

b) if C halts then $\text{output}_M(\text{final}(C))= \text{putput}_{M'}(\text{final}(C'))$ and

2) For all $C'\epsilon M'(S_0')$, there exists $C\epsilon M(S_0)$ such that

a) C' halts if and only if C halts;

b) if C halts then $\text{output}_{M'}(\text{final}(C')) = \text{output}_M(\text{final}(C))$■

It is clear that output equivalence is reflexive, symmetric and transitive. Therefore we state without proof:

THEOREM 2. Output equivalence is an equivalence relation■

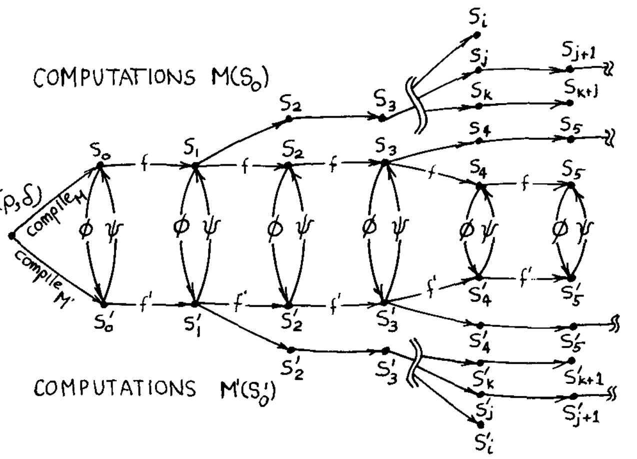
Taking output equivalence as our definition suffers on two counts. The first is that the definition given does not give much of a clue as to how a proof of equivalence should proceed. Indeed, a proof in the one step that the definition does suggest would be very difficult. We need a way of breaking the problem into smaller pieces. A standard way of doing this is to consider the intermediary steps of the computations to show what we wish.

The second defect, as noted by J.S. Hutchison [Hut 71], concerns the fact that some of the interesting multi-tasking computations, such as operating systems, are continuing computations which produce meaningful output as they execute but which do not halt! (Of course, systems crash or are closed down for maintenance, but the systems themselves are not supposed to halt.) If output equivalence is used as the criterion for equivalence, then two operating systems, even if their outputs are totally different and they use completely different scheduling policies, are equivalent simply because both do not halt. This cannot be! The approach of considering snapshots at each intermediary step during the computations and checking that their accumulated outputs are equal and that they are doing the "same" thing at each step seems a more suitable approach.

Fortunately, there is the notion of operational equivalence [Weg 72], which considers snapshots at each step of the computations, compares output, and uses induction on the steps to show the desired properties. We, in fact, will use a generalization of the McGowan mapping technique [McG 70, 71a,b, 72a,b, MW 71]. It turns out that the technique provides both 1) the tractable proof technique which, as we show, can be used to establish output equivalence (if that is what we want) and 2) the more suitable operational equivalence which we shall take as our definition of equivalence.

Schematically, we represent the set of computations in one model arising from a program p in L and an input δ in INPUT by a tree (the top half of the figure below). The sequence of snapshots lying in one directed path from the pair (p,δ) is one computation.

Suppose we have two sets of computations of the same program in two different models that we wish to



COMPUTATIONS M(S0)
COMPUTATIONS M'(S0')
(p,δ)
compile M
compile M'
S0 S1 S2 S3 S4 S5 Si Sj Sj+1 Sk Sk+1
S0' S1' S2' S3' S4' S5' Si' Sj' Sj+1' Sk' Sk+1'
f
f'
φ ψ

prove equivalent. They may be pictured as two trees sprouting from the same root. (Lest we clutter the diagram we label only a few horizontal lines with the transformation functions f and f' and show ϕ and ψ between one corresponding pair of computations.)

We take a hint from the side-by-side treatment of sections 5 and 6 and from the diagram above. Let the two NDISMs in consideration be $M=(I,I_0,F)$ and $M'=(I',I_0',F')$ and let ϕ and ψ be mappings from snapshots of one model to (corresponding) snapshots of the other; in particular, $\phi:I\to I'$ and $\psi:I'\to I$. Suppose that for each given computation $C=\langle S_0,S_1,\ldots,S_i,S_{i+1},\ldots\rangle$ in M we construct the sequence of snapshots of M', $\langle S_0',S_1',\ldots,S_i',S_{i+1}',\ldots\rangle$ such that for each i, $S_i'=\phi(S_i)$. (NB: we have not yet called this sequence a computation.) Suppose we can show

1) that this sequence is a bona-fide computation in M'. (We show this inductively, i.e. by showing

a) S_0' is an initial snapshot

b) each S_{i+1}' is obtainable by some transformation f' from S_i' if S_i is not the final snapshot of C

c) the last snapshot S_n' of the sequence (if it exists) is a genuine final - intransformable - snapshot); and

2) that the output of S_i is equal to the output of S_i' for all S_i in C.

If we can do this for all computations C in M and if we can do likewise for all computations C' in M' using ψ, then we might say that M and M' are operationally equivalent.*

This suggested definition is not enough. Suppose we wish to prove that CM, in which all inaccessible contours and processors are deallocated immediately, is operationally equivalent to a version of the contour model CM' in which no contours and processors, once allocated, are deallocated. (We assume sufficient memory so that running out of free storage does not become an issue.) We expect CM and CM' to be operationally equivalent because of the following. The only cells which can affect a computation are those that are accessible to some awake processor. Only inaccessible cells are ever deallocated. Therefore the presence or absence of inaccessible cells should make no difference to the net effect of the computation. However, this cannot be proved using the suggested technique. It is easy to get a mapping from snapshots of the non-deallocating model to

*Properly speaking, this form of operational equivalence should be called "one-to-one operational equivalence" (see item 2 of section 11). Since this definition is the only one used in this paper, we continue to call it simply "operational equivalence".

snapshots of the deallocating model. The mapping function merely throws out all inaccessible contours and processors*. However, the mapping in the other direction is a bit difficult because it is impossible to reconstruct already deallocated contours and processors.

Suppose we were allowed to look at the entire history of a computation up to the present snapshot when constructing the mapping from the deallocating model to the non-deallocating model. All of the contours and processors that we would need for constructing the snapshot in the non-deallocating model are in some snapshot between the initial and the present snapshots of the deallocating model. This suggests a definition of operational equivalence which is more general than the suggested one. The two mappings ϕ and ψ will be on sequences of snapshots of one model to a single snapshot of the other. In particular,

$$\phi(S_0,S_1,\ldots,S_i) = S'_i \text{ , and}$$
$$\psi(S'_0,S'_1,\ldots,S'_i) = S_i.$$

We incorporate this into our definition of operational equivalence which we now state formally.

Definition 7. Let $M=(I,I_0,F)$ be an NDISM for a programming language L. Let $M'=(I',I'_0,F')$ be an NDISM for a programming language L. Then M is operationally equivalent to M' if and only if there exist mappings**

$$\phi\colon I^+ \to I'$$
$$\psi\colon I'^+ \to I$$

such that for all $p \epsilon L$, $\delta \epsilon$INPUT,

1) If $S_0=\text{compile}_M(p,\delta)$, for all $C=<S_0,\ldots,S_i,S_{i+1},\ldots>\epsilon M(S_0)$, if for all i such that S_i is in C $S'_i=\phi(S_0,\ldots,S_i)$, then
   a) $S'_0=\text{compile}_{M'}(p,\delta)$.
   b) if $S_i \neq \text{final}(C)$ then $S'_{i+1}\epsilon f'(S'_i)$ for some $f \epsilon F'$.
   c) if $S_n=\text{final}(C)$ then S'_n is a final snapshot in M'.
   d) for all i such that S_i is in C, $\text{output}_M(S_i)=\text{output}_{M'}(S'_i)$.

2) If $S'_0=\text{compile}_{M'}(p,\delta)$, for all $C=<S'_0,\ldots,S'_i,S'_{i+1},\ldots>\epsilon M'(S'_0)$, if for all i such that S'_i is in C' $S_i=\psi(S'_0,\ldots,S'_i)$, then
   a) $S_0=\text{compile}_M(p)$.
   b) if $S'_i \neq \text{final}(C)$ then $S_{i+1}\epsilon f(S_i)$ for some $f \epsilon F$.
   c) if $S'_n=\text{final}(C')$ then S_n is a final snapshot in M.
   d) for all i such that S'_i is in C', $\text{output}_{M'}(S'_i)=\text{output}_M(S_i)$.

It is clear that using $S'_i=\phi(S_i)$ and $S_i=\psi(S'_i)$, as in the suggested technique, is a special case of definition 7.

As an added bonus, it is easy to show that if two NDISMs M and M' are operationally equivalent then they are also output-equivalent. We state without proof:

THEOREM 3. Let $M=(I,I_0,F)$ be an NDISM for a programming language L. Let $M'=(I',I'_0,F')$ be an NDISM for a programming language L.

M is operationally equivalent to M'
implies that
M is output-equivalent to M'■

In order to set up chains of equivalences, it would be nice if operational equivalence were an equivalence relation. The following theorem shows that it is.

THEOREM 4. Operational equivalence is an equivalence relation.

Proof. Let $M=(I,I_0,F)$, $M'=(I',I'_0,F')$ and $M''=(I'',I''_0,F'')$ be NDISMs for a programming language L. We must show

A) M is operationally equivalent to M.
B) M is operationally equivalent to M' implies that M' is operationally equivalent to M.
C) M is operationally equivalent to M' and M' is operationally equivalent to M" imply that M is operationally equivalent to M".

Proof of A: Trivial. Let ϕ and ψ be the identity mappings.

Proof of B: Trivial. Since M is operationally equivalent to M' there exist $\phi\colon I^+ \to I'$ and $\psi\colon I'^+ \to I$ which satisfy the required properties. Simply switch the role of ϕ and ψ in the proof.

Proof of C: Since M is operationally equivalent to M' there exist mappings $\phi'\colon I^+ \to I'$ and $\psi'\colon I'^+ \to I$ satisfying conditions 1 and 2 of definition 7.

Since M' is operationally equivalent to M" there exist mappings $\phi''\colon I'^+ \to I''$ and $\psi''\colon I''^+ \to I'$ satisfying conditions 1 and 2 of definition 7.

Let $\phi\colon I^+ \to I''$ and $\psi\colon I''^+ \to I$ be such that $\phi(S_0,\ldots,S_i)=\phi''(\phi'(S_0),\ \phi'(S_0,S_1),\ldots,\phi'(S_0,\ldots,S_i)$ and

$$\psi(S''_0,\ldots,S''_i)=\psi'(\psi''(S''_0),\ \psi''(S''_0,S''_1),\ldots,\psi''(S''_0,\ldots,S''_i)).$$

We must show that conditions 1 and 2 of definition 7 are satisfied by ϕ and ψ. We show here that condition 1 is satisfied. Suppose that $p\epsilon L$ and $\delta\epsilon$INPUT. Let $S_0=\text{compile}_M(p,\delta)$ and let $C=<S_0,\ldots,S_i,\ldots>\epsilon M(S_0)$. Let $S'_i=\phi'(S_0,\ldots,S_i)$ for all i such that S_i is in C. By properties of ϕ' we know that these $<S'_0,\ldots,S'_i,\ldots>$ must form a computation C' of S'_0 in M'. Let $S''_i=\phi(S_0,\ldots,S_i)=\phi''(S'_0,\ldots,S'_i)=\phi''(\phi'(S_0),\ldots,\phi'(S_0,\ldots,S_i))$ for all i such that S_i is in C. We must show that each of $S''_0,\ldots,S''_i,\ldots$ so constructed satisfies parts a,b,c and d of condition 1.

Proof of a: Show that $S''_0=\text{compile}_{M''}(p,\delta)$. This follows immediately from the fact that ϕ'' satisfies (1a), i.e. $S''_0=\text{compile}_M(p,\delta)$.

Proof of b: Show that if $S_i\neq\text{final}(C)$ then $S''_{i+1}\epsilon f''(S''_i)$ for some $f''\epsilon F''$. If $S_i\neq\text{final}(C)$ we know by property (1b) of ϕ' that $S'_{i+1}\epsilon f'(S'_i)$ for some $f'\epsilon F'$. But then $S'_i\neq\text{final}(C')$ since S'_i has a successor. Therefore, by property (1b) of ϕ'', $S'_i=\text{final}(C')$ implies that $S''_{i+1}\epsilon f''(S''_i)$ for some $f''\epsilon F''$.

Proof of c: Show that if $S_n=\text{final}(C)$ then S''_n is a final snapshot. If $S_n=\text{final}(C)$ we know by property (1c) of ϕ' that S'_n is a final snapshot in M'. If we can show that $S'_n=\text{final}(C')$ then by property (1c) of ϕ'' we have what we want, i.e. S''_n is a final snapshot. We know S'_n is a final snapshot; we must show it is the final snapshot of C'. This follows directly from theorem 1 and definition 5.

Proof of d: Show that for all i such that S_i is in C, $\text{output}_M(S_i)=\text{output}_{M''}(S''_i)$. By property (1d) of ϕ' and ϕ'', $\text{output}_M(S_i)=\text{output}_{M'}(S'_i)$ for all i such that S_i is in C and $\text{output}_{M'}(S'_i)=\text{output}_{M''}(S''_i)$ for all i such that S'_i is in C', which by the preamble of the proof of C, is all i such that S_i is in C. Transitivity of equality gives us what we want.

The proof of condition 2 of ψ follows reasoning similar to that of the proof of condition 1■

8. Operational equivalence of CM and CR

We now get down to the business of proving that CM is operationally equivalent to CR. We first define new models $\mathcal{CM}$ and $\mathcal{CR}$ which are modifications of CM and CR respectively. These two new models are such that they are equivalent to the models of which they are modifications. In addition, $\mathcal{CM}$ and $\mathcal{CR}$ are more easily proved operationally equivalent to

*It is interesting that the mapping here is the garbage collect function.

**Notation: $I^1=I$, $I^n=I\times I^{n+1}$, and $I^+=\bigcup_{i=1}^{\infty} I^i$.

each other than are CM and CR.

Additions to CM

We make the following changes in the snapshots and transformation function of CM to produce the model $\mathcal{CM}$

1) We add to each contour a unique contour number in its upper right-hand corner.

2) We add to the snapshots a block entry count generator, BECG. In the initial snapshot the BECG reads 1. At each block or procedure entry the current value of the BECG becomes the contour number of the contour allocated for the entry. Then the BECG is incremented by 1.

3) We add to each processor a unique processor number.

4) We add to the snapshot a task name generator, TNG. In the initial snapshot the TNG reads 2 and the initial processor is numbered 1. Whenever a new processor is created, it is numbered with the current value of the TNG and then the TNG is incremented by 1.

5) To each contour we add an antecedent link which points to the begin or procedure delimiter of the block or procedure of the algorithm whose entry resulted in the allocation of that contour. The identifiers in the declaration array of a contour are precisely those which are declared (explicitly or implicitly) in the block or procedure to which its antecedent link points.

6) We add to the transformation the stipulation that no contours and processors are deallocated. This results in a snapshot containing every contour and processor that was ever allocated.

We use the technique of definition 7 to show that CM is operationally equivalent to $\mathcal{CM}$.

LEMMA 1. CM is operationally equivalent to $\mathcal{CM}$

Proof: Let CM=(I,I_0,F) and $\mathcal{CM}=(\mathcal{I},\mathcal{I}_0,\mathcal{F})$. We must produce mappings

$$\phi: I^+ \to \mathcal{I}$$
$$\psi: \mathcal{I}^+ \to I$$

which satisfy the conditions of definition 7.

Let $\phi(S_0,\ldots,S_i)=\mathcal{I}_i$, where $\mathcal{I}_i$ is obtained from S_0 through S_i by the following steps:

1) Find all contours and number them: Find the latest (in the history of the computation) of each contour that has ever appeared during the computation. These are the contours that appear in $\mathcal{I}_i$. By tracing through the history of the computation (at any given time in a computation there is only a finite number of snapshots in the computation thus far), it is possible to determine the order in which the contours in $\mathcal{I}_i$ were allocated. Number the contours in $\mathcal{I}_i$ in that order.

2) Add antecedent links: For each contour C, by tracing the history of the computation from S_0 to S_i, it is also possible to determine the first snapshot S_j in which the contour C appears. By examining the snapshot before and seeing which processor changed during the transformation to S_j, it can be determined which block entry or procedure call was executed. Merely look at the ip of the changing processor in S_{j-1}. (Note that since a contour is allocated during transformation, the statement executed is necessarily a block entry or procedure call possibly with a task option and not, say, a goto to the same statement which is hard to detect.) The antecedent link of contour C is a pointer to the begin or procedure delimiter of the entered block or procedure body.

3) Find all processors and number them: Similarly, find the latest copy of each processor in the computation and number each of them in order of creation. These are the processors of $\mathcal{I}_i$.

4) Compute the values of the BECG and the TNG: the BECG of $\mathcal{I}_i$ is set to 1 more than the number of contours in $\mathcal{I}_i$ and the TNG of $\mathcal{I}_i$ is set to 1 more than the number of processors in $\mathcal{I}_i$.

5) Copy over everything else from snapshot S_i.

Let $\psi(\mathcal{I}_0,\ldots,\mathcal{I}_i)=\mathcal{I}_i$ with all inaccessible contours and processors, and all contour numbers, antecedent links, processor numbers, the BECG and the TNG removed. (Note that $\mathcal{I}_0,\ldots,\mathcal{I}_{i-1}$ are ignored in this direction.)

It is clear that ϕ and ψ meet conditions 1 and 2 of theorem 7 simply because both CM and $\mathcal{CM}$ are using essentially the same transformation. Therefore we conclude that CM is operationally equivalent to $\mathcal{CM}$.

Additions to CR

We make the following changes to the snapshots and transformation function of CR to produce the model $\mathcal{CR}$.

1) Add σ and α to the list of identifiers labelling the columns of the storage.

2) When row i of the storage is initialized (marked "decl") as a result of a block entry, the location (σ,i) is set to the delimiter subscript of the block or procedure in which the entered block is immediately nested (to form a static link). The location (α,i) is set to point to the begin of the block just entered (to form an antecedent link).

3) When row i of the storage is initialized as a result of a procedure call, say of p_j, the text which is in the value of p_j is examined to obtain the subscript k of the begin or procedure delimiter which immediately surrounds the procedure body. The location (σ,i) is set to k. Then the location (α,i) is set to a copy of the ip of the value of p_j so that it points to the called body.

We now show that CR is operationally equivalent to $\mathcal{CR}$.

LEMMA 2. CR is operationally equivalent to $\mathcal{CR}$

Proof: Let CR=(I,I_0,F) and $\mathcal{CR}=(\mathcal{I},\mathcal{I}_0,\mathcal{F})$. We must produce mappings

$$\phi: I^+ \to \mathcal{I}$$
$$\psi: \mathcal{I}^+ \to I$$

which satisfy the conditions 1 and 2 of definition 7.

Let $\phi(S_0,\ldots,S_i)=\mathcal{I}_i$, where $\mathcal{I}_i$ is formed from S_0 through S_i by the following steps:

Add σ and α to identifiers labelling columns of the storage and compute (σ,i) and (α,i) for each row i of the storage:

Take S_i and add α and σ to the list of identifiers labelling the columns of the storage. For each j of the storage, by tracing the history of the computation from S_0 to S_i, locate the snapshot in which row j was first initialized. By looking at the previous snapshot and determining which task stack changed in the transformation it can be determined which task stack executed which instruction (which is necessarily a block or procedure entry). Construct (σ,j) and (α,j) according to the rules given for block and procedure entry in the definition of $\mathcal{CR}$.

Let $\psi(\mathcal{I}_0,\ldots,\mathcal{I}_i)=S_i$, where S_i is obtained by erasing σ and α from the column labels of the storage and erasing all values of (σ,j) and (α,j) for all j, $1<j<\text{becg}-1$, where becg is the value of the BECG of $\mathcal{I}_i$.

It is clear that ϕ and ψ satisfy the conditions 1 and 2 of definition 7, since the transformations of CR and $\mathcal{CR}$ do essentially the same things. Thus we conclude that CR is operationally equivalent to $\mathcal{CR}$.

Major lemma

We are now in a position to show that $\mathcal{CM}$ is equivalent to $\mathcal{CR}$. To get an intuitive feeling for the mappings and how they work, we consider snapshots S_5' and S_5 from the execution of the example program in $\mathcal{CM}$ and in $\mathcal{CR}$ respectively*. These are the last pair of snapshots in sections 5 and 6.

There are a number of invariances between the two snapshots which form the basis of the mapping functions.

1) There is the same number of task stacks in S_5 as there are processors in S_5'. Furthermore, the task stacks and processors are numbered the same.

2) The bottom text on the task stack in S_5 is precisely the same as the algorithm of S_5'.

3) The ip of the task stack in S_5 points to the "same" instruction that the ip of the processor points to in S_5'. The same holds for ip's of label and procedure values in both snapshots.

4) The BECGs and the TNGs of S_5 and S_5' are respectively the same.

5) There are as many rows of the storage in S_5 which are initialized or are marked declared as there are contours in S_5'.

6) Row i of the storage in S_5 has the same identifiers initialized or marked declared as the ith contour in S_5' has in its declaration array.

7) There is the same number of contours in S_5' on the return path of the processor as there are texts (except for the bottom) on the task stack of S_5. (The return path is via the static link for contours whose antecedent link points to a block and via the ep of $\mathfrak{R}$ for contours whose antecedent link points to a procedure.) The same may be said of label values.

8) The ip of the task stack in S_5 points to a statement S in the top text of the stack. This statement S is nested inside some set of nested modified blocks and procedures. The numbers which subscript the begin's and procedure's of these blocks and procedures are the same as the numbers of the contours which are in the accessing environment of the processor in S_5'. In particular, the ep of the processor in S_5 points to the contour numbered the same as the innermost block's <u>begin</u>. The same can be said for ep's of the label <u>and</u> procedure values in S_5' as opposed to the top text and text of label and procedure values in S_5.

9) The ip stored in (α,i) points to the same statement in the program as the antecedent link of contour i.

10) The number stored in (σ,i) is the same as the number of the contour pointed to by the static link of contour i.

<u>LEMMA 3.</u> $\mathcal{CM}$ is operationally equivalent to $\mathcal{CR}$.

<u>Proof:</u> We exhibit mappings

$$\phi\colon I_{\mathcal{CR}} \to I'_{\mathcal{CM}}$$
$$\psi\colon I'_{\mathcal{CM}} \to I_{\mathcal{CR}}$$

which satisfy the conditions of the special case of definition 7.

First, we describe ϕ which maps a snapshot in $\mathcal{CR}$ to the corresponding snapshot in $\mathcal{CM}$. (We use snapshots S_5 and S_5' as occasional examples.)

Let S be a snapshot in some computation in $\mathcal{CR}$ $(\text{compile}_{\mathcal{CR}}(p,\delta))$ for some $p\varepsilon$GBSL and some $\delta\varepsilon$INPUT. The snapshot $S'=\phi(S)$ is constructed as follows.

A) The algorithm of S' is a copy of the bottom text on task stack tn_1 (any task stack will do).

B) We form the contours of S' from the storage component of S as follows:

From the bottom-of-stack text of task stack tn_1 we get the program p. We form the finite set ID= {id|id is an identifier and id is used in p}$\cup\{\mathfrak{R}\}$. For each i, $1\le i\le becg-1$, where becg is the value of the BECG in S, do the following four steps:

1) Create contour i with one cell in its declaration array labelled id for each idεID, such that (id,i) has a value or is marked "decl". (For example, (a,1) and (p,1) are the only locations which are initialized in row 1 of the storage of S_5; therefore contour 1 of S_5' has cells for a and p in its declaration array.)

2)** The static link of contour i points to the contour whose number is the value of (σ,i). A reference to contour 0 is considered a null static link.

3) The antecedent link of contour i points to line j of the algorithm where j is the value stored in (α,i).

4) Store into each cell id of the contour i the value converted from the contents of (id,i) according to the following data type conversion rules:

a) integer, real, character and boolean values are copied as is.

b) pointer values (id',j') are converted into pointers pointing to the cell id of contour j.

c) label values consisting of an ip and a stack s are converted to ip' and ep' by the following: take ip' as a copy of ip. In the text on top of the stack s, the ip points either to the begin of the outer block or to some statement immediately nested inside some modified block or procedure whose begin or procedure delimiter is numbered k. In the first case ep' is taken as nil. In the second case ep' is set to point at contour k.

d) procedure values consisting of an ip and a text t are converted to ip' and ep' by the following: take ip' as a copy of ip. In the text t, the ip points to a procedure body immediately nested inside a modified block or procedure whose begin or procedure delimiter is numbered k. Take ep' to point to contour k.

***e) task values consisting of a task name tn_j are converted into pointers to processor number j.

***f) semaphore values consisting of an integer v in the value field and a task name list $<tn_{i_1}, tn_{i_2},\ldots,tn_{i_n}>$ in the queue field are converted into an integer value field v' and a header of a doubly linked list h' as follows: The integer v' is a copy of v. The queue head h' is made the head of a doubly linked list of the processors numbered i_1, ... through i_n. The processors are linked in the order given in the list $<tn_{i_1},\ldots,tn_{i_n}>$ with processor i_1 first and processor i_n last.

g) "decl" marks are converted into blank cells.

C) We form the processors of S' from the task stacks of S. For each task stack tn_i, $1\le i\le tng-1$ where tng is the value of the TNG of S, do the following two steps:

1) Create processor i with the same **state** as task stack tn_i.

2) The ip and stack of tn_i are used to form the ip' and ep' of processor i by the same conversion as described in B4c) for label values.

*We shall follow the notational convention of priming (') $\mathcal{CM}$-related items and not priming $\mathcal{CR}$-related items.

**This is done at this time for exposition purposes. Strictly speaking, since all contours have not yet been created, this step should be done in a separate pass over the rows.

***Strictly speaking, e) and f) should be done after section C below.

D) The BECG of S' is a copy of the BECG of S.
E) The TNG of S' is a copy of the TNG of S.
F) The input list, input pointer, output list and output pointer of S' are copies of those of S.

Secondly, we describe ψ which maps a snapshot in $\mathcal{CM}$ to the corresponding snapshot in $\mathcal{CR}$.

Let S' be a snapshot in some computation in $\mathcal{CM}$(compile$_{\mathcal{CM}}$(p,δ)) for some p∈GBSL and some δ∈INPUT. The snapshot S'=ψ(S) is constructed as follows:

A) We form the task stacks of S from the processors of S'. For each processor numbered i, $1 \le i \le$ tng-1 where tng is the value of the TNG of S', do the following five steps. (First we note that the task stack tn_i is to consist of modified texts of the program p that reflect each activation of a block or procedure that has not yet been exited by the task tn_i. The first step is to form the dynamic chain list by starting with the contour pointed to by processor i's ep and tracing down the chain of returns.)

1) Let C be the contour pointed to by the ep of processor i.

a) add the contour number of C to the end of the list.

b) find the contour C_r which will become the processor's immediate environment if the processor left C by a normal block or procedure exit. If C is the outer contour (C has a null static link) then go to 2 (below). If C is that of a non-outer block entry (no $\aleph$ cell in contour C and a non-null static link) C_r is the contour pointed to by the static link of C. If C is that of a plain procedure call (an $\aleph$ cell with a label value in contour C), C_r is the contour pointed to by the ep of the label in the cell for $\aleph$ in C. If C is that of a tasked procedure call (an $\aleph$ cell with a nill value in contour C) then go to 2 below.

c) Let C be C_r and go to 1a) above.

2) Reverse the order of the dynamic chain list, creating list $D=<D_1,\dots,D_n>$. (In S'_5 the dynamic chain list of processor 1 is <2,1> and the reversed list is <1,2>.)

3) Now create the stack of task stack tn_i from the bottom up. The bottom of the stack, $Stack_0$, is a copy of the algorithm of S'. Now for j, $1 \le j \le n$ (the length of D), do the following two steps:

a) Let b be the block or procedure whose entry caused creation of contour k, where k is the value of D_j. This is obtained from the antecedent link of contour k.

b) $Stack_j$ is obtained by taking the unmodified program and subscripting with k the delimiter of b and each identifying occurrence of each identifier **declared** in b. Then we follow the static chain from contour k to obtain the subscripts for the blocks and procedures surrounding b. In particular, for each (finitely many) successively surrounding contour n of k, subscript with n the corresponding delimiter (obtained by the antecedent link) and each identifying occurrence of each identifier declared in the corresponding block or procedure.

B) We form the storage component of S by using the contours of S'. For i, $1 \le i \le$ becg-1, where becg is the value of the BECG in S, do the following:

1) For each cell labelled id in contour i, its contents are converted for storing into (id,i) of the storage for S by the following data-type-dependent rules:

a) integer, real, character, and boolean values are copied as is.

b) a pointer pointing to the cell id' in contour j' is converted to the pair (id',j').

c) label values consisting of ip' and ep' are converted into an ip and a stack of texts; the ip is a copy of ip'. Form the stack by following steps A1, 2 and 3 above with C initially as the contour pointed to by ep'.

d) procedure values consisting of ip' and ep' are converted into an **ip** and a modified text: take ip as a copy of ip'. Form a stack of texts by following steps A1,2 and 3 above with C initially as the contour pointed to by ep. Then take the top text on this stack as the text of the procedure value.

e) a task value pointing to processor i is converted into task name tn_i.

f) a semaphore consisting of the integer V' in its value field and the head h' of a doubly linked list of processors is converted to the value field V and task name list ℓ: The integer V is a copy of V'. From the doubly linked list of processors headed by h' form a list $<i_1,\dots,i_n>$ of the processor numbers of these processors. The list $<i_1,\dots,i_n>$ is ordered the same as the doubly linked list of processors. Form the list ℓ as $<tn_{i_1},\dots,tn_{i_n}>$.

g) blank cells are converted to "decl" marks.

2) (σ,i) is taken to be the number of the contour pointed to by the static link of contour i. (α,i) is taken to be a copy of the ip which is the antecedent link of contour i.

C) The BECG of S is a copy of the BECG of S'.
D) The TNG of S is a copy of the TNG of S'.
E) The input list, input pointer, output list and output pointer of S are copies of those of S'.

Thirdly, we must show that ϕ and ψ satisfy the conditions of theorem 7. We show only that ϕ satisfies condition 1. The proof that ψ satisfies condition 2 follows the same lines.

Let p∈GBSL and δ∈INPUT, and let S_0=compile$_{\mathcal{CR}}$(p,δ). Furthermore, let $C=<S_0,S_1,\dots,S_i,S_{i+1},\dots>\in\mathcal{CR}(S_0)$. Let $S'_0=\phi(S_0),\dots,S'_i=\phi(S_i),\dots$. Then it suffices to show that subconditions a,b,c and d hold.

<u>Proof of a</u>: Clearly S'_0=compile$_{\mathcal{CM}}$(p,δ) for the initial snapshot in $\mathcal{CR}$ consists of:
- A single task stack tn_1 with
  - a single unmodified text, p,
  - the ip pointing to line 1 of this text
- An empty storage component
- An input list δ with the input pointer pointing to the first item on this list

This maps under ϕ to:
- A single processor numbered 1 with
  - an ip pointing to line 1 of the algorithm
  - and a null ep
- The program p as the algorithm
- An input list δ with the input pointer pointing to the first item on this list.

This is exactly the initial snapshot produced by compile$_{\mathcal{CM}}$(p,δ).

<u>Proof of b</u>: Suppose that $S_i\ne$final(C) and thus, in fact, $S_{i+1}\in f(S_i)$ where f is the transformation function of $\mathcal{CR}$. We have that $S'_i=\phi(S_i)$ and that $S'_{i+1}=\phi(S_{i+1})$. It suffices to produce a transformation f' of $\mathcal{CM}$ such that $S'_{i+1}\in f'(S'_i)$; **i.e., to show that** S'_{i+1} **could be** a valid successor snapshot to S'_i. Since there is only one transformation to choose from, we must merely show that S'_{i+1} is <u>one</u> of the results of applying the transformation f' to S'_i.

We shall consider the five steps of the transformation in $\mathcal{CR}$ at the gross level and show that they are mimicked by the five steps of the transformation of $\mathcal{CM}$.

Since S_i is not final, the following steps were done to obtain S_{i+1}.

An awake task stack tn_j was selected; the statement s pointed to by its ip was picked up from the top text; its ip was advanced to point to the next statement, ns; and the statement s was executed. This leaves the ip of tn_j pointing to some statement ss which, if s was not a goto or procedure call, is ns; otherwise it is the target of the goto or call.

By the way ψ works, we know the following: In S'_i the processor numbered j is awake and its ip points to the statement s' which is an unmodified (unsubscripted) version of s. In S'_{i+1}, the ip of processor j points to ss' which is the unmodified version of ss.

It is clear that the following instance of the transformation of f' would transform S'_i to S'_{i+1}:

Select awake processor j, pick up the statement s' which is pointed to by processor j's ip, advance the ip to point to the next statement, ns', and execute s', leaving the ip pointing to ss'.

We must verify that in S'_{i+1} the ip of processor j points to an unmodified version of the statement that the ip of tn_j points to in S_{i+1}. If s was not a goto or call and thus in S_{i+1} tn_j's ip points to ns, the successor of s, then s' is not a goto or call and thus processor j's ip points to the successor of s' which by ϕ must be ns'. If s was a goto or call so was s', and we leave showing that the resulting ip's point to the "same" statement to the section on gotos and calls.

It remains now to show for each statement type that the execution of s' mimics the execution of s.

I. Identifier accessing

One common element to almost all statements is identifier accessing. First we show that if id_n appears in the statement s being executed by task stack tn_j in S_i then 1) id appears in the statement s' being executed by processor j in S'_i and 2) the cell for id in the environment of processor j is in contour n. The first follows directly from the property of the mapping.

To show the second we first show that if the list or delimiter subscripts in outward order of the blocks and procedures that surround s is $<n_1,n_2,\ldots,n_m>$ then the list of the numbers of the contours of the environment of processor j in outward order is $<n_1,n_2,\ldots,n_m>$. By the mapping, we know that for any row k of storage in S_i the contents of (σ,k) is equal to the number of the contour pointed to by the static link of contour k. Therefore, it suffices to show that for any block or procedure on the top of the task stack, if its delimiter subscript is k, then the delimiter subscript of the immediately enclosing block or procedure is equal to the contents of (σ,k). This follows immediately from the construction of (σ,k). To get back to the problem at hand, namely proving (2), we know by the mapping that there is a cell for id in contour n. We simply must show that there are no other contours inside contour n and outside the processor j which contain a cell for id. Suppose there was one, say in contour m. Then by the fact that the list of delimiter numbers is equal to the list of contour numbers in the environment, there must be a block or procedure enclosing S whose delimiter is subscripted by m and in which id is declared. (If id were not declared in this block or procedure m then there would be no cell for id in contour m.) This means that there is a declaration of id closer in the text to s than the one in block or procedure n, namely in block or procedure m. Therefore the subscript on id in S must be m. We have a contradiction. Thus, the innermost contour with a cell for id is contour n. From this argument we can conclude the following important subresult.

II. Values which are obtained or assigned as a result of identifier accessing

Since the mapping ϕ constructs the value v' stored in the id cell of contour n by using the value v in location (id,n), it follows that if an identifier is used in statement s then the value which is obtained from the storage in S_i and/or is assigned to storage in S_{i+1} maps under ϕ the value which is obtained from the contours in S'_i and/or is assigned to the contours in S'_{i+1} as a result of the use of an unsubscripted version of the identifier in statement s'.

III. Block entry

In S_i and S'_i the BECGs are equal; say it has the value m. Then in $\mathcal{CR}$, the block or procedure is entered and a new text is pushed onto the stack with the begin delimiter and the identifying occurrences of the declared identifier subscripted by m. This maps under ϕ to the processor's ep pointing to a new contour m and the declared identifiers being in contour m. This is exactly what would happen upon block or procedure entry in $\mathcal{CM}$ with the BECG equal to m.

IV. Block exit

We know that the number of the begin of the block being exited is the same as the number of the contour being exited. Suppose this number is m. In $\mathcal{CR}$, the top text on the task stack is popped. The task stack is now executing inside a block or procedure whose delimiter is subscripted by the value of $(\sigma,m) = \sigma m$. This maps under ϕ to the processor's ep pointing to contour σm which is precisely the effect in $\mathcal{CM}$ of exiting a block when the ep points to the contour m.

V. Procedure call

In S_i and S'_i the BECGs are equal; say that they have the value m. We know that the procedure value in S_i which consists of an ip and a text, maps to the corresponding procedure value (ip',ep') in S'_i. If the call is executed in $\mathcal{CR}$ the text is pushed into the stack, and in this text the entered procedure's delimiter and formal parameters are subscripted by m. The row m of the storage is suitably initialized with parameter values and the return label, and in particular (σ,m) is set to the subscript of the begin or procedure delimiter in which the called procedure body is nested. By property of ϕ this subscript must be ep'. The ip of tn_j is set to point to the first statement of the body and the BECG is set to m+1. This maps under ϕ to a new contour numbered m with cells for formal parameters initialized to parameter values, the cell for $\mathcal{R}$ initialized to a return label (in the next section we need the fact that this return label in $\mathcal{CM}$ is the result of the mapping ϕ applied to the return label in $\mathcal{CR}$), and the static link set to ep' (which is the value of (σ,m)). The processor j's ep points to contour m and its ip points to the first statement of the procedure body. The BECG is m+1. This is precisely what would happen if processor j called the procedure (ip',ep') with the mapped version of the parameters.

VI. Procedure exit and goto's

By the properties of the mapping ϕ, argument II and, in the case of return labels, the observation made in V above, we know that the ip and stack of the label in S_i maps under ϕ to an ip' and ep' of a label in S'_i. In $\mathcal{CR}$ the goto results in replacing the ip and stack of texts of the executing task stack by that of the label. By our observation just above, this maps to the corresponding processor having ip' and ep' as its site of activity. This is precisely the site of activity that would result by the execution of a goto with a label value (ip',ep').

VII. Task creation - with label

We know that the label in S_i which consists of an ip and a stack of texts, maps under ϕ to the corresponding label in S_i' which consists of an ip' and an ep'. We also know that the TNGs of S_i and S_i' have the same value, say k. If the statement is executed in $\mathcal{CR}$, a new task stack is created named tn_k whose ip and stack of texts are a copy of that of the label. Then TNG is incremented to k+1. This maps under ϕ to a new processor numbered k whose ip' and ep' are that of the label, and to TNG being k+1. This is precisely what would happen in $\mathcal{CM}$ if the task creation were executed.

VIII. Task creation with call

We know that the procedure in S_i which consists of an ip and a text, maps to the corresponding procedure in S_i' which consists of an ip' and ep'. Furthermore, the TNGs and BECGs of S_i and S_i' are the same, say t and b respectively. If the task creation is executed in $\mathcal{CR}$, a new task stack named tn_t, whose stack of texts consists only of the original program and a suitably subscripted-by-b copy of the text of the procedure, and whose ip points to the first statement of the procedure body. Row b of the storage is suitably initialized with parameters and the TNG and BECG are incremented to t+1 and b+1 respectively. This maps under ϕ to a new processor numbered to whose ep points to a newly allocated and suitably initialized contour number b and whose ip points to the entry point of the procedure. The TNG and BECG are incremented to t+1 and b+1 respectively. This is also the effect of executing the task creation in $\mathcal{CM}$.

IX. Semaphore usage

We know that the integer value field of the selected semaphore will be the same in S_i and S_i'. Furthermore, we know that the task stack names on the list of the semaphore in S_i and S_{i+1} are the same, in order, as the numbers of the processors on the queue of the semaphore in S_i' and S_{i+1}' respectively.

Now suppose a request is done in $\mathcal{CR}$. The value field will be decremented by 1 and if it is less than 0, tn_j will be put to sleep and be added to the list; if greater than or equal to 0, nothing will happen. This maps under ϕ to the value field being decremented to the same integer value and if less than 0, the processor j being put to sleep and added to the queue or if greater than or equal to 0, nothing. This is the effect of executing request in $\mathcal{CM}$.

Suppose release is done in $\mathcal{CR}$. The value field will be incremented by 1 and if less than or equal to 0 then some task stack whose name, say tn_k, is on the list will be removed from the list and awakened; if greater than 0, nothing happens. This maps under ϕ to the value field being incremented to the same value and if less than or equal to 0, a processor, say k, being removed from the list and awakened, or if greater than 0, nothing. This is one of the possible results of executing release in $\mathcal{CM}$. Note that it is nondeterministic as to which task stack and which processor are removed from the list and awakened in their respective transformation. This makes no difference to the proof since like-numbered task stacks and processors must be on the respective lists, by virtue of ϕ. The transformation f' certainly allows selection of the processor whose number happens to be the same as the task stack name selected by f.

XI. Input/output

We dispense with this by noting 1) that ϕ copies as is the input list, the input pointer, the output list, and the output pointer, and 2) that the subtransformations for input-output are the same in $\mathcal{CM}$ and $\mathcal{CR}$.

Proof of c: If S_n is a final snapshot then $\phi(S_n)=S_n'$ is a final snapshot. This is because the states of the task stacks are copied exactly into the states of the processor. Thus if there are no awake task stacks in S_n there are no awake processors in S_n'.

Proof of d: That $\text{output}_{\mathcal{CR}}(S_i)=\text{output}_{\mathcal{CM}}(S_i')$ for all S_i in C follows immediately from the fact that ϕ copies the output list of S_i to make the output list of S_i'.

This completes the outline of the proof of (1). The proof of part (2) follows a similar exhaustive line. Therefore we conclude that $\mathcal{CR}$ is operationally equivalent to $\mathcal{CM}$∎

Major theorem

We complete the proof that CM is equivalent to CR by appeal to transitivity of operational equivalence (theorem 4).

THEOREM 5. CM is operationally equivalent to CR (also output-equivalent, by theorem 3)∎

10. Conclusions

This paper has given a formal basis for proving equivalence of models of multi-tasking languages and their implementations. The proof techniques developed herein have been applied to an informal proof of equivalence of two particular models of a multi-tasking block structured language, namely the contour model and the copy rule. The paper has also struck what the author believes to be a reasonable level of rigor for dealing with such large-scale definitions and proofs. Certainly it can be said that such proofs can be very revealing in terms of the models themselves and the reasons for their equivalence.

The operational proof techniques given here have the property that they deal only with the differences between the two models being proved equivalent. Parts of the models which are the same in both cases are dispensed with by use of the identity mapping. Thus, as observed by Wegner [Weg 71b], the complexity of an operational proof is related to the degree of difference between the two models rather than the complexity of the models themselves. In particular, proof that a model is operationally equivalent to itself is trivial.

In most practical applications, the proof techniques will be applied to models which differ by a small change for which there is a strong intuition as to why the change works. If there are several changes they can be handled by a sequence of such equivalence proofs. Thus a seemingly big job is easily cut down to manageable proportions.

The proof techniques given suggest a methodology for designing an implementation of a programming language so that the designer is sure that the implementation is correct. The methodology assumes that the definition of the programming language or some implementation of it is already given as an NDISM. Starting with one of these NDISMs, the designer should massage the form of the snapshots in the original model until he gets the desired "looking" snapshots for the new model. Then the transformation of the original model would be changed so that it in fact does the computation in the massaged version; i.e., so that it transforms the massaged version of a snapshot into the massaged version of the successor. He should also make sure that the massage is reversible, i.e. that he can do the same sort of massage from the new model to the old model. If the designer can do this, the massages become the mappings needed in the proof of the equivalence of the original and new models. Therefore, we say ...

The proof is the massage.

It helps greatly if the designer has several models already shown to be equivalent to the definition of the language to choose from. He can choose the one which is closest to the desired new model, thus making the massages easier and the transformation changes smaller.

This technique was used by McGowan in the design of two of his lambda calculus interpreters, the fixed program machine [McG 71b] and the contour model lambda calculus machine [McG 72a].

Indeed, the author used this method in the preparation of this paper to extend an already existing version of the copy rule [Bry 71c] to cover tasking. He knew

1) the nature of the massages (mappings) between the single-task CM and single-task CR from an earlier proof [Bry 71c], and

2) what tasking in CM looks like [Joh 71, Bry 71b].

He applied the same massages to the CM tasking facilities, with a few fudges and kludges here and there, to come up with what had to be the CR tasking facilities.

11. Future work

There are several remaining questions and problems which have been suggested by the research entailed in this paper.

1) Develop techniques for proving the correctness of a compiler for multi-tasking languages,

2) The definition of operational equivalence given in this paper (definition 7) presumes that the computations in both models proceed at the same rate; that is, if S_i and S'_i are corresponding snapshots under the mappings then their successors S_{i+1} and S'_{i+1} are also corresponding snapshots under the mapping. This definition is not general enough to deal with models such that many steps in a computation in one model simulate one step in the other (and vice versa even within one computation). One way of dealing with this is to modify the transformations of the models so that they do less or more work in one step. A better way is to generalize the definition of operational equivalence. Such a generalized definition has already been given for single-task languages [McG 71a,b, MW 71, Weg 71b]. It should be easy to obtain a similar definition for multi-tasking languages which would include the one given in this paper as a special case.

Acknowledgements. Thanks go to John Hutchinson, John Johnston, John Kelly and Peter Wegner for their incisive comments and suggestions. Special thanks go to Clem McGowan for encouragement in extending to the proof techniques used in this paper. Finally, thanks go to Trina Avery for sacrificing her time in the final production of this paper.

Bibliography

NOTE: DSIPL (pronounced "disciple") is Proceedings of ACM Symposium on Data Structures in Programming Languages, SIGPLAN Notices, February 1971. PAAP is Proceedings of ACM Symposium on Proofs of Assertions about Programs, January 1972.

Bry 71a Berry, D.M., "Introduction to Oregano", DSIPL.

Bry 71b Berry, D.M., "Tasking in Oregano", Proc. 5th Princeton Conf. IS&S, 1971.

Bry 71c Berry, D.M., "Block structure: retention or deletion?, Third ACM Symposium on Theory of Computing (May 1971).

Bet 70 Betourne, C., et al., "Process management and resource sharing in the multi-access system ESOPE", CACM 13:12, 727 (1970).

BG 70 Boyle, J.M., and Grau, A.A., "An algorithmic semantics for Algol 60 identifier denotation", JACM 17:2, 361 (1970).

Dij 68 Dijkstra, E.W., "Cooperating sequential processes" in Genuys, ed., Programming languages, London: Academic Press (1968).

HJ 70 Henhapl, W., and Jones, C.B., The block concept and some possible implementations, with proofs of equivalence, IBM Laboratory Vienna, Tech. Rep. TR 25.104 (1970).

Hut 71 Hutchison, J.S., private communication, 1971.

JL 71 Jones, C.B., and Lucas, P., "Proving correctness of implementation techniques" in Engeler, ed., Symposium on Semantics of Algorithmic Languages, Berlin: Springer-Verlag (1971).

Joh 71 Johnston, J.B., "The contour model of block structured processes", DSIPL.

McG 70 McGowan, C., "The correctness of a modified SECD machine", Second ACM Symposium on Theory of Computing (1970).

McG 71a McGowan, C., "An inductive proof technique for interpreter correctness", Courant Institute Symposium on Formal Semantics of Programming Languages (1971).

McG 71b McGowan, C., Correctness results for lambda calculus interpreters, Ph.D. thesis, Cornell University (1971).

McG 72a McGowan, C., "A contour model lambda calculus machine", PAAP.

McG 72b McGowan, C., "The 'most recent' problem - its causes and correction", PAAP.

MW 71 McGowan, C., and Wegner, P., "The equivalence of sequential and associative information structure models", DSIPL.

Nau 63 Naur, P., "Revised report on the algorithmic language Algol 60", CACM 6:1 (1963).

vWn 69 van Wijngaarden, A., et al., "Report on the algorithmic language Algol 68", Num. Math. 14, 79-218 (1969).

Wlk 69 Walk, K., et al., Formal definition of PL/1, ULD Version III IBM Vienna (1969).

Weg 70 Wegner, P., "Information structure models for programming languages", TR-70-22, Center for Computer and Information Sciences, Brown University, September 1970.

Weg 71a Wegner, P., "Data structure models for programming languages", DSIPL.

Weg 71b Wegner, P., "Programming language semantics", Courant Institute Symposium on Formal Semantics of Programming Languages (1971).

Weg 72 Wegner, P., "Operational semantics for programming languages", PAAP.