PROGRAMMING WITH TRANSITION DIAGRAMS

(Joint report with
Artificial Intelligence Department)

by

John C. Reynolds
(Syracuse University)

PROGRAMMING WITH TRANSITION DIAGRAMS

John C. Reynolds

Syracuse University

ABSTRACT In situations where the avoidance of goto statements would be inhibiting, programs can be methodically constructed by using transition diagrams. The key requirement is that the relevant assertions must be simple enough to permit exhaustive reasoning. The method is illustrated by programs for fast exponentiation, merging, and path-finding in a directed graph.

Work supported by National Science Foundation Grant MCS 75-22002

Introduction

The goto controversy has generated more heat than light, and has exaggerated the importance of control structures in programming methodology. Yet hiding behind this controversy is a serious question. It is not whether goto's are good or bad, nor whether some new control structure can provide a desireable balance between the flexibility of goto's and the intelligibility of simple iteration and conditional statements. The real question is how - in situations where their flexibility is really needed - to use goto's methodically, reliably, and clearly.

By means of three examples, I would like to suggest that the answer is to use transition diagrams with assertions at their nodes, and to keep these assertions simple enough that they can be reasoned about exhaustively.

Fast Exponentiation

As a first example, consider computing x^n in time log n. We want a program satisfying the specification

$$\{n \geq 0\} \text{ "Compute } x^n\text{"} \{y = x^n\} \quad .$$

To construct this program in a top-down fashion, without using goto's, we begin with the invariant

$$I \equiv y \times z^k = x^n \ \& \ k \geq 0 \quad .$$

Since this invariant embodies the essential idea behind the algorithm, one would expect the rest of the program development to be straightforward.

The invariant can be attained by an obvious initialization, and it implies the goal of the program when k = 0. Thus "Compute x^n" expands into

```
begin integer k; real z;
y := 1.0; z := x; k := n;
"Achieve k = 0 while maintaining I";
end ,
```

where "Achieve k = 0 while maintaining I" obviously expands into

```
while k ≠ 0 do
     "Decrease k while maintaining I" .
```

It would be correct to expand "Decrease k while maintaining I" into the statement

STEP ≡ begin k := k - 1; y := y × z end ,

which satisfies {I & k ≠ 0} STEP {I}. But using only STEP gives a slow algorithm; for speed we must also employ the statement

HALVE ≡ begin k := k ÷ 2; z := z × z end ,

which satisfies {I & even(k)} HALVE {I}.

Thus to complete the algorithm we must compose "Decrease k while maintaining I" out of the slow STEP, which always works, and the fast HALVE, which only works for even k. However, there are at least two such compositions which are correct:

```
if odd(k) then STEP else HALVE                          (I)

begin if odd(k) then STEP; HALVE end .                  (II)
```

Moreover, each of these versions is in some respect worse than the other.

Their limitations are more evident at the level of "Achieve k = 0 while maintaining I":

```
while k ≠ 0 do                                          (I)
    if odd(k) then STEP else HALVE

while k ≠ 0 do                                          (II)
    begin if odd(k) then STEP; HALVE end .
```

In (I) there is redundant testing; each execution of STEP will produce an even number which will be subjected unnecessarily to the test odd(k). In (II) the loop will conclude with a final execution of HALVE which will not affect the result, but which can cause unnecessary overflow.

It is at this point that the prohibition of goto's seems to chafe. To avoid the limitations of (I) and (II), we will collapse the structure of our algorithm slightly, and express "Achieve k = 0 while maintaining I" directly in terms of STEP and HALVE, using a transition diagram.

The nodes of this transition diagram will be the possible states of knowledge about the conditions which are relevant to the program. Except for the invariant, which is always true at this level of abstraction, the only relevant conditions are k = 0 and even(k). For one of these conditions there are three states of knowledge: k = 0, k ≠ 0, and "don't know". For the two conditions, there would be nine states if the

conditions were independent, but since k = 0 implies even(k) there are only six states, described by the assertions:

true (i.e., don't know)
k = 0
k ≠ 0
even(k)
odd(k)
k ≠ 0 & even(k) .

Two kinds of arcs will occur in the transition diagram. Tests will be represented by complementary pairs of arcs indicated by dashed lines, and operations will be represented by individual arcs indicated by solid lines. The test arcs are obvious - we simply put in all possible tests of the relevant conditions which provide increased information.

The next step is to determine the preconditions for performing the various operations. If k = 0, exit should occur, since the goal of the program has been achieved. If k ≠ 0 & even(k), HALVE can be performed. If odd(k), then STEP must be performed.

Finally, we must analyze STEP and HALVE to determine the information about k = 0 and even(k) which will be known after execution. It is easily seen that STEP and HALVE satisfy

{I & odd(k)} STEP {I & even(k)}

{I & k ≠ 0 & even(k)} HALVE {I & k ≠ 0}

(which are more stringent than the specifications given earlier). These specifications directly determine the placement of the operation arcs.

Thus "Achieve k = 0 while maintaining I" is realized by the transition diagram shown in Figure 1. Entrance occurs at the node true since the precondition of the program does not imply any information about k = 0 or even(k).

The foregoing argument should convince the reader of conditional correctness. For total correctness, one must also be sure that there are no dead ends - which is obvious by inspection - and that no loops can run on forever. To clarify the latter question, the geometry of Figure 1 has been chosen so that, if true is taken as the origin, then increasing distance represents increasing information. As a consequence, the only arcs which do not increase distance from the origin - at least one

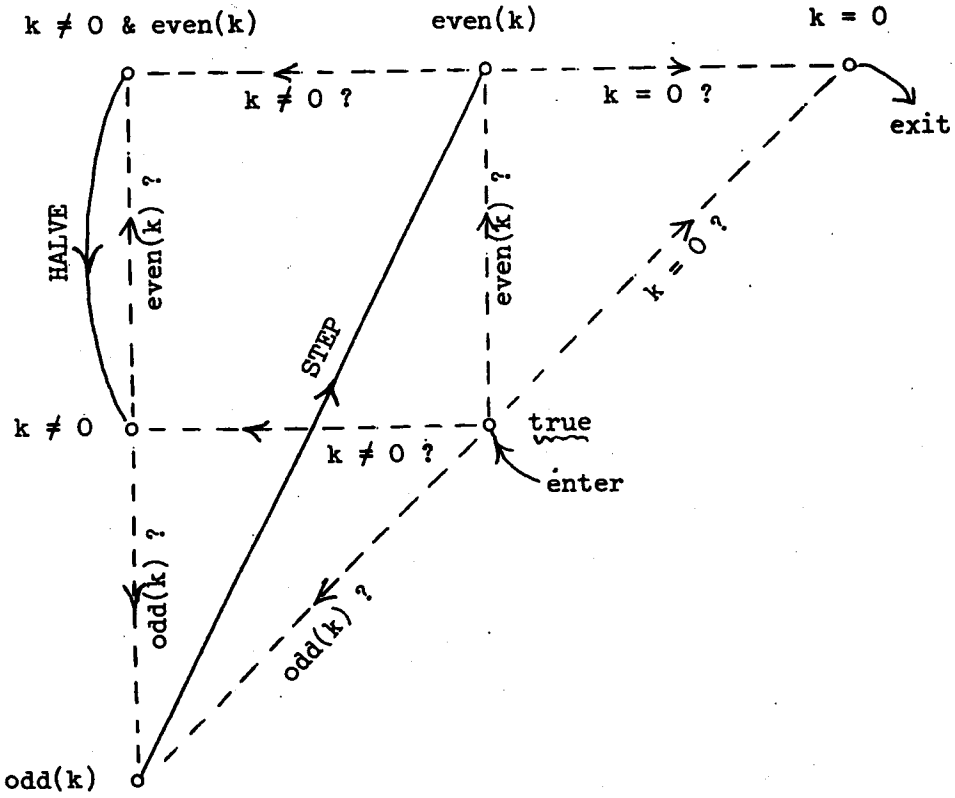

Figure 1. A Transition Diagram for Fast Exponentiation.

of which must occur in every loop - are the operations STEP and HALVE. Thus termination is assured since, when their specified preconditions are true, both of these operations will always decrease k without making it negative.

Emanating from the entrance node are two pairs of arcs denoting distinct tests, so that the transition diagram is nondeterminate. However this is an advantage: we have shown that all possible executions of the diagram will be correct, so that we are free to resolve the nondeterminacy without regard to correctness. In this case, the decision doesn't matter much, though a weak argument can be made that beginning with the test odd(k) will usually be faster.

The result is a program which avoids the defects of both earlier versions. Moreover, the program is still essentially structured, since the nature of the invariant I only affects the details of STEP and HALVE and plays no role in the construction of the transition diagram. In effect, the transition diagram is a program for a machine whose primitive instructions, STEP and HALVE, are guaranteed by the machine designer to preserve the integrity of the machine (i.e. the invariant) when used in accordance with their specifications. Of course, it is precisely this suppression of the invariant which permits us to limit our attention to the conditions k = 0 and even(k), which are so simple that we can easily deal with all possible states of knowledge.

On the other hand, our program is certainly less structured than versions (I) or (II), and consequently more complicated. But the result is not chaos. There is still enough structure that the program can be constructed and understood methodically.

To represent a transition diagram in a conventional language, one can use a block in which each node appears as a label with an attached assertion, each operation is a sequence of statements beginning with a label and ending with a goto statement, and each test is a conditional statement containing a pair of goto statements. The invariant, which must hold at all labels, is stated at the beginning of the block. For example, the program for fast exponentiation is:

```
{n ≥ 0}
begin integer k; real z;
y := 1.0; z := x; k := n;
     begin {invariant: y × z^k = x^n & k ≥ 0}
     enter: {true}              if odd(k) then goto od else goto ev;
     nz:    {k ≠ 0}             if odd(k) then goto od else goto nzev;
     ev:    {even(k)}           if k = 0 then goto zr else goto nzev;
     od:    {odd(k)}            k := k - 1; y := y × z; goto ev;
     nzev:  {k ≠ 0 & even(k)}   k := k ÷ 2; z := z × z; goto nz;
     zr:    {k = 0}
     end
end
{y = x^n} .
```

For clarity, we have avoided passing through a label from the preceding statement. The labels have been ordered to make termination evident: Each backward jump is preceded by a statement which decreases k.

To the author's surprise, D. Gries has found a goto-free program (using only ";", conditional, and while constructions to combine statements) which is schematically equivalent to the above. Its construction, and the question of whether it is more natural than the use of transition diagrams, is left as an exercise for the reader.

Merging

As a second example, consider merging two ordered arrays, X, with subscripts from ax to bx, and Y, with subscripts from ay to by, into an array Z, with subscripts from az to bz, which is just the right size to receive the result. A precise specification is

```
{ax ≤ bx+1 & ay ≤ by+1 & az ≤ bz+1
     & (bx - ax + 1) + (by - ay + 1) = (bz - az + 1)
     & ordered(X, ax, bx) & ordered(Y, ay, by)}
"Merge"
{ordered(Z, az, bz) & merged(X, ax, bx, Y, ay, by, Z, az, bz)} ,
```

where

$$\text{ordered}(X, a, b) \equiv (\forall i, j)\ a \le i < j \le b \text{ implies } X(i) \le X(j) ,$$

and merged(...) asserts that Z is a rearrangement of the concatenation of X and Y, i.e.,

$$\begin{aligned}
&\text{merged}(X, \underline{ax}, \underline{bx}, Y, \underline{ay}, \underline{by}, Z, \underline{az}, \underline{bz}) \equiv \\
&\quad (\exists P)\ P \in \text{perm}(\{i \mid \underline{az} \le i \le \underline{bz}\}) \ \& \\
&\qquad (\forall i)\ \underline{az} \le i \le \underline{bz} \text{ implies } Z(P(i)) = \\
&\qquad\quad \underline{\text{if}}\ i - \underline{az} + \underline{ax} \le \underline{bx}\ \underline{\text{then}}\ X(i - \underline{az} + \underline{ax}) \\
&\qquad\quad \underline{\text{else}}\ Y(i - \underline{az} - (\underline{bx} - \underline{ax} + 1) + \underline{ay}) \ ,
\end{aligned}$$

where perm(S) is the set of permutations (one-to-one functions) from the set S onto itself.

Since all three arrays will be scanned from left to right, each array will be partitioned into a processed and an unprocessed part, as shown in Figure 2. The invariant describes this partitioning and asserts that the unprocessed part of Z is the right size, that the processed part of Z is ordered and is a rearrangement of the concatenation of the processed parts of X and Y, and that all processed values are smaller or equal to all unprocessed values:

$$\begin{aligned}
I \equiv\ & \underline{ax} \le \underline{kx} \le \underline{bx}+1 \ \& \ \underline{ay} \le \underline{ky} \le \underline{by}+1 \ \& \ \underline{az} \le \underline{kz} \le \underline{bz}+1 \\
& \& \ (\underline{bx} - \underline{kx} + 1) + (\underline{by} - \underline{ky} + 1) = (\underline{bz} - \underline{kz} + 1) \\
& \& \ \text{ordered}(Z, \underline{az}, \underline{kz}-1) \\
& \& \ \text{merged}(X, \underline{ax}, \underline{kx}-1, Y, \underline{ay}, \underline{ky}-1, Z, \underline{az}, \underline{kz}-1) \\
& \& \ \text{all}\underline{\le}(Z, \underline{az}, \underline{kz}-1, X, \underline{kx}, \underline{bx}) \\
& \& \ \text{all}\underline{\le}(Z, \underline{az}, \underline{kz}-1, Y, \underline{ky}, \underline{by}) \ ,
\end{aligned}$$

where

$$\begin{aligned}
&\text{all}\underline{\le}(X, a, b, Y, c, d) \equiv \\
&\quad (\forall i, j)\ \ a \le i \le b \ \& \ c \le j \le d \text{ implies } X(i) \le Y(j) \ .
\end{aligned}$$

This invariant can be attained by making the processed array parts empty, and it implies the goal of the program when the unprocessed parts of both X and Y (and therefore of Z) are empty. Thus "Merge" expands into:

```
begin integer kx, ky, kz;
kx := ax; ky := ay; kz := az;
"Achieve kx > bx and ky > by while maintaining I"
end .
```

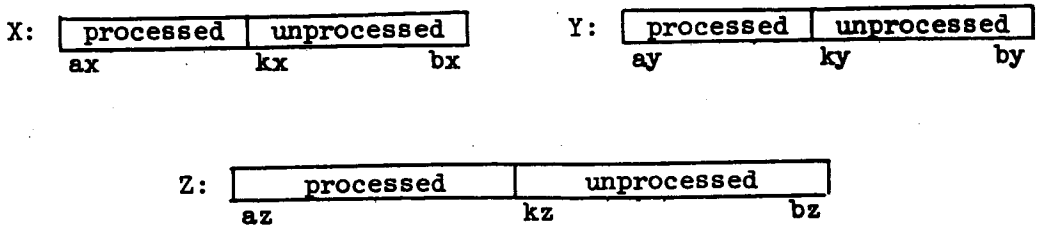

Figure 2. Arrays for Merging.

The rest of the program will be built out of two operations:

COPYX ≡ begin Z(kz) := X(kx); kx := kx+1; kz := kz+1 end

COPYY ≡ begin Z(kz) := Y(ky); ky := ky+1; kz := kz+1 end .

Roughly speaking, either COPYX or COPYY will preserve the invariant, depending upon whether X(kx) or Y(ky) is the smallest unprocessed value. However, a precise specification must take into account the possibility that the unprocessed part of X or Y may be empty:

{I & kx ≤ bx & (ky > by or (ky ≤ by & X(kx) ≤ Y(ky)))} COPYX {I}

{I & ky ≤ by & (kx > bx or (kx ≤ bx & Y(ky) ≤ X(kx)))} COPYY {I} .

At this stage we may suppress the rather formidable invariant, and focus on the remaining conditions kx ≤ bx, ky ≤ by, and X(kx) ≤ Y(ky). The first two conditions are independent, and therefore lead to 3 × 3 = 9 states, but X(kx) ≤ Y(ky) is only meaningful when both kx ≤ bx and ky ≤ by are true (i.e. when both unprocessed array parts are nonempty), and therefore leads to only two additional states.

Exit can occur when kx > bx & ky > by, and the conditions when COPYX and COPYY can occur are shown clearly by their specifications. Finally, if we examine the effect of COPYX separately for the two nodes at which it will occur, we can deduce the more stringent specification

{I & kx ≤ bx & ky > by} COPYX {I & ky > by}

{I & kx ≤ bx & ky ≤ by & X(kx) ≤ Y(ky)} COPYX {I & ky ≤ by}

since COPYX does not affect the emptyness of the unprocessed part of Y. Similarly,

{I & ky ≤ by & kx > bx} COPYY {I & kx > bx}

{I & ky ≤ by & kx ≤ bx & X(kx) ≥ Y(ky)} COPYY {I & kx ≤ bx} .

Thus "Achieve kx > bx and ky > by while maintaining I" can be realized by the transition diagram shown in Figure 3.

In comparison with most merging programs, this version has the virtue of avoiding redundant testing of the emptyness of unprocessed array parts. Moreover, in the nondeterministic version, the program preserves the symmetry between X and Y which characterized the original problem. However, this symmetry must be broken to resolve the nondeterminacy (which occurs at the entrance node and at kx ≤ bx & ky ≤ by when X(kx) = Y(ky

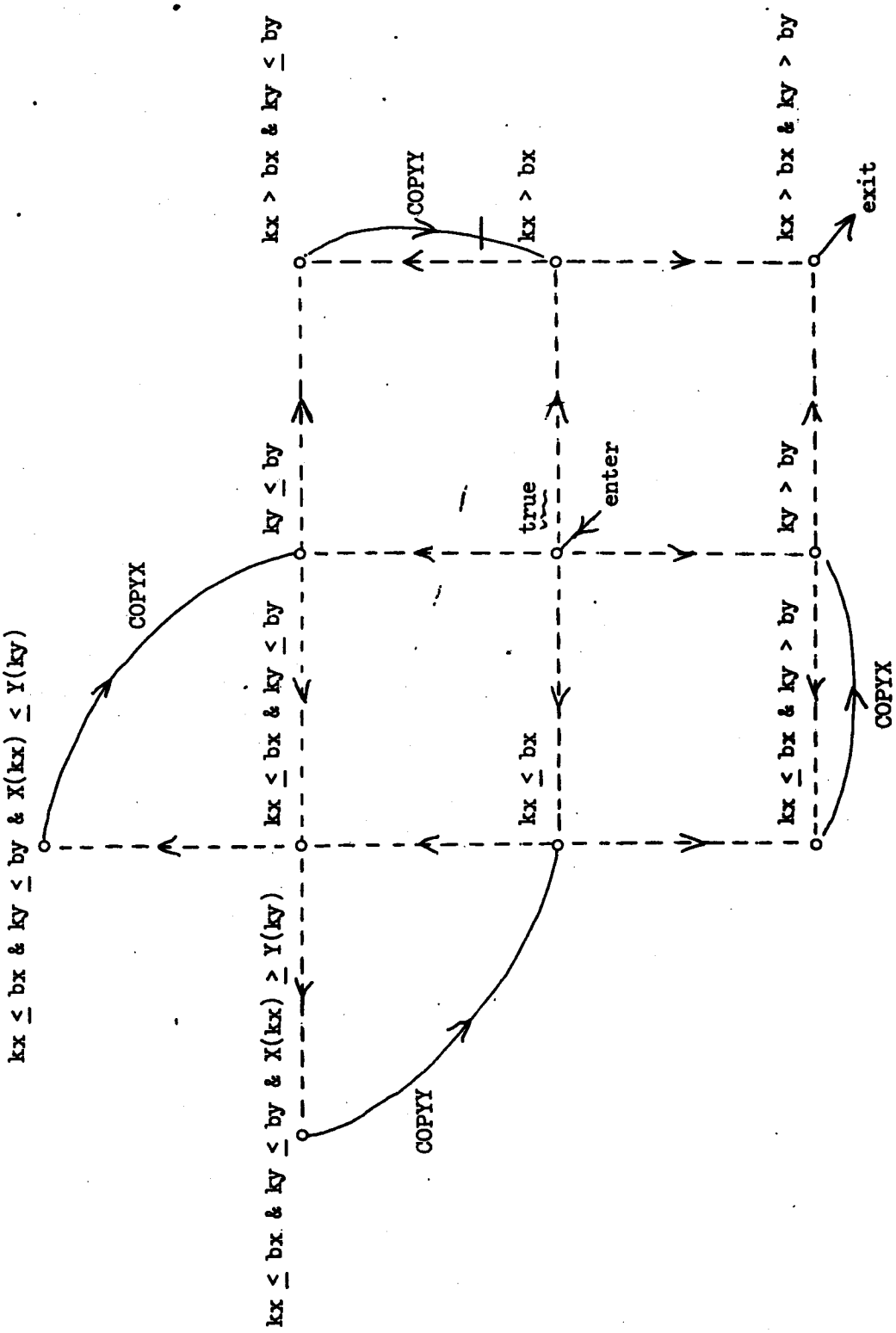

Figure 3. A Transition Diagram for Merging.

Path-Finding in a Directed Graph

The final example shows the use of transition diagrams in an abstract, rather than a concrete program, and it also illustrates the utility of goto's which jump out of blocks and procedures. We will construct a program which, given a finite directed graph and two sets X and Y of nodes, will determine whether a path exists from some member of X to some member of Y. The program will be abstract in the sense that node and nodeset will be used as primitive data types, with appropriate primitive operations.

For a node set S, let $\Gamma(S)$ ($\Gamma^*(S)$, $\Gamma^\dagger(S)$, $\Gamma^{\dagger*}(S)$) stand for the set of immediate successors (successors, immediate predecessors, predecessors) of S, i.e., the nodes which can be reached in one step forward (zero or more steps forward, one step backward, zero or more steps backward) from members of S. This notation can be used to describe the existence of a path from X to Y in three equivalent ways:

$$\begin{aligned} G &\equiv \text{nonempty}(\Gamma^*(X) \cap Y) \\ &\equiv \text{nonempty}(X \cap \Gamma^{\dagger*}(Y)) \\ &\equiv \text{nonempty}(\Gamma^*(X) \cap \Gamma^{\dagger*}(Y)) \ . \end{aligned}$$

The program will send control to the label success if G is true, or to the label failure if G is false.

The basic method of computation is to maintain two disjoint sets: TX containing successors of X, and TY containing predecessors of Y. There will also be a subset SX of TX containing nodes whose immediate successors are known to belong to TX, and a subset SY bearing the analogous relation to TY. The situation can be described by the two invariants:

$$I_1 \equiv SX \subseteq TX \subseteq \Gamma^*(X) \ \& \ SY \subseteq TY \subseteq \Gamma^{\dagger*}(Y) \ \& \ \text{empty}(TX \cap TY)$$

$$I_2 \equiv X \subseteq TX \ \& \ \Gamma(SX) \subseteq TX \ \& \ Y \subseteq TY \ \& \ \Gamma^\dagger(SY) \subseteq TY \ .$$

The first of these invariant plays a special role. Although we will not carry the development of our program to the point of designing a concrete data representation, it is important to realize that I_1 provides a constraint which could be used at the concrete level to permit a more efficient choice of data representation than would otherwise be possible. But to take advantage of this constraint, we must insure that I_1 holds,

not just at certain points, but at all points in the abstract program.

We begin by devising operations for adding elements to the sets TX and TY while maintaining I_1. It is easily seen that

$$\{z \notin TY \ \&\ I_1 \ \&\ z \in \Gamma^*(X)\}\ TX := TX \cup \{z\}\ \{I_1\}\ .$$

On the other hand, $z \in TY \ \&\ I_1 \ \&\ z \in \Gamma^*(X)$ implies G, which permits a jump to the label success. Thus

$$\{I_1 \ \&\ z \in \Gamma^*(X)\}\ \underset{\sim}{\text{if}}\ z \in TY\ \underset{\sim}{\text{then}}\ \underset{\sim}{\text{goto}}\ \underline{\text{success}}\ \underset{\sim}{\text{else}}\ TX := TX \cup \{z\}\ \{I_1\}.$$

In effect, we have an operation which behaves like $TX := TX \cup \{z\}$, yet is miraculously guaranteed to keep TX and TY disjoint, since any program which performs the operation with $z \in TY$ will cease to be executed.

Next, using an obvious notation for iterating over a set (in an unspecified order), we can build an operation for adding an entire set to TX:

```
procedure putx(nodeset W);
    for z ε W do if z ε TY then goto success else TX := TX ∪ {z} ,
```

which behaves like $TX := TX \cup W$, yet satisfies

$$\{I_1 \ \&\ W \subseteq \Gamma^*(X)\}\ putx(W)\ \{I_1\}\ .$$

Applying similar reasoning to TY gives

```
procedure puty(nodeset W);
    for z ε W do if z ε TX then goto success else TY := TY ∪ {z} ,
```

which behaves like $TY := TY \cup W$, yet satisfies

$$\{I_1 \ \&\ W \subseteq \Gamma^{+*}(Y)\}\ puty(W)\ \{I_1\}\ .$$

Having developed these basic operations, we can return to the overall program, and proceed from the top downwards. By induction on path length, it can be shown that $I_2 \ \&\ SX = TX$ implies $\Gamma^*(X) \subseteq TX$ and that $I_2 \ \&\ SY = TY$ implies $\Gamma^{+*}(Y) \subseteq TY$. Thus $I_1 \ \&\ I_2 \ \&$ (SX = TX or SY = TY) implies that G is false. This suggests the program:

```
begin nodeset TX, TY, SX, SY;
"Achieve I1";
"Achieve I2 while maintaining I1";
"Achieve SX = TX or SY = TY while maintaining I1 & I2";
goto failure
end .
```

It is straightforward to fill in the first two operations:

```
begin nodeset TX, TY, SX, SY;
TX := TY := SX := SY := {};
putx(X); puty(Y);
"Achieve SX = TX or SY = TY while maintaining I1 & I2";
goto failure
end .
```

(However, note that TX := X; TY := Y would be an incorrect second operation, since I_1 would not be preserved if X and Y had members in common.)

To develop the rest of the program, we will construct a transition diagram using the following pair of operations:

```
STEPX ≡ begin node u;
        u := choose(TX - SX); SX := SX ∪ {u};
        putx(Γ(u))
        end

STEPY ≡ begin node u;
        u := choose(TY - SY); SY := SY ∪ {u};
        puty(Γ†(u))
        end ,
```

which increase SX or SY while preserving both invariants:

$$\{I_1 \;\&\; I_2 \;\&\; SX \neq TX\}\ \text{STEPX}\ \{I_1 \;\&\; I_2\}$$

$$\{I_1 \;\&\; I_2 \;\&\; SY \neq TY\}\ \text{STEPY}\ \{I_1 \;\&\; I_2\}\ .$$

Here choose(S) is a primitive operation which accepts a nonempty set and produces an unspecified member. It is vital to leave this operation nondeterministic at the abstract level in order to provide an adequate degree of freedom for the design of a data representation. (Indeed, this is probably the most vital role of nondeterminism in programming.)

The relevant conditions for the transition diagram are SX = TX and SY = TY, which are independent and therefore lead to a transition diagram with nine states. Exit can occur in any state in which either SX = TX or SY = TY. On the other hand, STEPX and STEPY should only be performed in states where there is no possibility of exiting or of performing a test which might lead to an exit. The only such state is SX ≠ TX & SY ≠ TY, which permits either STEPX or STEPY. Moreover, in this state STEPX will

maintain SY ≠ TY, since it does not change SY or TY, and STEPY will similarly maintain SX ≠ TX. Thus we have the transition diagram shown in Figure 4.

Most of the nondeterminism can be resolved by exiting whenever possible and by arbitrarily choosing the test SX = TX at the initial state. (As a consequence, the two states SX = TX & SY = TY and SY = TY become disconnected, and vanish from the transition diagram.) However, the remaining nondeterminism is more problematical. Dropping STEPX or STEPY would give a correct but inefficient program which only searches forward from X or backward from Y. For most graphs, it is better to search in both directions so that, if a path exists, TX and TY will intersect at an intermediate point. The simplest way of accomplishing this is to alternate STEPX and STEPY, which can be done by spliting the node SX ≠ TX & SY ≠ TY as shown in Figure 5.

An Afterthought

After devising these ideas, I have had the experience of teaching them, and my enthusiasm has been tempered by a significant minority of students who have employed them both incorrectly and unnecessarily. Transition diagrams are a tool for constructing complicated programs systematically, but they do not justify the use of such programs when simpler ones would do the job. They are unlikely to be useful in the hands of programmers who habitually underestimate the limitations of their own intellects.

ACKNOWLEDGEMENTS

I am indebted to the members of IFIP Working Group 2.3, who have provided motivation, inspiration, and helpful criticism. I am also grateful for the hospitality of the University of Edinburgh and the support of the Science Research Council during the period when this paper was written.

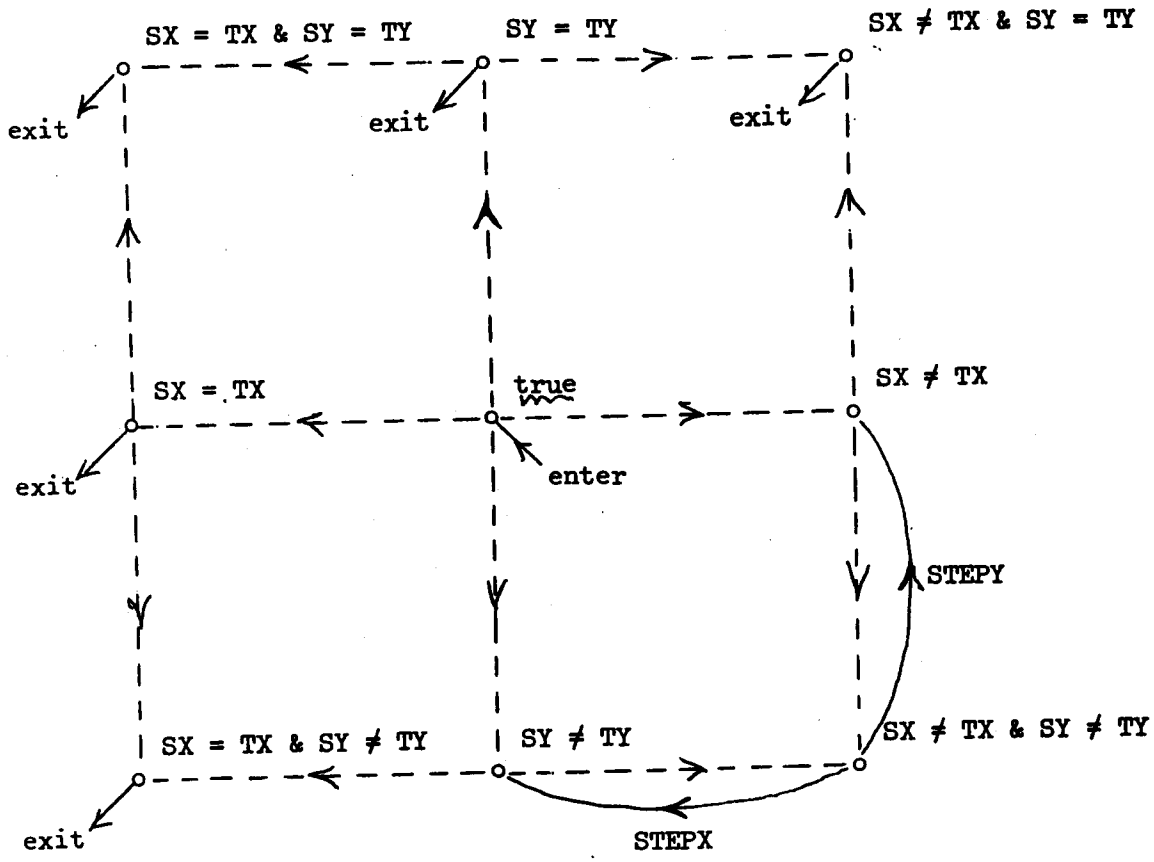

Figure 4. A Transition Diagram for Path Finding.

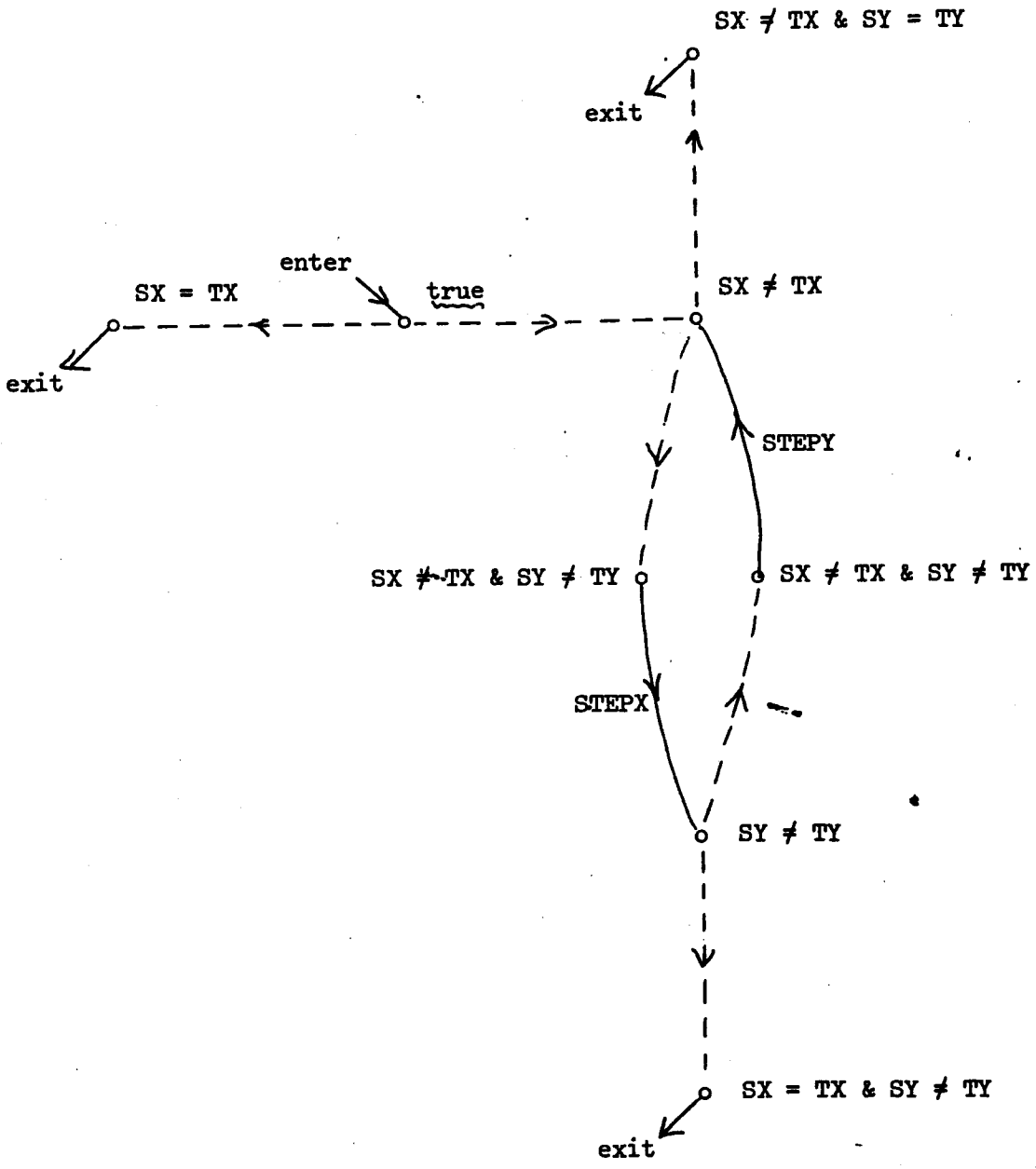

Figure 5. A Determinate Transition Diagram for Path Finding.