The Algorithmic Transformation of Schemas to Structured Form

G. OULSNAM

Department of Computer Science, University College, Cork, Republic of Ireland

Algorithms are given to transform unstructured program schemas into equivalent structured forms. These algorithms are shown to have a computational complexity which is linearly related to schema size for almost all schemas, but at worst exponential with an exponent greater than but asymptotically close to one for large problems. Structuring is achieved by first identifying the forward paths of the schema and reducing them to an equivalent structured elementary path E. Each back path of E is then recursively structured to an equivalent single back arc, thus reducing the original schema to a single elementary path together with a set of possibly overlapping back arcs. The remaining unstructuredness is removed by recursive application of a loop-structuring algorithm. The algorithms are illustrated by application to a complex hypothetical schema and to two practical problems.

Received August 1984

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

1. INTRODUCTION

This paper presents algorithms to convert unstructured program schemas into equivalent structured forms. Unstructuredness occurs in four ways, namely jumps into and out of decision and loop constructs, to give multi-entry and multi-exit control structures. An earlier paper[1] gave general transforms for the structuring of such constructs, identified the minimum set of such transforms needed, and the necessary and sufficient conditions for their effective application to any schema. Not considered were the problems of identifying unstructured subschemas and the general strategy for achieving structuredness. That omission is now rectified to give between the two papers a complete theory for the algorithmic structuring of schemas.

It must be emphasised that the theory relates to *schemas* rather than to any particular interpretation of them as *programs* and as such has universal applicability. None the less, as will be shown in an example, consideration of a particular interpretation can give rise to further simplifications dependent upon that interpretation and therefore beyond the scope of the present theory. Even so, the algorithms presented here will often produce programs that are not only structured and computationally equivalent to the original forms but also logically pleasing.

2. SCHEMAS AND THEIR STRUCTURING TRANSFORMS

Schemas

A *schema* is a labelled graph $G = (V, \Gamma, \Sigma)$, where V is a set of nodes, Γ is the immediate successor function mapping V into sets of nodes over V, and Σ is an alphabet of operators representing predicates denoted by $p, q, \ldots$ and their negations $\sim p$, $\sim q, \ldots$, and basic blocks denoted by $a, b, \ldots$. If $u, v \in V$, $v \in \Gamma u$, and $\alpha \in \Sigma^*$ then $(u, v{:}\alpha)$ is an arc directed from u to v with label α. The set of all arcs over G is denoted by E. The labels of arcs are represented by $\alpha, \beta, \ldots$ and are always in the form of regular expressions over Σ. The empty basic block is written λ. Whenever the label of an arc is of no concern the arc will be written in the form (u, v).

A *path* $[u, v]$ is a sequence of arcs (u_0, u_1) $(u_1, u_2) \ldots (u_{n-1}, u_n)$ with $u = u_0$ and $v = u_n$. Paths will also be written in the form $a-b-\ldots-u-v$ where $a, b, \ldots, u, v$ is the sequence of nodes on the path. An *elementary path* is a path on which the same node does not occur more than once; a *simple path* is one on which the same arc does not occur more than once. Any simple path $[u, u]$ is a *cycle* or *loop*.

The nodes of a schema are of five types: *start*, *halt*, *collector*, *decision* and *chain* having respectively in-degrees of 0, 1, 2, 1 and 1, and out-degrees of 1, 0, 1, 2 and 1. A schema G is in *reduced form* if it has exactly one start node s and exactly one halt node h, and for all other nodes v there is a path $[s, v]$ and a path $[v, h]$ in G. A schema is in *standard form* if it contains no loop of the form $(u, v{:}\alpha)$ $(v, u{:}p\beta)$, no decision of the form $(u, v{:}p\alpha)$ and $(u, v{:} \sim p\beta)$, and no chain node. The first two constructs can be replaced by the single arcs $(u, v{:}\alpha.(p\beta.\alpha)^*)$ and $(u, v{:}p\alpha + \sim p\beta)$ respectively, whilst chain nodes can be elided by concatenation of the incident arcs and their labels to form a single arc. The equivalent regular expressions $(\alpha.p.\beta)^*.\alpha. \sim p$ and $\alpha.(p.\beta.\alpha)^*. \sim p$ will be written in the form $(\alpha.p\beta)^+ . \sim p$.

A reduced schema in standard form is a *structured schema* if and only if it comprises a single arc labelled by a regular expression over Σ.

The structuring transforms

The structuring transforms already developed[1] are summarized in Figs 1 and 2. Fig. 1*a* illustrates a jump into a decision ID from R to C and a jump out of a decision OD at D to S. The paths α and β each contain zero or more instances of further ID and OD constructs, whereas a, b ... represent basic block computations. The preferred order of structuring[1] is to remove the ID constructs first and then the ODs. Proceeding in this manner and structuring the ID at C using the technique of arc duplication yields the schema shown in Fig. 1*b*, and then structuring the OD at D gives the schema shown in Fig. 1*c*. The introduced symbol Q in Fig. 1*c* is a predicate flag which does not occur in the original schema and is used to record the value of predicate q as computed at D.

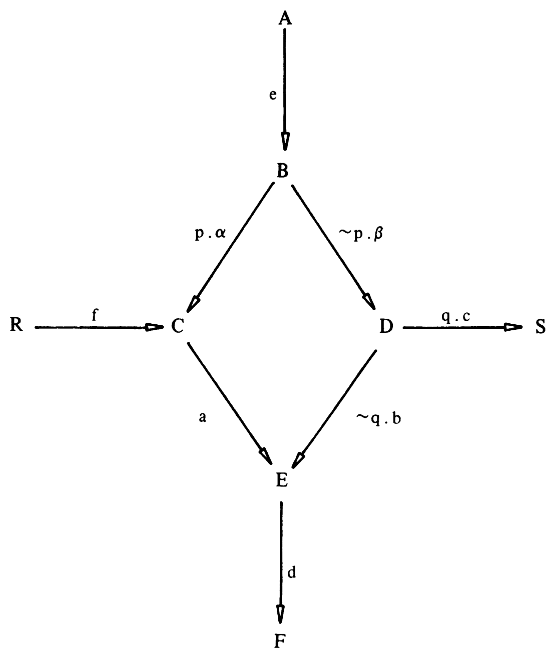

Figure 1*a*. Paradigms of ID and OD constructs.

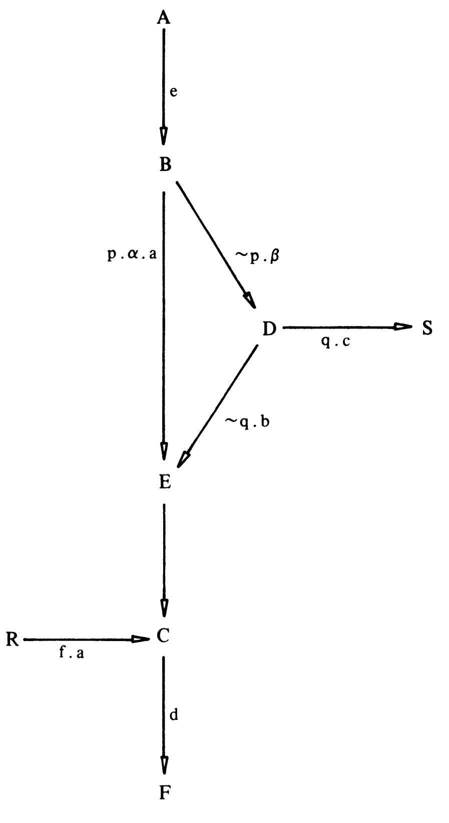

Figure 1*b*. Result of applying transform ID-0.

The transformation from Fig. 1*b* to 1*c* is called OD-1, the -1 indicating that one introduced predicate flag is needed to effect the structured equivalent form. Similarly, the transformation from Fig. 1*a* to 1*b* is called ID-0, the -0 indicating that no predicate flag is required. Structuring of ID can also be effected without the need to duplicate the arc labelled 'a' by introducing a predicate flag P and making the following changes to the labels in Figs 1*b* and 1*c*.

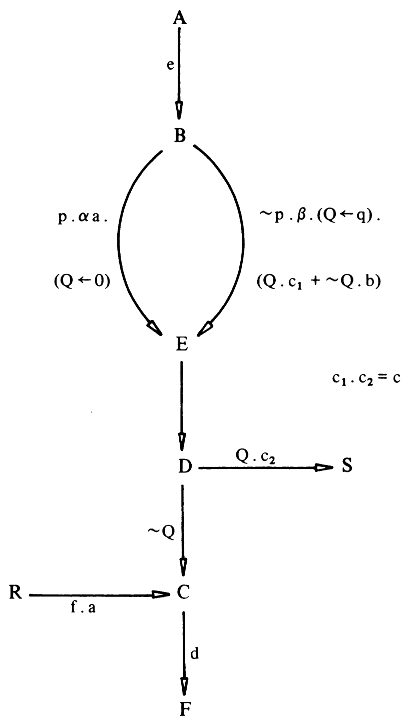

Figure 1*c*. Result of applying the transform ID-0 followed by OD-1.

Replace

e	by	$e.(P:=p)$
$p.\alpha.a$	by	$P.\alpha$
$\sim p.\beta$	by	$\sim P.\beta$
$f.a$	by	$f.(P:=\text{true});$
d	by	$(P.a+\sim P).d$

This transform is called ID-1 to denote that a single introduced predicate is required to effect structuring.

It is immaterial which transformation is chosen to structure ID; the essential point is that the structuring of ID repositions its entry node immediately after the decision exit node, and the structuring of OD repositions its exit node likewise. As the ID and OD transforms are independent (apart from the requirement that where a choice exists the ID transforms are performed first) the following result is immediate:

Lemma 1

Let v be an immediate predecessor node of the decision exit node w of a decision subschema. If v is a collector, then applying either the ID-0 or the ID-1 transforms to the ID at v repositions v to become the immediate successor of w, and the previous immediate successor of w becomes the immediate successor of the newly positioned v. If v is a decision node, then applying OD-1 to the OD at v repositions v likewise.

In the event that both α and β label arcs rather than paths, then following structuring, the path AD in Fig. 1*c* is replaced by a single arc labelled, in the case where ID-0 is used, by

$$e.(p.\alpha.a.(Q:=\text{false}) + \sim p.\beta.(Q:=q).(Q.c_1+\sim Q.b))$$

thus eliding the decision entry node B and the decision

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

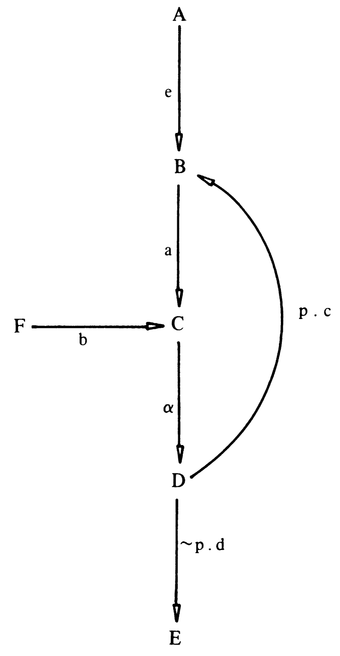

Figure 2*a*. Paradigm of the IL construct.

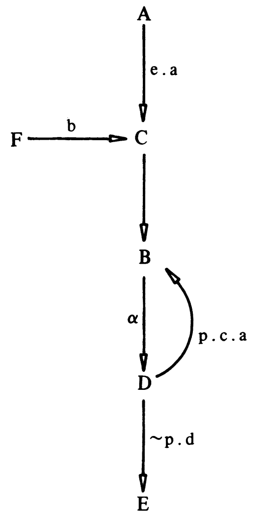

Figure 2*b*. Result of the transform IL-0.

exit node E. B and E are also elided if ID-1 is used instead, but with a different labelling of AD. Hence

Lemma 2

Let D be an unstructured decision subschema with entry node v and exit node w, and let u be the immediate predecessor of v, and x the immediate successor of w. Then the structuring of D by successive structuring of the topologically last instances of unstructuredness in D yields an elementary path $[u, x]$ from which v and w have been elided and the remaining nodes are ordered in accordance with Lemma 1.

That the structuring process defined in Lemma 2 must terminate is obvious, since each application of the ID or OD transformations, when applied in that order whenever a choice exists, reduces the number of occurrences of ID or OD in D by at least one.[1]

Fig. 2*a* illustrates a jump into a loop (called IL) where again α labels a path containing zero or more ILs and jumps out of the loop (OLs). As shown in Ref. 1, if a schema contains unstructured loops and there is no ID or OD construct present then there is always at least one loop of the form shown in Fig. 2*a*, that is, the immediate successor of the loop entry collector node is also a collector. It will be shown in section 3 that it is always sufficient to consider the retreating path DB of Fig. 2*a* as an arc, so that the IL construct depicted in Fig. 2*a* is the only form of loop unstructuredness that needs to be considered. Let this form of subschema be called a *simple loop subschema*. Fig. 2*b* shows the structured form of a simple loop using the technique of arc duplication, called IL-0. Analogously with ID-1, it is also possible to avoid arc duplication at the expense of an introduced predicate flag Q by relabelling Fig. 2*b* as follows:

replace

$e.a$	by	$e.(Q{:=}\text{true})$
b	by	$b_1.(Q{:=}\text{false})$
α	by	$(Q.a + {\sim}Q.b_2.(Q{:=}\text{true})).\alpha$
$p.c.a$	by	$p.c$

where

$$b_1.b_2 = b$$

to give the IL-1 transform. If α labels a single arc then the path $[C, E]$ becomes, in standard form, a single arc labelled in the case of IL-0 by

$$[\alpha.p.c.a]^{+}.\sim p.d$$

thus eliding nodes B and D. The same form of structured subschema, but with a different labelling derived from the foregoing substitutions, is obtained for IL-1.

Let L be a simple loop and let v be the collector entry of L, u the immediate predecessor of v, x the immediate successor of v and w the decision exit of L. Then:

Lemma 3

Structuring L at x repositions x as the immediate successor of u and as the immediate predecessor of v.

Lemma 4

Structuring L results in an elementary path $[u, x]$ from which v and w have been elided, and the remaining nodes of L are ordered according to Lemma 3.

The verification of all four lemmas is straightforward and left to the reader.

3. THE GENERAL STRUCTURING ALGORITHM

Before proceeding to the general structuring algorithm for schemas, some additional concepts need to be defined.

Forward paths and forward path subschemas

A *forward path* F in a schema G is an elementary path $[s, h{:}\,\alpha]$ from the start node s to the halt node h, with label α. Let u, v be nodes on F. Any path $[u, v]$ of G not in F is called a *back path* of F. A maximal union of forward paths of G such that the union is an acyclic schema is called a *forward path schema* (FPS) of G. Whilst the FPS of a structured schema is unique, it is not necessarily so for an unstructured one. For instance, referring to Fig. 3, the subschemas S_1 and S_2 obtained by deleting (c, d) and

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

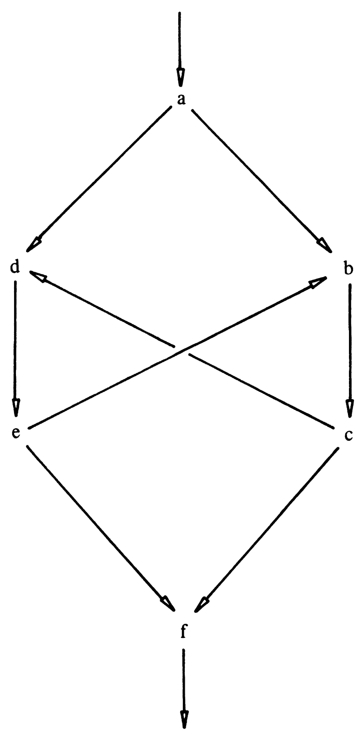

Figure 3. Schema with two possible FPSs.

(e, b) from G respectively are both FPSs of G, but their union is not, as it contains the cycle d-e-b-c-d. This situation arises because (c, d) is a back arc of S_1 but not of S_2, whereas the converse is true for (e, b). For the purposes of structuring it is immaterial which of possibly several FPSs is chosen as the one to work with.

An efficient algorithm for the recognition of an FPS is given below. Its time complexity is $0(e)$, where e is the number of arcs in the schema, since each arc of the schema is considered at most once.

Algorithm 1

Input: a schema $G(V, E)$, with start node s and halt node h.

Output: a forward path subschema $F(V_F, E_F)$ of G.

Method

Starting from some initial node u, a depth-first search is carried out until either a halt node v is found – in which case the depth-first path $[u, v]$ is accepted as an elementary path from u to v – or no further progress is possible. To improve computational efficiency the set of goal nodes is extended to include all those nodes already found to lie on some elementary path from u to v. The process is applied recursively beginning with $u = s$ and $v = h$.

Let U be the set of nodes whose membership of F is yet to be determined. Initially, $U = V - \{h\}$.

```
procedure forward_path (u:node);
  begin
  $U := U - \{u\}$;
  for all $x$ such that $x \in \Gamma u \cap U$ do
    forward_path ($x$);
  for all $x$ such that $x \in (\Gamma u - \{u\}) \cap V_F$ do
    $(V_F, E_F) := (V_F + \{u\}, E_F + \{(u, x)\}$
  end.
```

The desired forward path F is found from

```
  $V_F := \{h\}$;   $U := V - V_F$;
  forward_path ($s$).
```

A proof of correctness that the above algorithm returns a maximal forward path subschema of G can be found in Ref. 2.

It might be thought that a simpler method of finding an FPS would be to first identify the depth-first spanning tree of G and then partition the arcs of G into the four classes: tree arcs, forward arcs, cross arcs and back arcs.[3] Unfortunately this method can sometimes fail to recognise even the simplest of decision subschemas. Consider for instance a subschema comprising the forward path 1-4-2-3, the forward arc 1-2 (forming the decision subschema 1-4-2, 1-2), and back arc 3-4. A possible depth-first order of the nodes is 1, 2, 3, 4, from which 4-2 would be deduced as a back arc and the paths 1-2-3-4, 1-4 as forming a decision subschema! This conclusion is impossible with the given forward path algorithm when node 3 is nominated as the halt node.

An *augmented FPS* is defined as follows. Let u, v be nodes on the FPS F of a schema G such that there exist arcs $(u, w: a)$ and $(x, v: b)$ in G but not in F. If $w = v$ (or equivalently $x = u$ and $a = b$) then add two distinct arcs $(u, v': a_1)$ and $(u', v: a_2)$ to F such that $a_1 . a_2 = a$, otherwise add $(u, w: a)$ and $(x, v: b)$ to F. ($w = u$ or $x = v$ cannot arise in a reduced schema.) Applying the foregoing to all the nodes of F gives the augmented FPS A of G.

Structuring the augmented forward path subschemas

A *basic decision subschema* is a decision subschema that properly contains no other decision subschema. In order to structure an FPS F it is sufficient to repeatedly identify and structure the topologically last basic decision subschema D in F until F is fully structured.

Identification of D is straightforward. Let U be a topologically ordered set of decision nodes in the FPS F such that each node d of U has both of its immediate successors in F, and let u be the last member of the set U. Then u is the decision entry node of a basic decision subschema. The proof is immediate: suppose that S is a decision subschema with u as its decision entry node, but that S is not basic. Then contrary to assumption S must contain a decision node topologically later than u. To find the basic decision exit node and the two paths to it from u proceed as follows. Let the nodes of F be numbered in topological order, and let the number of any node v be $T(v)$. The required algorithm is:

Let v, w be the necessarily distinct immediate successors of u.

```
$V_D := \{u, v, w\}$; $E_D := \{(u, v), (u, w)\}$;
$x := v$; $y := w$;
while not $(x = y)$ do
  begin
  if $T(x) < T(y)$ then
    begin
    $z := V_F \cap \Gamma x$;
    $(V_D, E_D) := (V_D + z, E_D + (x, z))$;
    $x := z$;
    end
  else
    swap $(x, y)$;
  end;
```

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

(V_D, E_D) is the desired basic decision subschema, with entry node u and exit node x (equal to y) and is structured using the ID and OD transforms as given in section 2.

Simple loop subschemas

As already stated in section 2, a simple loop subschema comprises a single elementary path from the start node to the halt node together with a set of overlapping back arcs. This is the form to which all (sub)schemas are reduced after removal of all ID and OD constructs by the structuring algorithm to be given below. The IL to be considered first is found as follows. Let d be the first decision node on the forward path and let c be that immediate successor of d which also topologically precedes it, that is, let (d, c) be a back arc. Then c is the loop entry collector and the loop is $[c, d]\,(d, c)$. This loop comprises ILs alone and that which occurs at the immediate successor of c is the one sought. After structuring, the remaining ILs are found similarly.

The structuring algorithm for schemas

The desired structuring algorithm for an unstructured schema G with start node s and halt node h is given below. It is based on the requirement[1] that for the structuring transforms to be effective, that is, to always reduce the number of basic unstructured forms by at least one on each application, ID and OD constructs must be identified and removed ahead of IL constructs in each single-entry single-exit subschema being structured. The recognition and removal of unstructured forms is done recursively.

Algorithm 2

Input: a schema G.
Output: a structured schema in the form of a single arc labelled with a regular expression.

```
0.  If $G$ is not a single arc then goto step 1 else to step 7.
1.  Put $G$ into reduced standard form.
2.  Identify a forward-path subschema $F$ of $G$, and
    construct the augmented forward-path subschema $A$
    of $F$. ($A$ is acyclic.)
3.  Structure $A$ using the ID and OD transforms to give
    an elementary path $E$. (Typically $E$ will not be a single
    arc.)
4.  Set $G = (G-F) \cup E$;
5.  Let the nodes of $E$ be numbered in topological order
    and let each decision node on $E$ (except the start and
    halt nodes) be placed in an ordered set $U$.
    For each $d \in U$, in order of membership of $U$, do:
    recursively structure $G$ using $d$ as the start node and
    the immediate predecessor of $d$ in $E$ as the halt node.
    ($G$ will now comprise only IL and OL constructs at
    most, and every back path of $G$ will be a single arc.)
6.  Structure $G$ using the IL transforms.
7.  Stop. ($G$ now comprises a single arc with a label in the
    form of a regular expression, that is, $G$ is fully
    structured.)
```

The use of topological ordering in step 5 reduces the computational effort required to identify the FPSs in each recursive structuring of G. This is because the predecessors of d on E can only be collector nodes, and therefore the search for the FPS of G with start node d is necessarily restricted to those arcs reachable from the back path of E starting from d.

The proof of correctness for algorithm 2 proceeds by induction and is briefly sketched below.

Basis

1. Steps 1–4 reduce an acyclic schema to a single arc.
2. Steps 1–4 and step 6 reduce a schema whose back paths are all single arcs to a single arc.

Induction

Every subschema defined in step 5 is a proper subschema of that from which it is derived, hence there must exist some such subschema S of G which either has no back path or whose every back path is a back arc. The subschema H produced by structuring S is equivalent to G but has fewer arcs. Applying the argument repeatedly to H and its derivatives must therefore produce a structured schema of just one arc.

The application of algorithm 2 to some practical examples is given in the next section.

4. EXAMPLES

Three examples will be given. The first illustrates the application of the algorithm to a complex hypothetical schema whilst the remaining examples relate to two practical problems: the non-recursive post-order traversal of a binary tree, and the file merge problem. These latter examples will show that whilst the structuring algorithm is capable of producing well-designed equivalent structured programs under some conditions, further refinements which are dependent upon the particular interpretation of a schema may be possible too, and therefore beyond the scope of the present structuring algorithm.

Example 1. A complex hypothetical schema

Consider the hypothetical schema depicted in Fig. 4*a*. The arcs are left unlabelled, as the purpose of this example is to show the progressive steps in the structuring process rather than to produce an explicit equivalent regular expression. Writing algorithm 2 in the procedural form *structure* (s, h), where s and h denote start and halt nodes respectively, the progressive recursive calls on *structure* become:

```
      {Fig. 4a}
structure (0, 10);
      {see text below}
  structure (5, 2);
      {Fig. 4b}
  structure (13, 12);
      {Fig. 4c}
    structure (20, 17);
      {see text below}
    structure (5, 2);
      {Fig. 4d}
  structure (9, 13).
```

Fig. 4*a–d* illustrate the progressive forms of the schema immediately before the next call on *structure*, and show the FPSs about to be identified and structured, the FPS arcs being drawn as full lines and the remainder dashed.

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

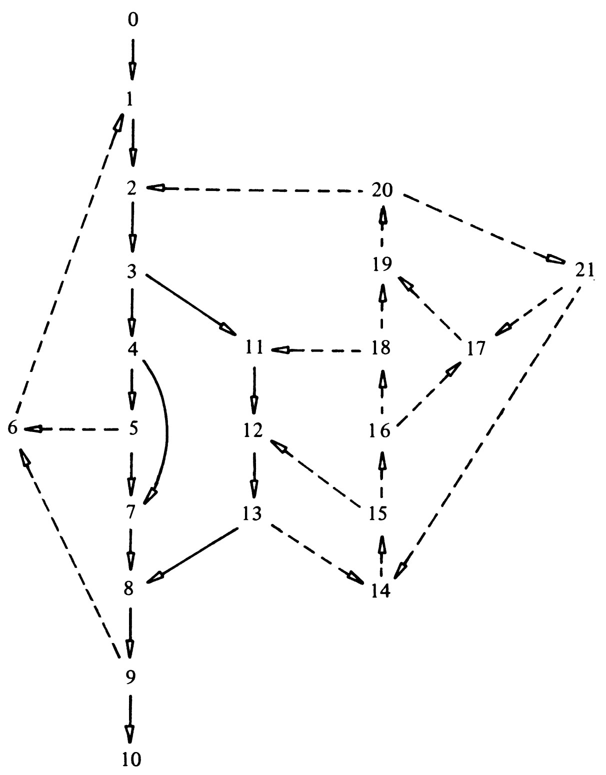

Figure 4*a*. A hypothetical schema *G*, showing its FPS.

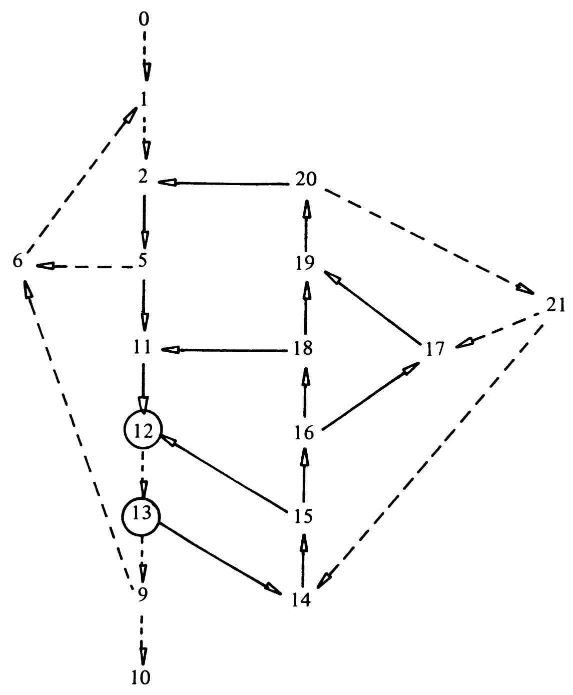

Figure 4*b*. *G*, after structuring its FPS.

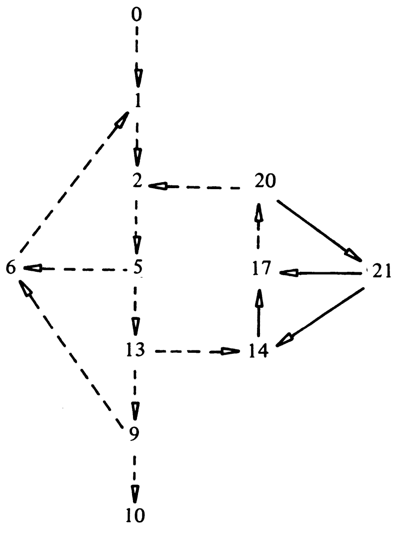

Figure 4*c*. *G*, in final stage of structuring subschema. [13, 12]

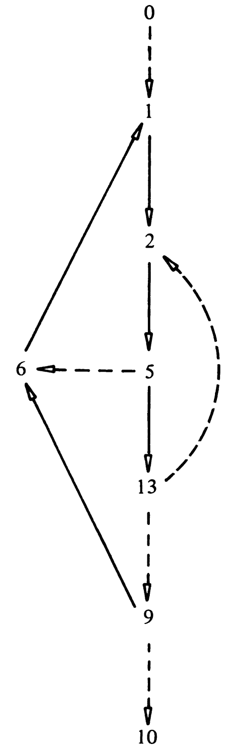

Figure 4*d*. *G*, after structuring subschema. [13, 12]

Following the structuring of the FPS in Fig. 4*a* the schema shown in Fig. 4*b* is obtained, for which the next step is *structure* (5, 2). As the FPS is simply the elementary path 5-6-1-2 it requires no further structuring. Following *structure* (20, 17) within *structure* (13, 12) the structured path 13-14-20-2 is obtained together with the structured cycle 20-14-20. Thus this construct reduces to the single arc (13, 2) when put into standard form. The next step in *structure* (13, 12) is *structure* (5, 2), which as already seen requires no further action. To improve computational efficiency it would be advantageous to note such steps on their first occurrence and avoid their subsequent recomputation. The FPS of Fig. 4*d* comprises the elementary path 9-6-1-2-5-13 with back arc (5, 6) and hence two IL constructs: the first at 1 from 0 and the second at 2 from 13. Structuring these yields the elementary path 0-1-2-13-9-10 within *structure* (0, 10) and two structured nested cycles whose back arcs are (13, 2) and (9, 1). As this is now fully structured the standard form reduces to a single arc and the process terminates.

Example 2. Non-recursive post-order tree traversal

An iterative solution to the post-order tree traversal

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

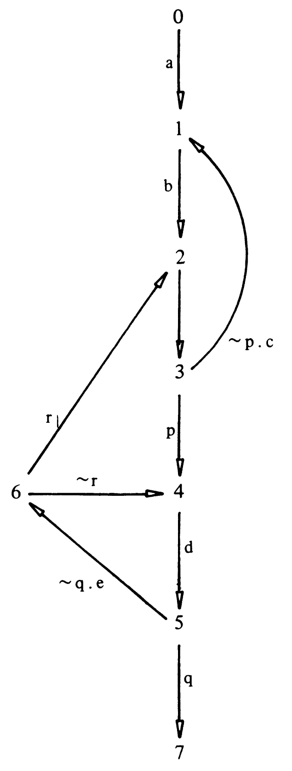

Figure 5*a*. Schema G_2, for iterative post-order traversal of binary tree.

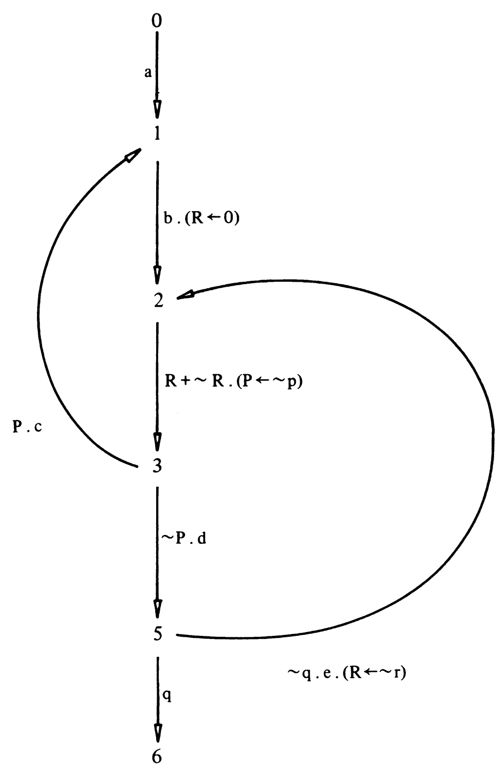

Figure 5*b*. G_2, structured to simple loop format.

problem is given below in terms of abstract data-type operations on the tree and a stack. In this and the next example statement groupings will be denoted by indentation rather than **begin–end** pairs.

```
   {t is the non-empty tree to be traversed}
   s := CreateStack;   {s is empty}
1: while not EmptyTree (left(t)) do   {go left in the tree}
      push (s, ⟨t, goright⟩);
      t := left(t);
2: if not EmptyTree (right(t)) then   {go right in the tree}
      push (s, ⟨t, goback⟩);
      t := right(t);
      goto 1;
3: process (root(t));
   if not EmptyStack (s) then   {go back up the tree}
      ⟨t, action⟩ := top(s);
      pop(s);
      if action = goright then
         goto 2
      else
         goto 3.
```

The corresponding schema is shown in Fig. 5*a* with the following particular interpretation.

```
a: s := CreateStack;
b: while not EmptyTree (left(t) do
      push (s, ⟨t, goright⟩);
      t := left(t);
c: push (s, ⟨t, goback⟩); t := right(t);
d: process (root(t));
e: ⟨t, action⟩ := top (s); pop(s);
p: EmptyTree (right(t))
q: EmptyStack (s)
r: action = goright.
```

Arc (2, 3) represents an empty basic block.

The solution is obtained by the steps

```
   {Figure 5a}
structure (0, 7);   {FPS is the elementary path [0, 7]}
   structure (3, 1); {FPS is the single arc (3, 1) – no action needed}
   structure (5, 4);   {FPS has an OD at 3 to 1 and an ID at 2 from 1. Apply OD-1 and ID-1 respectively}
   {Fig. 5b}   {G has IL at 2 from 5 – apply IL-1}
```

to yield the regular expression

$$a.(P := \text{true}).[\{P.b.(R := \text{false}) + \sim P.e.(R := r)).(R + \sim R.(P := \sim p)).P.c\}^+.\sim P.d.(P := \text{false}).\sim q]^+.q$$

The term ($P := \text{false}$) is clearly redundant and can be omitted. Writing DoRightTree for $\sim R$ and EmptyRight for $\sim P$, the interpretation for the regular expression is the structured program

```
s := CreateStack;
EmptyRight := false;
repeat   {until the tree is fully traversed}
   repeat   {until empty right subtree encountered}
      if not EmptyRight then
         while not EmptyTree (left(t)) do   {go left in the tree}
            push (s, ⟨t, goright⟩);
            t := left(t);
         DoRightTree := true
      else   {go back up the tree}
         ⟨t, action⟩ := top(s);
         pop(s);
         DoRightTree := action = goright;
      if DoRightTree then
         EmptyRight := EmptyTree (right(t));
```

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

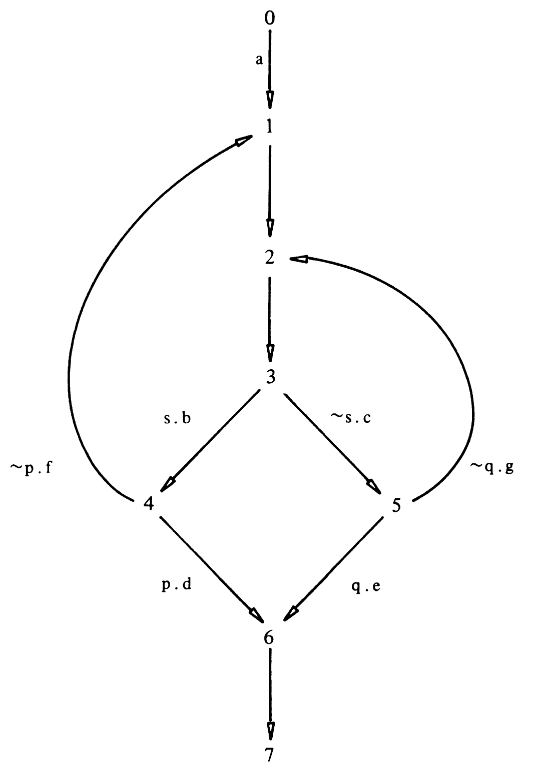

Figure 6. Schema for merging sorted files.

```
    if not EmptyRight then   {go right in the tree}
      push (s, ⟨t, goback⟩);
      t := right(t);
  until EmptyRight;
  process (root(t));
until EmptyStack (s).
```

Apart from the slightly misleading setting of EmptyRight to false in the second line (which can be overcome by writing RightTreeDone for EmptyRight throughout) this form of the algorithm is a clear re-statement of the original, and was obtained with the minimum of post-structuring manipulation (the elision of the term $P := \text{false}$).

The reader is invited to repeat the structuring process using ID-0 and IL-0 in place of ID-1 and IL-1 respectively.

Example 3. The file merge problem

Let l and r be two files arranged in ascending sorted order which are to be merged into an ascending-order sorted file f. The following abstract data-type operations on files are assumed to be available.

eof (f): return true if f is empty, false otherwise;
get (f): if not eof (f) then return $\langle g, i\rangle$ where i is the first item in f and g is the remainder of f, else undefined;
put (f, i): append item i to f and return the updated file;
copy (f, g): return the composite file $f.g$

A relevant schema is depicted in Fig. 6 for which the necessary particular interpretation is

a: $\langle l, u\rangle := \text{get}(l)$; $\langle r, v\rangle := \text{get}(r)$
b: $f := \text{put}(f, u)$
c: $f := \text{put}(f, v)$
d: $f := \text{copy}(\text{put}(f, v), r)$
e: $f := \text{copy}(\text{put}(f, u), l)$
f: $\langle l, u\rangle := \text{get}(l)$
g: $\langle r, v\rangle := \text{get}(r)$
p: $\text{eof}(l)$
q: $\text{eof}(r)$
s: $u \leqslant v$

Since the arcs (1, 2) and (2, 3) represent empty basic blocks the schema is computationally completely symmetric as the target node of nodes 4 and 5 is in reality node 3. Clearly all that is required to structure the schema is the elimination of the two instances of OD: one at 4 to 1 and the other at 5 to 2. Choosing 4-1 first followed by 5-2 gives

$$a.[\{(s.b.(P := \sim p).(P.f + \sim P.d).(Q := \text{false}) + \sim s.c.(Q := \sim q).(Q.g + \sim Q.e.(P := \text{false}))).Q\}^{+}.\sim Q.P]^{+}.\sim P$$

which clearly lacks the symmetry of the original. Flow analysis for the flag P reveals that $(Q.g + \sim Q.e.(P := \text{false}))$ is computationally equivalent to $(Q.g + \sim Q.e).(P := \text{false})$, which makes the body of the loops symmetric, but still leaves the asymmetry of one loop nested within the other. Further flow analysis shows that it is possible to merge the two loops into one – a process that is completely beyond the structuring algorithm – by the introduction of a third predicate flag R. This is possible because both loops have the same target node for their back arcs, no computation is performed on the back arcs, and the condition that neither back arc is traversed is simply $\sim P$ **and** $\sim Q$. The regular expression now takes the form

$$a.[\{s.b.(P := \sim p).(R := \sim P).(P.f + \sim P.d) + \sim s.c.(Q := \sim q).(R := \sim Q).(Q.g + \sim Q.e)\}.\sim R]^{+}.R$$

One further possible refinement remains. It is now no longer necessary to introduce P and Q as distinct flags: their role can be undertaken by R as they are mutually independent variables. Thus the final structured form is

$$a.[\{s.b.(R := p).(R.d + \sim R.f) + \sim s.c.(R := q).(R.e + \sim R.g)\}.\sim R]^{+}.R$$

which is in the desired symmetric format.

Interpreting R as 'Finished', the final algorithm is

```
⟨l, u⟩ := get(l);   ⟨r, v⟩ := get(r);

repeat
  if u ⩽ v then
    f := put (f, u);
    Finished := eof(l);
    if Finished then
      f := copy (put(f, v), r)
    else
      ⟨l, u⟩ := get(l)
  else
    f := put(f, v);
    Finished := eof(r);
    if Finished then
      f := copy (put(f, u), l)
    else
      ⟨r, v⟩ := get(r)
until Finished.
```

It is clear that this form of the algorithm could not have been found by Algorithm 2 alone as it is heavily dependent upon the properties of the particular interpretation of the original schema. None the less, the

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024

structuring algorithm provided the basis from which to start, and this being structured made subsequent flow analysis easier.

5. DISCUSSION

Whilst it has been shown that it is always possible to put an unstructured schema into structured form, it remains to be shown that the process is computationally tractable for large schemas. This is now taken up, and it will be shown that the time complexity is almost always linear in the number or arcs in the schema, and at worst is asymptotically linear from above.

Let e be the number of arcs and v the number of nodes in a schema G. Then for all G $e \geqslant v+1$, so it is appropriate to use e as the measure of size for the problem. Referring to algorithm 2, section 3, it can be shown that each of the operations involved in steps 1–4 and step 6 is of time complexity $0(e)$. For step 5 suppose that the maximum number of subschemas generated is a, and that the maximum number of arcs in any such subschema is b. The time complexity $T(e)$ of Algorithm 2 is given by

$$T(1) = c$$
$$T(e) \leqslant a\,.\,T(b)+c\,.\,e$$

where c is some constant.

Writing $b = e/n$, the solution for $T(e)$ *is* $0(e ** \log_n a)$ for $a > n$, $0(e)$ for $a < n$, and $0(e\,.\log_n e)$ for $a = n$.[3] Consider first $a > n$. This could arise, for instance, if G has a single forward path of three arcs and a back path subschema of e-3 arcs. If this pattern is repeated over all subschemas generated in step 5 of Algorithm 2 then $a = 1$ and $n = e/(e-3)$, so that although $T(e)$ is exponential in e it is asymptotically $0(e)$ for large e. Needless to say, schemas of such a form are highly unlikely in any practical situation. To achieve $a \geqslant n$ in general will require similarly contrived and highly improbable forms, and it is conjectured that these too will be asymptotically of $0(e)$ from above. For any practical program it is reasonable to expect that at each level of recursion in step 5 the sum total of arcs considered is less than the total in the (sub)schema from which they are derived. That is, $a\,.\,b < e$ and hence $a < n$. It is claimed therefore that for almost all schemas $T(e)$ is $0(e)$, and in any case for all schemas is at worst exponential in e with an exponent greater than but asymptotically approaching unity for large e.

Since every arc in a schema must be considered at least once, it is clear that algorithm 2 is very nearly optimal. It is easy to see that the space complexity is similarly $0(e)$.

When schemas contain ID and IL constructs the question arises as to whether type 0 or type 1 transforms should be used, that is, whether it is better to duplicate an arc (representing a basic block) or to duplicate the associated decision node through a predicate flag. Predicate duplication has the disadvantage of introducing decisions simply to force the flow of control into structured frameworks, with no guarantee that the resulting code is as clear or clearer than the original. Arc duplication on the other hand preserves the original logic more closely but brings the burden of extra code. This can be minimised for large basic blocks by converting them to subroutines and replacing them in the structured code by subroutine calls, but for small basic blocks the overheads of the subroutine call have to be balanced against the overheads of space duplication. If execution time is the overriding concern then the type 1 transforms will almost always be preferred, unless the basic block that would otherwise be duplicated is very small. (One could well ask: if execution time is critical, why structure at all if the original code was well designed for time efficiency?)

With the overwhelming approbation accorded to structured programming the presentation of an algorithm to automatically translate programs to structured form would seem to need no apology, however, a few words of caution regarding its use are in order. Because structuring involves forcing the flow of control into newly created paths there is no guarantee that the logic of the structured program will be any more intelligible than the original – structuredness and lucidity are not synonymous terms! Further, it must be remembered that structuredness is achieved only at a price, namely space and time overheads. None the less, as shown by the examples in section 4, used as a design aid rather than as a panacea the structuring algorithm can assist in the generation of clear well-formed programs by providing a soundly based structured schema from which further refinements peculiar to particular interpretations can then more easily be made.

In conclusion, a fully algorithmic and practical solution to the problem of recasting unstructured schemas into equivalent structured form has been presented and shown to run in time linearly related to the number of arcs in the schemas.

REFERENCES

1. G. Oulsnam, Unravelling unstructured programs. *The Computer Journal* **25**, 379–387 (1982).
2. G. Oulsnam, Untangling unstructured programs. Ph.D. Thesis, National University of Ireland, University College, Cork, Republic of Ireland (1984).
3. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts (1983).

Downloaded from https://academic.oup.com/comjnl/article/30/1/43/407792 by guest on 10 April 2024