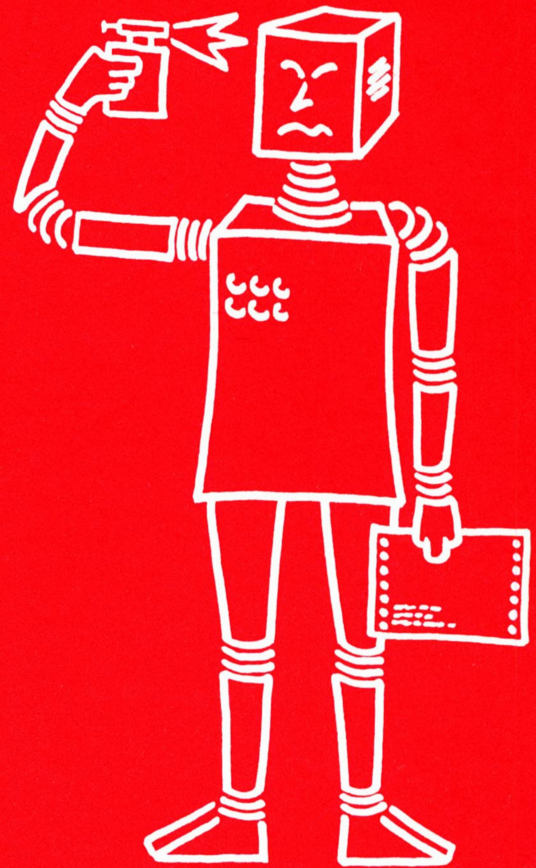
SEKI-PROJEKT

SEKI MEMO

Institut für Informatik III
Universität Bonn
Bertha-von-Suttner-Platz 6
D 5300 Bonn 1, W. Germany

Institut für Informatik I
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1, W. Germany



Memo SEKI-BN-81-06

SYNTHESIZING MINIMAL PROGRAMS FROM TRACES OF OBSERVABLE BEHAVIOUR

Christoph Beierle

Synthesizing minimal programs from traces of observable behaviour

Christoph Beierle

Institut für Informatik III
Universität Bonn
Bertha-von-Suttner-Platz 6
D-5300 Bonn 1 (West Germany)

Abstract

Automatic synthesis of non-recursive flowchart programs from traces of observable behaviour is investigated. Our program synthesis algorithm described here can be applied to sets of sequences of stores yielding minimal programs being capable of reproducing these sequences. An efficient decision procedure for solvability of program synthesis problems is presented. An extension of PA admits four different types of input traces. For all four types program synthesis remains NP-complete even under various constraints.

Keywords and Phrases: automatic program synthesis, example computation, identification in the limit, NP-completeness, observable behaviour, store traces, program synthesis problem, program synthesis algorithm

1. Introduction

Different approaches to program synthesis from example computations have been investigated (c.f.[Bi 76],[Bib 78],[Ha 75], [JK 81], [SSG 75],[Su 77]). Here we are interested in synthesizing non-recursive flow-chart programs from observable changes a computing device undergoes when given some input. As an extension to [BK 76] where only instruction sequences together with information on the executed tests are considered, the input to our system are traces of store descriptions with no indication where a test might have been executed. In section 2 we give some example input our algorithm is capable to deal with.

One of the problems we have to cope with is combinatorial explosion. This stems from the fact that three different types of hypothesis have to be made for every store transition in the input:

(1) What instruction I has caused the transition?
(2) What tests had been executed before I was executed?
(3) What other instructions can be merged with I (i.e. which loops can be constructed)?

Section 3 gives an overview over the methods, strategies and heuristics we use to choose these hypotheses appropriately.

The synthesis algorithm given in section 4 has been proved to be **correct** , **minimal** and **complete** in the following sense:

(1) the synthesized program P' has the same observable behaviour as given by the input,
(2) there is no shorter program (w.r.t. the number of assignments) with this property,
(3) for every infinite enumeration of all computation sequences of the observed programm P there is some finite k, such that after having seen the first k traces of the enumeration the synthesis algorithm will synthesize a program P' that is observably equivalent to P; i.e. every P will be identified in the limit [Go 67].

Some extensions of the synthesis algorithm are mentioned in section 5. In [Be 80] the problem of program synthesis is shown to be NP-complete; in section 6 several subcases of the general problem are given which are still NP-complete.

2. Store traces

Suppose we are given a computing device with four registers r1, r2, r3, r4 (and no extra storage). Each of the registers can hold an integer. The contents of the registers is displayed permanently and can be seen while there is no way to look directly at the program the calculator executes. The instruction set of the machine is given by (where $i,j \in \{1,..,4\}$):

assignments:

```
        IC(i):   ri:= ri + 1
        DC(i):   ri:= ri - 1
        A(i,j):  ri:= ri + rj
        S(i,j):  ri:= ri - rj
```

tests:

LZ(i): **true** iff $ri \leq 0$

The program P (fig.1) can be executed by this machine. P terminates for all input-quadrupels over the integers; for r3 initially greater then zero it computes a generalized version of McCarthy's 91-function [Ma 74], namely:

f(r1) = **if** r1>r2 **then** r1-r3 **else** f(f(r1+r3+1))

which for r2=100 and r3=10 becomes identical to the original 91-function. P computes the least fixpoint of f (provided r3>0):

fix.f = λr1. **if** r1>r2 **then** r1-r3 **else** r2-r3+1

Every program node with instruction I has a **label** $l \in \mathbb{N}$ such that (l,I) identifies exactly one node; thus, (1,A(1,2)) and (2,A((1,2)) are the distinct instances of A(1,2) in P (fig.1).

A **store trace** $\langle s_1,...,s_n \rangle$ of program P and input store s is obtained by protocolling the observable behavior of P when looking at the store: s_1 is equal to s, and each time an assignment is made to the store s_i the resulting store is

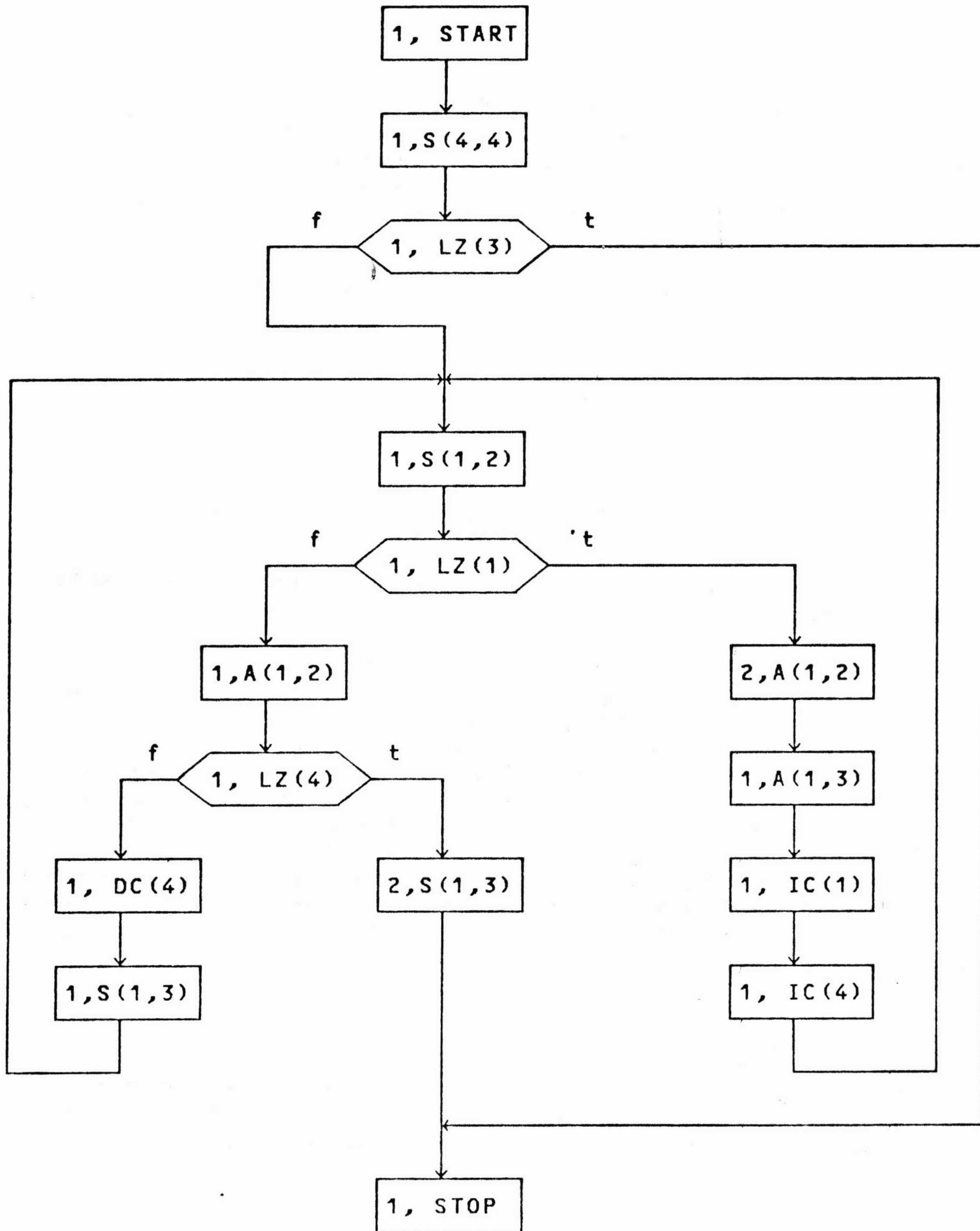


fig. 1: program P

given by s_{i+1}.

<u>Example</u>:

The store trace of P and $s_1 = (1,1,1,1)$ is:

$$S_1 = \langle(1,1,1,1),(1,1,1,0),(0,1,1,0),(1,1,1,0),(2,1,1,0),\\ (3,1,1,0),(3,1,1,1),(2,1,1,1),(3,1,1,1),(3,1,1,0),\\ (2,1,1,0),(1,1,1,0),(2,1,1,0),(1,1,1,0)\rangle$$

For P and $s_2 = (2,1,2,1)$, $s_3 = (2,1,0,1)$, $s_4 = (-3,-3,4,2)$, and $s_5 = (4,2,2,2)$ we have:

$$S_2 = \langle(2,1,2,1),(2,1,2,0),(1,1,2,0),(2,1,2,0),(0,1,2,0)\rangle$$

$$S_3 = \langle(2,1,0,1),(2,1,0,0)$$

$$S_4 = \langle(-3,-3,4,2),(-3,-3,4,0),(0,-3,4,0),(-3,-3,4,0),\\ (1,-3,4,0),(2,-3,4,0),(2,-3,4,1),(5,-3,4,1),\\ (2,-3,4,1),(2,-3,4,0),(-2,-3,4,0),(1,-3,4,0),\\ (-2,-3,4,0),(-6,-3,4,0)\rangle$$

$$S_5 = \langle(4,2,2,2),(4,2,2,0),(2,2,2,0),(4,2,2,0),(2,2,2,0)\rangle$$

Such a set of traces will be called a **program synthesis problem** or **psp** for short.

3. Program synthesis from store traces

The program synthesis algorithm we will develop is divided into two phases: a static and a dynamic phase. During the static phase some kind of preprosessing is done yielding a normal form for program synthesis problems which will be called IH-graph. No backup will be nessecarry during this phase while during the dynamic search in the second phase backups will be possible.

3.1. Phase 1: Static prepocessing

The static phase can be seen as being devided into four successive steps. Although in an actual implementation these steps are interwined they will be described seperately.

3.1.1. Step 1: Compute matrix representation

The first step towards a hypothesis which instruction I has caused a particular store transition is to compute the set of all instructions that are capable of doing so. For store trace S_i the sequence MI_i gives this information, namely: $MI_{i,j} = \{I \in \text{assignments} \mid I(s_{i,j}) = s_{i,j+1}\}$. Since we consider complete and terminating traces, MI_i starts with {START} and ends with {STOP}.

<u>Example</u>: For store traces S_2 and S_5 we get:

$$MI_2 = \langle\{START\},\left\{\begin{matrix}DC(4),\\ S(4,2),\\ S(4,4)\end{matrix}\right\},\left\{\begin{matrix}DC(1),\\ S(1,2)\end{matrix}\right\},\left\{\begin{matrix}IC(1),\\ A(1,1),\\ A(1,2)\end{matrix}\right\},\left\{\begin{matrix}S(1,1),\\ S(1,3)\end{matrix}\right\},\{STOP\}\rangle$$

$$MI_5 = <\{START\}, \begin{Bmatrix} S(4,2), \\ S(4,3), \\ S(4,4) \end{Bmatrix}, \begin{Bmatrix} S(1,2), \\ S(1,3) \end{Bmatrix}, \begin{Bmatrix} A(1,1), \\ A(1,2), \\ A(1,3) \end{Bmatrix}, \begin{Bmatrix} S(1,2), \\ S(1,3) \end{Bmatrix}, \{STOP\}>$$

Looking at MI_i independently from other traces gives us $|MI.(i,0)| \bullet ... \bullet |MI.(i,n_i)|$ different possibilities to choose exactly one instruction from each set; for both MI_2 and MI_5 above we get $3 \bullet 2 \bullet 3 \bullet 2 = 36$ possibilities, yielding $36 \bullet 36 = 1296$ combinations for these two traces alone.

The second matrix which is computed in this step is the matrix of tests, MT. $MT_{i,j}$ gives the set of all tests which could have been executed successfully (yielding **true**) before $I \in MI_{i,j+1}$ was applied:

$MT_{i,j}=\{T \in tests \mid T(s_{i,j+1})= \mathbf{true}\}$

<u>Example</u>: For traces S_2 and S_5 we get:

$MT_2 = < \{\},\{LZ(4)\},\{LZ(4)\},\{LZ(4)\},\{LZ(1),LZ(4)\}>$
$MT_5 = < \{\},\{LZ(4)\},\{LZ(4)\},\{LZ(4)\},\{LZ(4)\}>$

<u>3.1.2. Step 2: Compute level graph representation</u>

Since every program has exactly one START-node every store trace originates from a sequence of instructions beginning with <u>START</u>. Since $MT_{2,0}=MT_{5,0}$, the first assignment node in the two respective instruction sequences must be identical as well because only tests can introduce a branch (i.e. an **if-then-else** construct) and there is no test which distinguishes the respective stores, namely $s_{2,1}$ and $s_{5,1}$.

In general, this observation leads to a partition of every row of MI induced by the following equivalence relation =e= on the index-pairs of MI:

$(i_1,j) =e= (i_2,j)$ **iff** $\forall l \in \{0,...,j-1\}.\ MT.(i_1,l)=MT.(i_2,l)$

Following the above argumentation $MI_{i_1,j}$ and $MI_{i_2,j}$ can be merged if $(i_1,j) =e= (i_2,j)$; only instructions I both in $MI_{i_1,j}$ and $MI_{i_2,j}$ may be selected as a hypothesis and only the same I for both at a time.

<u>Example</u>: For MI_2 and MI_5 above we have
$(2,j) =e= (5,j)$ for $j = 0,1,2,3,4$. Thus, merging MI_2 and MI_5 yields:

$$<\{START\}, \begin{Bmatrix} S(4,2), \\ S(4,4) \end{Bmatrix}, \{S(1,2)\}, \begin{Bmatrix} A(1,1), \\ A(1,2) \end{Bmatrix}, \{S(1,3)\}, \begin{cases} \{STOP\}> \\ \{STOP\}> \end{cases}$$

Now we are left with only $2 \bullet 2 = 4$ possibilities to choose exactly one instruction from each set for both MI_2 and MI_5.

In general the relation =e= induces the **level graph representation** LGR of MI and MT. The nodes of LGR are of the form $K=[l,\{i_1,...,i_K\}]$ where l is the **level** and $\{i_1,...,i_K\}$ the **index set** of K. K represents the merging of $MI.(i_1,l),...,MI.(i_K,l)$ and is labelled with the intersection of all these sets; all nodes of level l together represent

column l of matrix MI. The edges of LGR are labelled with the respective MT-entries defining the partitioning.

The complete level graph representation for the above five store traces $S_1,\ldots,S_5$ is given in fig.2.

3.1.3. Step 3: Reduce level graph representation

If a node K in LGR is labelled with the empty set, no instruction hypothesis is available for K and the node may be deleted from LGR. Furthermore, all hypotheses necessarily leading to K may be deleted as well. This process could eventually cause the whole graph to vanish indicating that there is no program capable of reproducing the given traces (see step 4 below). When dealing with store traces, LGR will always be a tree where it suffices to check that no node is labelled with the empty set. In [Be 80] program synthesis from other types of traces is investigated where the LGR may not be a tree and where the reduction and the deletion process is more complicated.

A second reduction step minimizes the index sets of the nodes. Whenever two indizes j and j' always appear together in the index sets, one of them is redundant and may be deleted.

Example: Step 3 leaves the above level graph representation (fig. 2) unchanged. If there was another store trace S_6=<(2,3,0,2),(2,3,0,0)>, indizes 3 or 6 would be redundant because MT_6=<{LZ(3)},{LZ(3),LZ(4)}>=MT_3. However, their redundancy appears only in the level graph representation: MI_6=<{START},{S(4,1),S(4,4)},{STOP}> would cause node [1,{3}] to be labelled with {S(4,4)}, thus leaving only one possible instruction hypothesis for that node.

3.1.4. Step 4: Apply realizability test

At this stage, it is easy to decide whether the synthesis process can be completed successfully or not.

Theorem: LGR is empty **iff** there is no P ∈ PROGRAMS with an observable behaviour as given by the input traces [Be 80].

Thus, the dynamic phase of the synthesis algorithm will only be entered if it is garanteed to succeed. In the following, we will therefore consider only realizable sets of traces.

The output of the preprocessing phase describes all relevant instruction hypotheses that have to be considered in the second phase. Thus, the output graph is called **instruction hypothesis graph (IH-graph)** of the given psp.

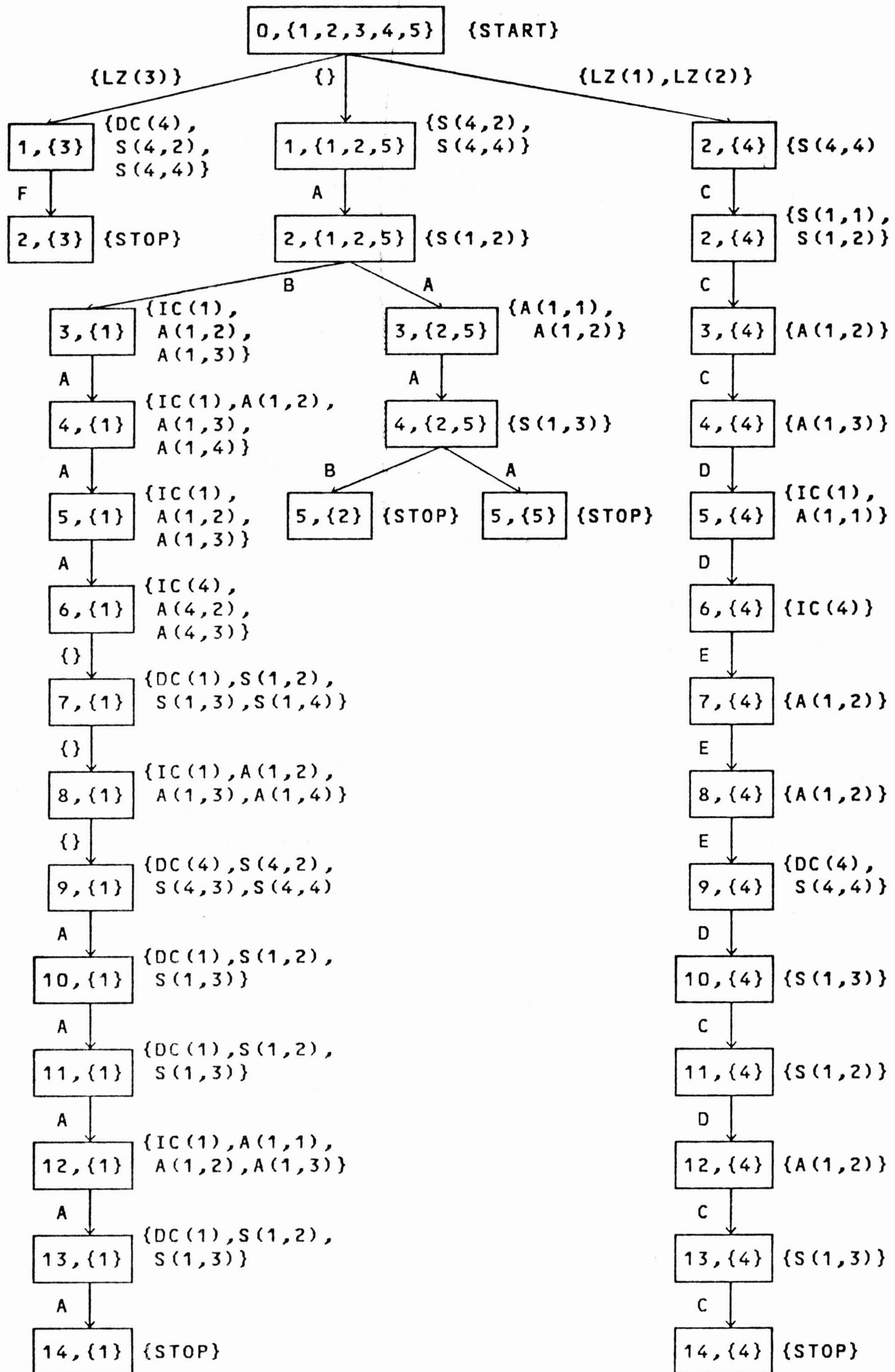


fig. 2: IH-graph G (sets A - F as in fig. 4)

3.2. Phase 2: Dynamic searching

3.2.1 Assignment approximations

The major task of the second phase is to generate an **assignment approximation** of the final program. An assignment approximation AP of a program P is a set of triples (v_1,m,v_2), where the $v_i=(l_i,I_i)$ are the assignment nodes of P and m is a set of tests each of which lies on a path via a **true** - branch from v_1 to v_2.

Example: Assignment approximation of P (fig. 1) is given in fig. 3 in its straightforward graph representation, where, e.g., ([1,S(1,2)],{LZ(1)},[2,A(1,2)]) is written as

```
                    {LZ(1)}
   [1,S(1,2)]  ----------->  [2,A(1,2)]  .
```

In order to construct from IH-graph G an assignment approximation, two different types of hypothesis have to be set up for each node K in G:

1. select an **instruction hypothesis** a(K)
2. select a **label hypothesis** u(K)

Let UG(a,u) be the set of all triples (v_1,m,v_2) such that $v_i=[u(K_i),a(K_i)]$, where the K_i are the nodes of G and there is an edge from K_1 to K_2 marked with m.

Example: (c.f. fig. 2) For K_1=[11,{1}], K_2=[12,{1}], we can choose $a(K_1)$=S(1,2), $a(K_2)$=A(1,2), with $u(K_1)=u(K_2)=1$, so that UG(a,u) contains the triple ([1,S(1,2)],{LZ(4)},{1,A(1,2)]).

UG(a,u) has the **program property** iff for all triples we have unique transitions from one node to the other in the following sense: for every triple $t=(v_1,m,v_2)$, $t'=(v_1',m',v_2')$, if $v_1=v_1'$ and m=m' then t and t' are identical, i.e. $v_2=v_2'$ as well.

Theorem: [Be 80]

(1) UG(a,u) is an assignment approximation of some program P' **iff** UG has the program property.

(2) If UG(a,u) has the program property, it can be transformed effectively into a program P' that reproduces all store traces of the given synthesis problem. Moreover, this transformation does not introduce any additional assignment nodes.

(3) If there is a program that reproduces all given store traces and where the number of assignment nodes is minimal, then there exists instruction and label hypotheses a,u for the IH-graph such that UG(a,u) has the same number of assignment nodes and UG(a,u) has the program property.

This theorem justifies the following strategy: first, find some hypotheses a,u such that UG(a,u) has the program property and the number of nodes in UG(a,u) is minimal, in a second step transform UG(a,u) into a program.

Example: For every IH-graph G there are trivial hyptheses a,u such that UG(a,u) has the program property: take any of the possible instructions for a and choose u in such a way that no two distinct nodes of G will be merged (except for STOP-nodes

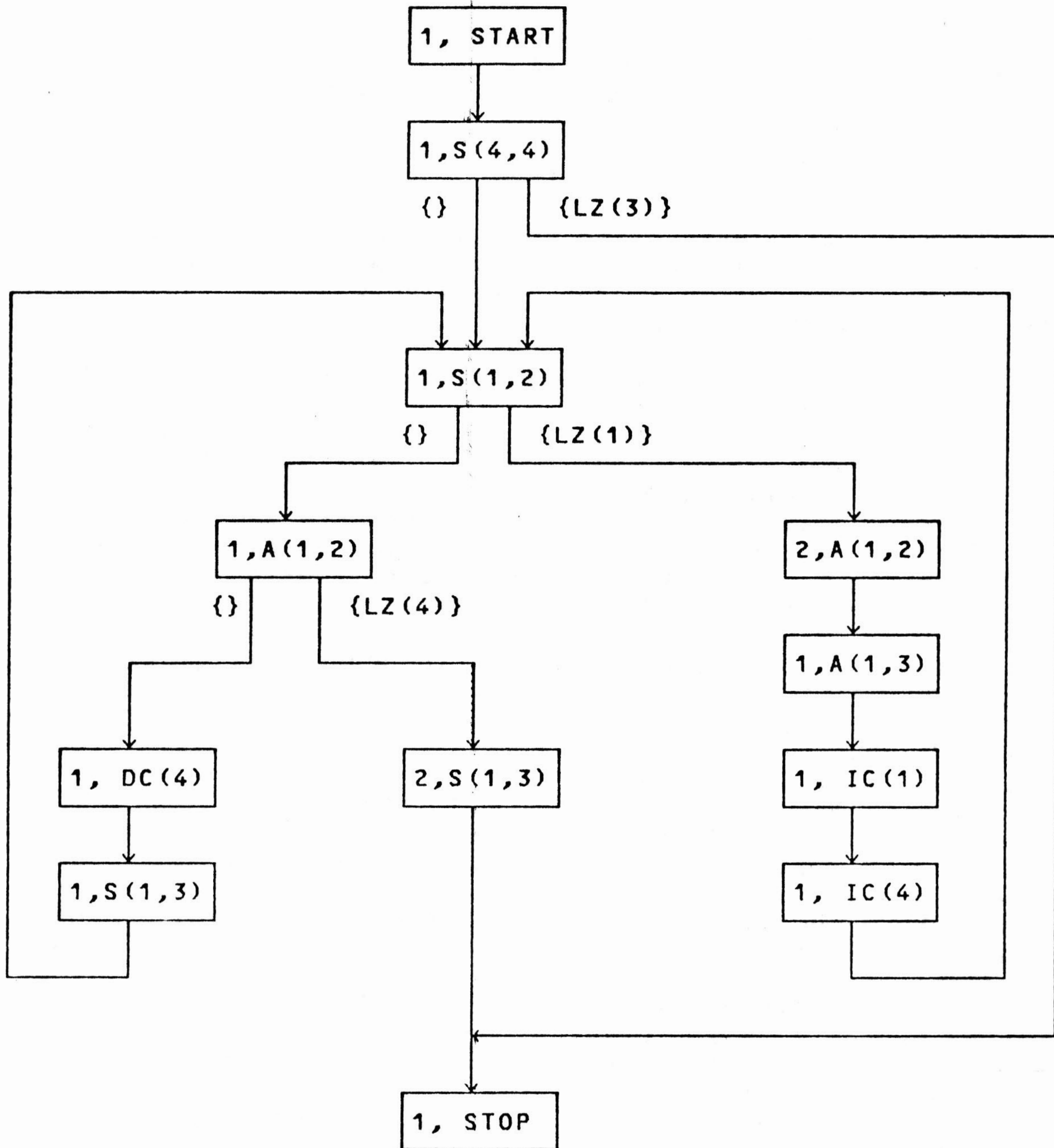


fig. 3: Assignment approximation of P

where the label hypothesis will always be 1). However, UG(a,u) will then be purely sequential, and the number of assignment nodes will be large. Therefore, we have to merge as many nodes of G as possible, or in other words, we want to construct **loops**.

3.2.2 Constructing branches and loops

The selection of instruction and label hypotheses determine the branches and loops of the synthesized program. Basically, the hypotheses are chosen in the following way:

(i) Start with the single node K at level 0 and set (u(K),a(K)):=(I,START)

(ii) As long as there are some nodes unvisited, do the following:
For all nodes $(K_1,...,K_n)$ - called the new **frontier** - which are direct descendants of the nodes considered in the previous step choose $a(K_i)$ and $u(K_i)$ such that UG(a,u) still has the program property and that the number of different nodes in UG(a,u) is minimal. When necessary, go back to the previous frontier and change some hypothesis u(K) and/or a(K).

This process will be explained on the IH-graph G (fig. 2).

1. Starting with K=[0,{1,2,3,4,5}}], (u(K),a(K)) is set to (I,START); c.f. fig 4.1.
2. The new frontier is (K_1,K_2,K_2) = ([1{3}],[1{1,2,5}],[1,{4}]). All K_i are merged by setting $(u(K_i),a(K_i))$ = (1,S(4,4)), i=1,2,3 (fig 4.2).
3. The new frontier is (K_1,K_2,K_3) = ([2,{3}],[2,{1,2,5}],[2,{4}]). For K_1 there is only one possible hypothesis, namely u(K),a(K)=(I,STOP). K_2 and K_3 cannot be merged with any previous nodes; they are merged together by setting $(u(K_i),a(K_i))$=(1,S(1,2)), i=1,2 (fig. 4.3).
4. None of the nodes (K_1,K_2,K_3)=([3,{1}],[3,{2,5}],[3,{4}]) can be merged with any previous ones; it is possible, however, to merge all three with $(u(K_i),a(K_i))$=(1,A(1,2)), i=1,2,3 (fig. 4.4)
5. The new frontier poses a problem: since [3,{1}] and [3,{2,5}] were merged in step 4, their direct descendants [4,{1}] and [4,{2,5}] must be merged as well because (a) the corresponding edges in G are both marked with the same set of tests, {LZ(4)}; and (b) we require UG(a,u) to have the program property. However, there is no instruction hypothesis for both [4,{1}] and [4,{2,5}], so no merge of the two nodes is possible. Thus, a back-up to step 4 is necessary leading to dismerge [3,{13}] and [3,{2,5}] which will cause the insertion of a test after (1,S(1,2)) (fig. 4.5).
6. Suppose the frontier has already been pushed to (K_1,K_2) = ([7,{1}],[7,{4}]) where UG(a,u) is as in fig. 4.6. It is possible to merge both K_1 and K_2 with an already existing node in UG(a,u) by setting $(u(K_i),a(K_i))$=(1,S(1,2)), i=1,2, thus leading to the construction of the loop indicated in fig. 4.6 by the dotted lines.

3.2.3 Transforming assignment approximations into programs

Let a,u be the hypotheses for G (fig.2) that yield UG(a,u) as

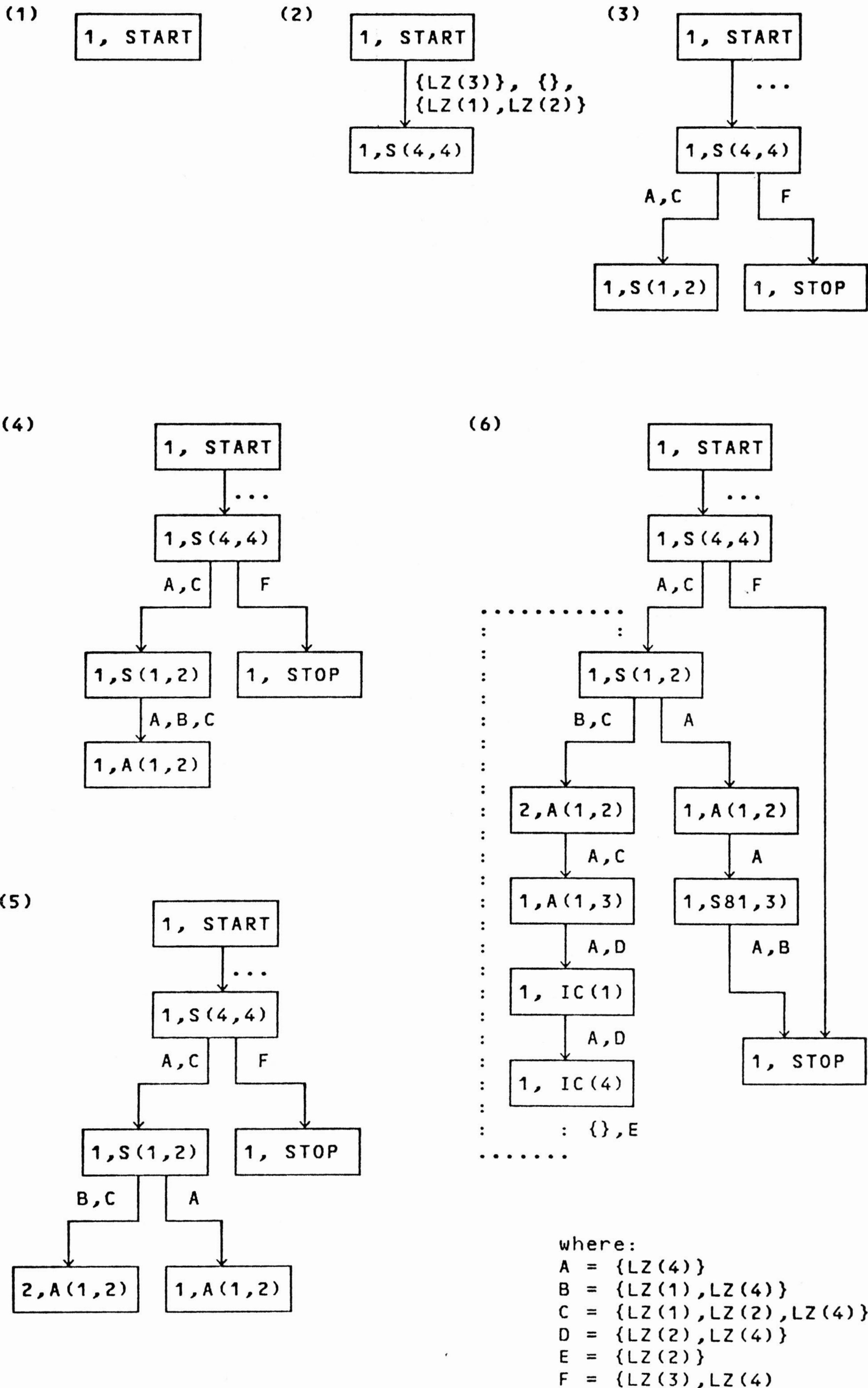


fig. 4: construction of branches and loops

in fig.5. In order to transform UG(a,u) into a program any multiple outgoing edges from assignment nodes have to be eliminated. For each node K we look at the outgoing edges. If there are two, we have to insert a text distinguishing the two edges, if there are more than two, we have to introduce a number of tests distinguishing all of them. Since UG(a,u) has the program property we are garanteed to find such tests. In fig. 5 there are three nodes with out-gegree 2. For (1,S(4,4)) the only distinguishing test is LZ(3); the result of inserting this test is shown in fig. 6. Similarly, tests LZ(1) after (1,S(1,2)) and LZ(4) after (1,A(1,2)) have to be inserted the result of which is identical to P (fig. 1).

4. The program synthesis algorithm PA

With the above definitions, PA works as follows:

1. M:={set of store traces} given as input.
2. G:= instruction hypothesis graph computed from M.
3. If G is empty, print "no program exists that reproduces the input traces" and stop.
4. L:= minimal number of assignment nodes that are needed at least for any successful hypotheses a,u.
5. Try to find hypotheses a,u such that UG(a,u) has at most L nodes.
6. If no successful hypotheses were found increment L by 1 and go back to 5.
7. Transform UG(a,u) into a program P, print P and stop.

PA is garanteed to terminate since for non-empty G there are always trivial hypotheses a,u (c.f. example in section 3.2.1).

One of the crucial points in PA is the selection of hypotheses a,u in step 4 which is done in a fashion sketched in section 3.2.2. However, now the value L plays a central role: at any time, only hypotheses inducing UG(A,u) to have L or fewer assignments are of interest.

4.1 Distinguishability

A powerful method to reduce the number of relevant hypotheses in advance is based on the distinguishability of nodes in the IH-graph. K and K' are **distinguishable** ($K \# K'$) iff they can never be merged in UG(a,u). # can be computed as follows:

$K \#_0 K'$ iff there is no instruction hypothesis for both K and K'

$K \#_{i+1} K'$ iff $K \#_i K'$ or there exists edges $(K,K_1),(K',K_1')$ in G both marked with the same set of tests and $K_1 \#_i K_1'$

is equivalent to the first $\#_i$ such that $\#_i = \#_{i+1}$.

Example: For G (fig. 2) we have:
[5,{1}] $\#_0$ [6,{1}], [4,{1}] $\#_1$ [5,{1}], [3,{1}] $\#_2$ [4,{1}], [3,{1}] $\#_1$ [3,{2,5}]

For two distinguishable nodes K,K' PA will never set up identical hypotheses a and u for both K,K'. Thus, PA will actually avoid the unsuccessful merging of [3,{1}] and [3{2,5}] that

1, START
{LZ(3)},{},{LZ(1),LZ(2)}
1,S(4,4)
A,C
{LZ(3),LZ(4)}
1,S(1,2)
{},A,D,E
B,C
1,A(1,2)
2,A(1,2)
{},E
A,C
A,C
1,A(1,3)
A,D
1, DC(4)
2,S(1,3)
1, IC(1)
A,D
A,B,C
A,D
1,S(1,3)
1, IC(4)
A,C
{},E
1, STOP

fig. 5: Assignment approximation UG(a,u) (A - F as in fig. 4)

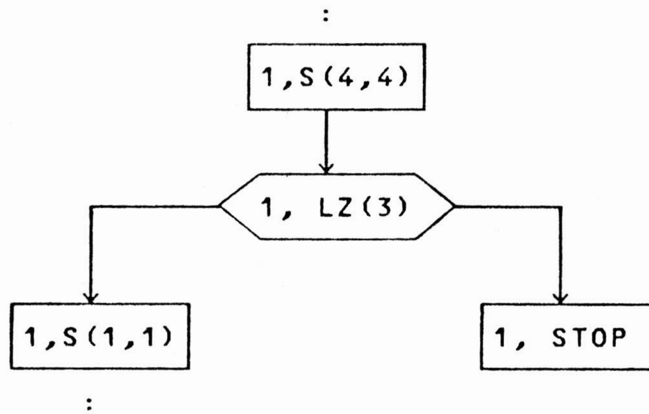


fig. 6: construction of a test node

was tried in section 3.2.2 (4); we merely used this merging as an example to illustrate the back-up process.

4.2 Lower bounds on hypotheses

Another search reducing method is to compute a lower bound Lmin on the number of nodes that will be needed for successful hypotheses. Thus, in step 4 of PA L is set to the maximum number of pairwise distinguishable nodes. Furthermore, when looking for the hypotheses for the new frontier, it is useful to know a lower bound Lmin(I) of how many instances of each particular assignment I will be needed. Lmin(I) is set to the maximum number of pairwise distinguishable nodes for which the only instruction hypothesis is I.

Example: For G (fig. 2) we have:
Lmin(I) = 1 for I = S(1,2),S(4,4),A(1,3),IC(4).
Lmin(S(1,3)) = 2 because [10,{4}] $*_1$ [13,{4}], and
Lmin(A(1,2)) = 2 because [3,{4}] $*_1$ [12,{4}].
Lmin = 9 since e.g. [3,{1}], [4,{1}], [5,{1}], [12,{1}], [2,{1,2,5}], [10,{4}], [13{4}], [6,{1}], [1,{1,2,5}] are all pairwise distinguishable.

The example shows that Lmin may be greater than the sum over all Lmin(I).

The failure avoiding techniques that use the * -relation and the Lmin-values bear some resemblence to the concept of failure memory in [BBP 75]. However, the algorithm described there works on a sequence of instructions, while PA works on a set of traces in parallel with an additional hypothesis dimension since PA's input consists of store traces.

5. Extensions of PA

PA has been designed to deal with a number of different types of traces. It is applicable (1) to store traces as described above, (2) to store traces where the new store is given after each assignment and each text execution, (3) to instruction sequences where only the assignments are given and (4) to instruction sequences where both assignments and tests are given. For all four types, PA is correct, minimal and complete (c.f. section 1).

6. NP-completeness of program synthesis from traces

For all four types of traces program synthesis is NP-complete, even under several constraints. For instance, if the instruction set of the observed machine contains only five assignments and one test, the decision problem whether for a set of traces M and a $k \in \mathbb{N}$ there exists a program of length k that reproduces all traces in M, is NP-complete even if M contains only one single trace [Be 80]. It is interesting to note that from this NP-completeness point of view program synthesis both from instruction and store traces has the same complexity.

References:

[Be 80] Beierle,C.: Programmsynthese aus Beispielsfolgen, Memo SEKI-BN-81-04 (Dec.80), Inst. f. Informatik III, Univ. Bonn

[Bib 78] Bibel,W.: On strategies for the synthesis of algorithms, Proc. AISB Meeting, Hamburg, 1978

[Bi 76] Biermann,A.W.: Approaches to automatic programming, in: Advances in Computers, Yovits and Rubinoff (Eds.), New York, 1976

[BBP 75] Biermann,A.W., Baum, Petry: Speeding up the synthesis of Programs from Traces, IEEE Trans. Comp., vol. C-24, 1975

[BK 76] Biermann,A.W., Krishnaswamy, R.: Constructing Programs from Example Computations, IEEE Trans.Soft.Eng., vol. SE-2, no 3, 1976

[Ha 75] Hardy,J.: Synthesis of LISP-functions from examples, 4th IJCAI, 1975

[JK 81] Jouannaud,J.P., Kodratoff,Y.: Program synthesis from examples of behaviour , NATO Adv. Study Inst. on Automatic Program Construction, Oct.1981

[Go 67] Gold,E.M.: Language identification in the limit, Information and Control 10, 1967

[Ma 74] Manna,Z.: Mathematical Theorie of Computation, McGraw-Hill, New York, 1974

[SSG 75] Shaw,D., Swartout,W., Green,C.: Inferring LISP programs from examples, 4th IJCAI, 1975

[Su 77] Summers,P.D.: A methodology for LISP program construction from examples, JACM no. 24, 1977