

AN ESSAY ON TREES AND ITERATION

W.P. de Roever

RUU-CS-78-6

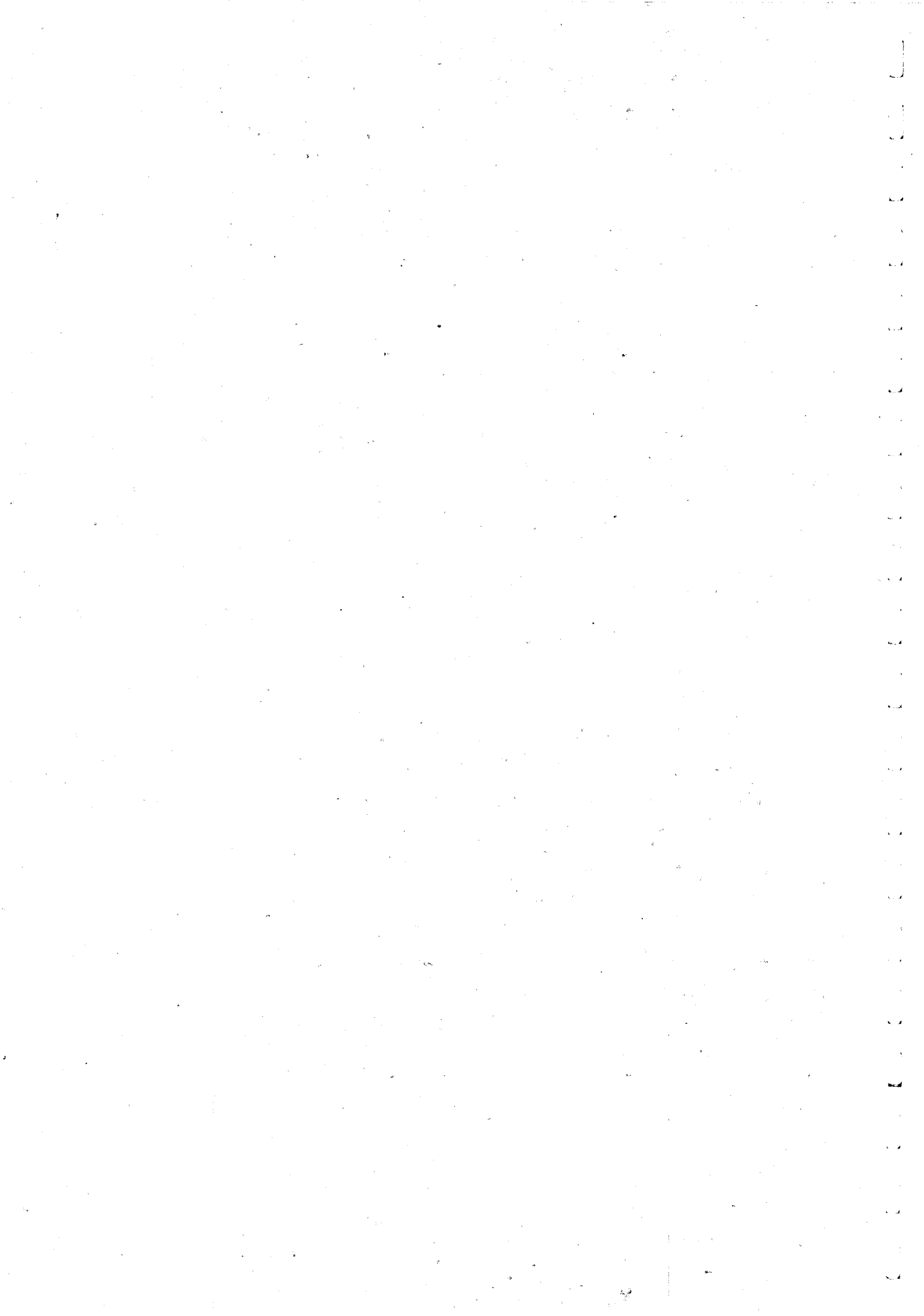
October 1978



Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030-531454
The Netherlands



1978/79

AN ESSAY ON TREES AND ITERATION

W.P. de Roever *)

Technical Report RUU-CS-78-6

October 1978

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht, the Netherlands

*) The major part of this work was prepared when the author was affiliated with the Computer Science Division, University of California, Berkeley, CA 94720.

This essay is based on lecture notes for CS 192: "Introduction to program correctness", taught by the author at UC Berkeley in the Spring of 1978. All comments and suggestions on these notes should be sent to

Dr. W.P. de Roever
Vakgroep Informatica
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
The Netherlands

<u>Table of contents</u>	pp.
1. Preface	1
2. Summary of the essay	3
Objectives of the essay	3
Technical tools	3
Suggestion for the outline of a course	4
AN ESSAY ON TREES AND ITERATION:	
3. Iterative traversal of a binary tree without auxiliary stack	6
Symbolic execution	13
Critique	13
4. An informal encounter with mathematical semantics	15
5. On correctness of backtracking (, and iterative tree-copying)	21
Example: 4-queen's problem recursive solution	22
And now for something altogether different? Iterative copying of binary trees	29
6. Correctness of the recursive characterization of backtracking for all solutions: a special case	30
7. The general case: correctness of a recursive problem specification	34
8. The Deutsch-Schorr-Waite marking algorithm: first step	40
The Deutsch-Schorr-Waite marking algorithm: second step	43
9. Termination of $M(\alpha, \sigma)$ - and, hence, of the Deutsch-Schorr- Waite marking algorithm	47
10. An exercise in Scott induction	55
11. References	60

1. PREFACE

The status of program correctness within the general theory of programming is as yet undetermined.

- Does program correctness constitute a bag of tricks used to convince oneself and others that a program "does what it is supposed to do", tricks which have been applied previously in mostly trivial cases? Certainly, considering most of the present day publications on the subject, this point of view has a right of existence.
- Or is program correctness part of mathematical semantics of programming languages - a subject which by the abstraction and complexity of its techniques is presently in danger of undergoing the fate of the proverbial hot-air balloon (, well known in the theory of computation, ridden as it is by fads and fashion,) disappearing out of sight of the main body of programming until the proliferation of its abstractions cools its air sufficiently to result in a quiet landing, not noticed by anybody except a small number of specialists?
- Or is program correctness part of programming in that it supplies the framework and concepts for reasoning about the execution of algorithms in an imaginative way, to the extent that the intricacy of their execution becomes transparent and communicable?

In this essay I shall try to emphasize the third point of view, focussing on iterative tree search as the guiding example. More in particular by discussing in depth a number of intricate algorithms concerning tree traversal, back-tracking, and graph traversal.

First I provide models for understanding these algorithms intuitively. Then I show how these intuitions can be made precise by introducing assertions. Finally I prove these assertions.

This process is not trivial at all. Appropriate formalisms have to be found. Once a formalism has been chosen, the assertions one intends to prove usually have to be generalized because they often do not provide enough information about the intermediate steps of execution of the algorithm to yield sufficiently strong inductive hypotheses.

New assertions have to be found and incorporated, since the inductive nature of program proofs requires that all inductive hypotheses "have to cooperate in step/phase". Previously overlooked "obvious" properties of algorithms start playing a crucial role in these generalizations.

Mutually dependent modules have to be characterized both in their individual contributions and in their interrelationship; finding these modules is often the main clue to understanding an algorithm.

Finally one wishes to push the developed concepts to their extremes, e.g., by investigating the possible merits of a "general back-tracking" algorithm or by obtaining an understanding of lower bounds on the auxiliary space needed for search. Sometimes one tries to tackle and maybe discover new algorithms extending the developed techniques.

These notes have been written to shed at least an entertaining light on the problems of program proving, within the wider context of getting acquainted with a fascinating collection of algorithms, which is partly still under development.

One note of warning: supplementary reading of Burstall's paper on structural induction, and an introductory paper on least fixed points such as the one by Manna, Ness and Vuillemin, is advised.

2. SUMMARY OF THE ESSAY

Objectives of the essay

The objective of the essay is to demonstrate the relevance of program proving to programming. To this end correctness of some nontrivial algorithms will be proved. These algorithms are taken from the area of iterative processing of tree-structures, or tree-structured computations.

(2) concurrent garbage collection.

This does not necessarily imply that the proofs themselves are considered to be the main focus of attention. Rather, the process of formulation and getting acquainted with an algorithm, and of developing concepts in order to better understand the way an algorithm functions, are considered to be the points of main importance.

Technical tools

The main technical tool in program correctness proofs is induction. The use of induction is either explicit, or implicit. Explicit induction proceeds either

(1) on the structure of the inputs presented

or

(2) on the recursion-depth of a program (in case it is recursive), or on the number of times iteration is performed (in case it is iterative), or on the length of computation.

In case (1) one speaks of structural induction of some sorts (Burstall). In case (2) the induction principle takes two forms:

(2a) n-step-n+1 induction scheme: if (A1) $A(0)$ can be proved,
and (A2) if one assumes $A(n)$ as hypothesis,
 then $A(n+1)$ can be proved,
then(A3) $\forall n. A(n)$ holds.

(2b) course-of-values induction scheme (also called truncation induction, Morris): if (B1) $A(0)$ can be proved,
and (B2) for every m , if for some $0 \leq m_1 < m_2 \dots < m_k \leq m$
 one assumes $A(m_1) \wedge \dots \wedge A(m_k)$ as hypothesis,
 then $A(m+1)$ can be proved,
then (B3) $\forall m. A(m)$ holds.

These two forms are equivalent:

(2b) \Rightarrow (2a): obvious.

(2a) \Rightarrow (2b): considered to be evident from:

introduce $A'(m) = A(0) \wedge \dots \wedge A(m-1) \wedge A(m)$;

then prove $A(m+1)$ using scheme (2a) on $A'(m+1)$.

Implicit induction is used to ascertain validity of proof methods using invariants (Floyd, Hoare).

In ordinary sequential programming both implicit and explicit induction are successful techniques. In concurrent programming (Gries-Owicki) the invariant method has been most successful.

Suggestion for the outline of a course *)

1. The method of invariants as introduced by Floyd-Hoare, and the generalization to intermittent assertions to embody termination (Burstall).
2. Explicit inductive methods will be demonstrated in the discussion of a family of algorithms concerned with iterative processing of tree-structures in general, and with tree traversal, backtracking, traversal of directed graphs, and list copying, in particular.

The members of this family have a simple idea in common, according to which the stack of return-links is coded directly into the data-structure traversed.

(a) This idea is first applied to ordinary traversal of a given tree (Dwyer e.a.).

(b) Then it is generalized to backtracking. This can be understood as follows:

In backtracking the search-space has a tree-structure. The search-space is generated during the computation. In fact, the process of searching for solutions itself can be defined most easily by tree-structured computations. The backtracking problem is to perform these computations in iterative fashion.

Obviously any technique for iterative traversal of a given tree in memory can be generalized to iterative processing of these tree-structured computations, i.e., to backtracking.

Also, we shall discuss what "correctness of backtracking" means.

*) See page i.

(c) The idea is applied to the traversal of directed graphs.

In order to traverse a directed graph one constructs spanning trees for the graph, to eliminate infinite traversal of cycles. The spanning trees are constructed during the process of traversal of the graph, using the unifying idea referred to above (Deutsch-Schorr-Waite).

(d) Finally these ideas have been generalized to copying list-structures (Lee-de Roever-Gerhart).

(In the future one could envisage an encyclopedia of algorithms, in which algorithms are referenced by and developed from the central ideas which they embody/employ. It would be interesting to determine these semantic bricks of programming and to find systematic ways of combining them!)

3. Program transformations (Burstall & Darlington, Gerhart, Lee-de Roever-Gerhart).

Postscript: Stanley Lee tackled the problem of proving correctness of three recently published list-copying algorithms (Robson, Fisher, Clark). As it turned out, the formalization of the notion of subproof, or step in a proof, illustrated rather implicitly in these notes, had to be brought out explicitly by introducing program transformations when dealing with correctness of state-of-the-art algorithms. The reason for this phenomenon is that the chain of reasoning leading up to such algorithms becomes too long and complicated to rely on intuition as a source of inspiration (for finding their correctness proofs). We found support in Susan Gerhart's work, which introduced a sufficiently general notion of transformation to provide the kind of steps and the framework required in proofs of such complexity; this is the subject of the paper "On the evolution of list-copying algorithms", POPL '79, which can be considered as a straightforward sequel to these notes.

4. Concurrent program proving.

Focus will be on the Gries-Owicki method and Dijkstra's view of it, including a proof of the correctness of Dekker's implementation of semaphores, and, finally, a proof of a concurrent garbage collector. The latter proof is interesting in that it concerns an algorithm for which the degree of interleaving of the various concurrently cooperating processes is so intricate that proving its correctness turns out to be the only way to obtain a correct formulation.

AN ESSAY ON TREES AND ITERATION

3. Iterative traversal of a binary tree without auxiliary stack

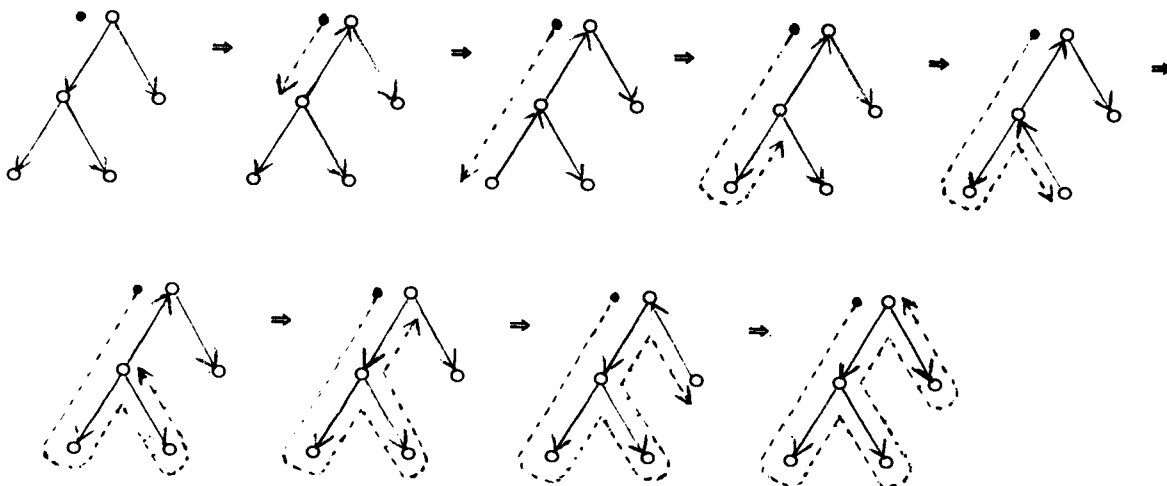
In this section we discuss iterative traversal of binary trees without using an auxiliary stack. (This feature makes the algorithm useful in a garbage collection environment.)

Consider the following pointer reversal game (due to Meertens):

Given a (binary) tree whose links to subtrees are indicated by arrows. Traverse this tree, visiting nodes according to the following rules:

- (i) The game begins at the root of the tree.
- (ii) Any arrow from the current node to another can be traversed; thereafter it must be reversed.
- (iii) The game ends at the root of the tree.
- (iv) Each node of the tree must be visited at least once.

In pictures:



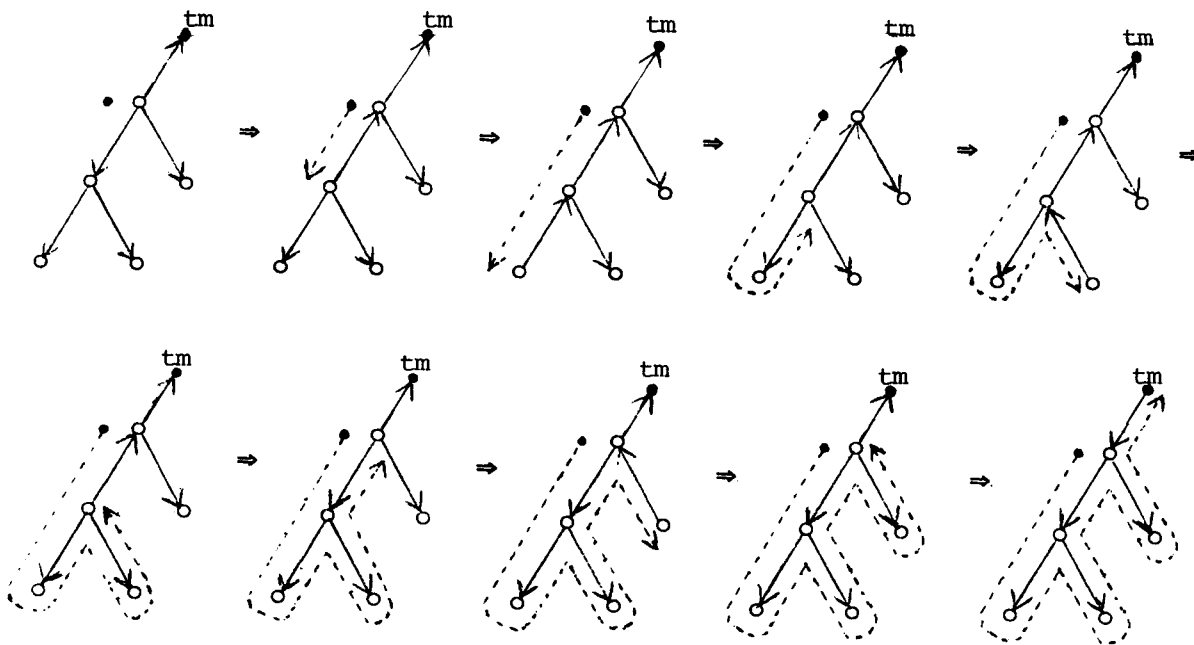
Note in the intermediate snapshots that the branching structure of the original tree is changed. In the last snapshot the branching structure has miraculously recuperated. Why?

Every arrow is traversed twice. Therefore it changes direction twice. This results in the same direction for all arrows at the end.

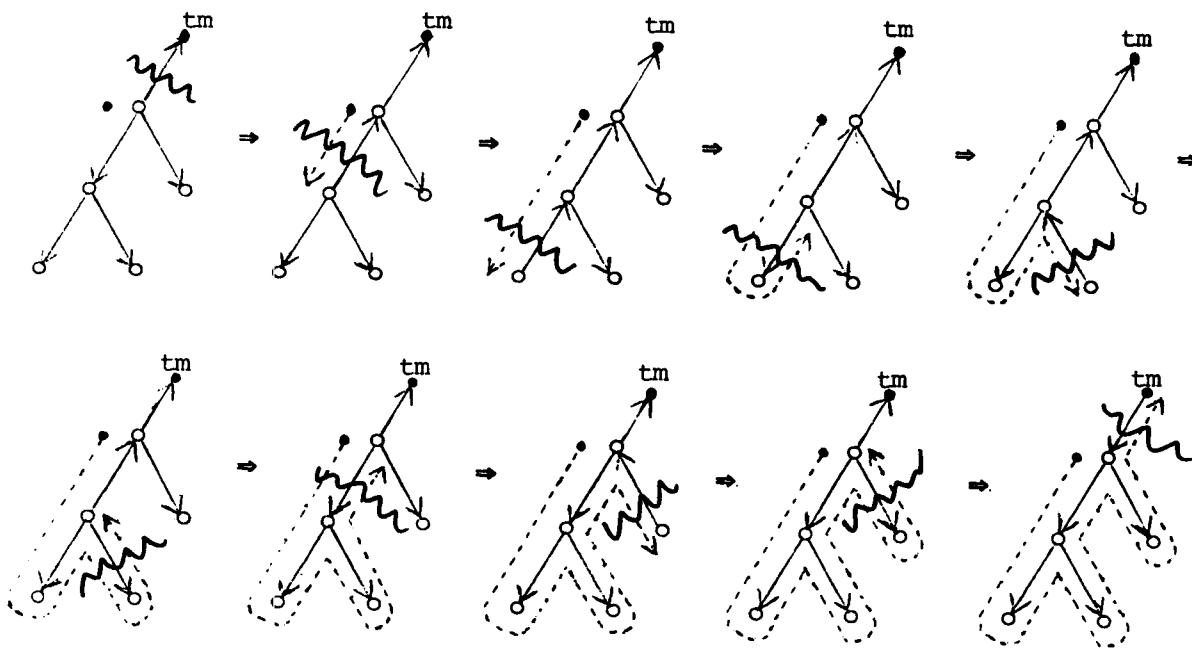
This simple observation is a central ingredient of the correctness of all the algorithms discussed in this essay.

Our next goal is to formulate the traversal-reversal game in algebraic notation so that it could be processed on a computer. To this end several observations/modifications are needed, because we intend to describe the algorithm, and therefore the intermediate stages of the drawing above, in terms of binary trees. But, clearly, ternary nodes appear in the snapshots above.

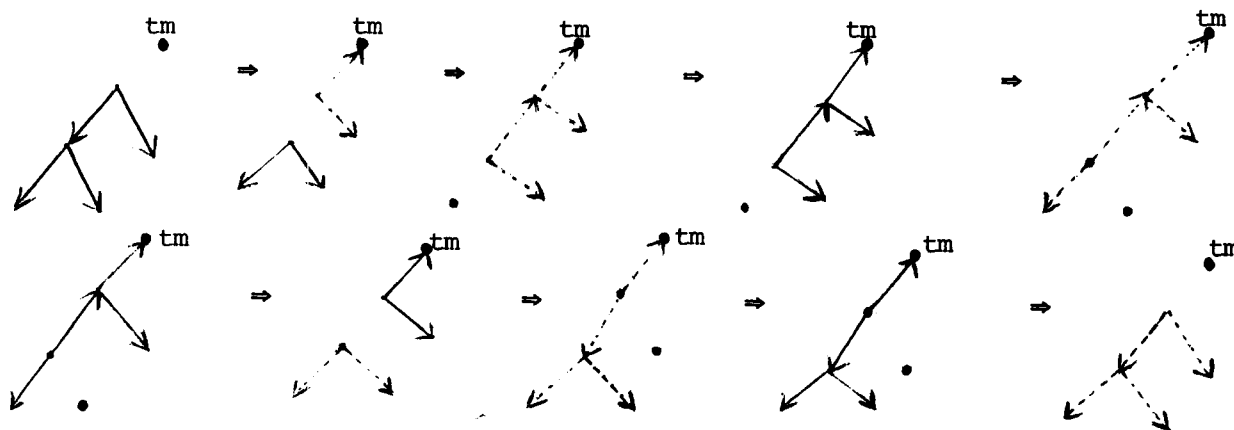
Observation 1. The algorithm has to end. In order to accomplish this, introduce a "termination marker" tm - an atom - which, when inspected, terminates the algorithm. This involves (a) a change of the atomic acts of the game, (b) a change of rule (iii). The termination marker is connected with the arrow structure as indicated below:



Observation 2. As remarked before, ternary nodes appear in the drawing above. Yet we want to represent/describe these snapshots by means of binary trees. The observation needed is that each of these snapshots can be represented by a pair of binary trees, by splitting each snapshot at the arrow most recently traversed. (By definition, in the first snapshot at the left upper corner of the drawing, the arrow most recently traversed is the one added after observation 1 above.) This is indicated below:



Observation 3. The representation of snapshots as two binary trees makes use of an ordered pair of binary trees, since always one of the two trees is traversed/inspected first:



(In this drawing, the tree which is traversed first has been drawn using straight lines and the other by using dotted lines.)

Next, a notation is required, and a characterization of the set of trees which must be traversed.

DEFINITION (binary trees).

Let A be an arbitrary set (of atoms). Then \mathcal{B} is the smallest collection (of binary trees) s.t.

(1) $A \subseteq B$

(2) if $l_1, l_2 \in B$ then $\text{cons}(l_1, l_2) \in B$,

where $\text{cons}: B \times B \xrightarrow{\text{onto}} (B-A)$ denotes a total, 1-1 function mapping pairs of trees onto (non-atomic) trees.

DEFINITION (car , cdr , at).

$\text{car}: (B-A) \rightarrow B$, $\text{cdr}: (B-A) \rightarrow B$ are functions which are (one-sided) inverses to cons satisfying:

$l \notin A \rightarrow l = \text{cons}(\text{car}(l), \text{cdr}(l))$, $l_1 = \text{car}(\text{cons}(l_1, l_2))$,

$l_2 = \text{cdr}(\text{cons}(l_1, l_2))$,

$\text{at}: B \rightarrow \{\text{true}, \text{false}\}$ is the characteristic predicate of $A \subseteq B$.

The reason for defining B as the smallest collection satisfying the properties listed above is that we want to restrict ourselves to finite trees only, since these are the only ones naturally represented in memory. (Representing directed graphs by means of infinite trees, where cycles are unfolded infinitely deep, is undesirable, since changing a pointer in the graph would result in an infinite number of changes in the infinite tree.)

(Note for the specialist: the definition of binary trees as given above is still unsatisfactory. The abstract data type of binary trees should be defined within a category-oriented framework as an initial object.)

Now the algorithm can be formulated. Intermediate stages are transformed as follows:

$\neg \text{at}(l): \langle l, r \rangle \Rightarrow \langle \text{car}(l), \text{cons}(\text{cdr}(l), r) \rangle$

$\text{at}(l): \langle l, r \rangle \Rightarrow \text{if } l = \text{tm} \text{ then } r \text{ else } \langle r, A(l) \rangle \text{ fi}$

The last line requires some explanation.

The act of inspecting an atom is performed by the constant A (; think of A as coloring an atom yellow, or of increasing a counter "inside" the atom without side-effect upon the other atoms or nodes of the structure).

Also, upon ending, the inspected/traversed tree is given as answer.

Note that all the preceding observations can be generalized to trees of arbitrary arity.

For purposes of proof, the algorithm is formulated in the following (pseudo-) recursive version, where it is assumed that the implementation of recursion does not require an auxiliary stack:

$Q(l, r) \leftarrow \text{if } \text{at}(l) \text{ then}$
 $\text{if } l = \text{tm} \text{ then } r \text{ else } Q(r, A(l)) \text{ fi}$
 $\text{else } Q(\text{car}(l), \text{cons}(\text{cdr}(l), r)) \text{ fi.}$

Note that, since the formalism introduced above does not incorporate any notion of address, one of the essential characteristics of the algorithm has not been explicitly described, namely that the stack of return-addresses is coded inside the tree itself. In fact, an implementation of cons which fetches a new address for each application, is legitimate. Yet the simple formulation above is useful for the purpose of proof analysis and as an exercise in abstraction. Later on a more elaborate formalism will be introduced which does capture the coding aspect.

What constitutes the correctness of algorithm Q?

Assuming that tm does not occur among the atoms of l, "execution of Q(l,tm) should lead to traversal of all of l". How can this be formulated in a more formal way?

Introducing the recursive procedure P(l):

$$P(l) \Leftarrow \underline{\text{if}} \text{ at}(l) \underline{\text{then}} A(l) \underline{\text{else}} \text{ cons}(P(\text{car}(l)), P(\text{cdr}(l))) \underline{\text{fi}}.$$

This recursive procedure is the only tool available to specify the effect of Q independent of the text of Q itself.

Correctness of Q is expressed by

$$\forall l \in \mathcal{B}. (\text{tm} \notin l \rightarrow Q(l, \text{tm}) = P(l)), \dots \quad (3.1)$$

here " \rightarrow " denotes implication, and $\text{tm} \notin l$ that tm does not occur among the atoms of l (; this can be expressed recursively, too).

We shall prove (3.1), after surmounting some initial difficulties, by induction on the complexity of l, i.e., by structural induction upon l. For binary trees, this principle can be formulated as follows:

Structural induction on binary trees

Let $As(l)$ denote an assertion on binary trees.

In order to prove $\forall l \in \mathcal{B}. As(l)$, one proves first (i) $As(l)$ for $l \in A$, i.e., one proves As for all atoms, and then one proves, for arbitrary $l_1, l_2 \in \mathcal{B}$, (ii)

$$As(\text{cons}(l_1, l_2)), \text{ given that } As(l_1) \text{ and } As(l_2) \text{ hold.}$$

Since \mathcal{B} is defined as the smallest collection containing atoms from A and closed w.r.t. cons, these two clauses establish validity of the method.

Regrettably, assertion (3.1) cannot be proved straightforwardly using structural induction:

Assume, $tm \notin l_i \rightarrow Q(l_i, tm) = P(l_i)$, $i = 1, 2$,

and try to prove $tm \notin \text{cons}(l_1, l_2) \rightarrow Q(\text{cons}(l_1, l_2), tm) = P(\text{cons}(l_1, l_2))$.

Assume $tm \notin \text{cons}(l_1, l_2)$. Then $tm \notin l_i$, $i = 1, 2$. Therefore

$Q(l_i, tm) = P(l_i)$, $i = 1, 2$.

$Q(\text{cons}(l_1, l_2), tm) = Q(l_1, \text{cons}(l_2, tm))$ follows from Q 's definition.

In order to say something about $Q(l_1, \text{cons}(l_2, tm))$, one should resort to the assumptions. But these say only something about $Q(l_1, tm)$. Not about $Q(l_1, r)$ for general r . I.e., we stumble across the obstacle that (.) only comments upon the initial and final states of the computation, and doesn't pronounce itself about the intermediate stages.

Hence we look for an assertion which also captures the intermediate stages of the computation of $Q(l, tm)$. This assertion is:

$$\forall l, r \in \mathcal{B}. (l \notin tm \rightarrow Q(l, r) = Q(r, P(l))), \dots \quad (3.2)$$

as reflected in the drawings!

Define $tm \notin l$ by the boolean procedure

$$tm \notin l \leftarrow \underline{\text{if}} \text{ at}(l) \underline{\text{then}} \neg(l = tm) \underline{\text{else}} tm \notin \text{car}(l) \wedge tm \notin \text{cdr}(l) \underline{\text{fi}},$$

which determines whether or not tm occurs amongst the atoms of l .

PROOF of (3.2):

By structural induction upon l .

If $tm \notin l$ does not hold the assertion holds trivially, hence assume $tm \in l$:

There are 2 cases:

(i) $\text{at}(l)$: Then $Q(l, r) =$ (since $\neg(l = tm)$ follows from $tm \notin l$)

$$Q(r, A(l)) = Q(r, P(l)).$$

(ii) $\neg \text{at}(l)$: Then there exist l_1, l_2 s.t. $l = \text{cons}(l_1, l_2)$. Assume (3.2) by hypothesis for l_1, l_2 :

$$Q(l, r) = Q(\text{cons}(l_1, l_2), r) =$$

$$Q(\text{car}(\text{cons}(l_1, l_2)), \text{cons}(\text{cdr}(\text{cons}(l_1, l_2)), r)) =$$

$$Q(l_1, \text{cons}(l_2, r)) = \text{(since } tm \notin l \text{ implies for } \neg \text{at}(l) \text{ that}$$

$l_1 \notin tm$ we derive from the induction hypothesis:)

$$Q(\text{cons}(l_2, r), P(l_1)) = Q(l_2, \text{cons}(r, P(l_1))) =$$

$$\text{(hyp.) } Q(\text{cons}(r, P(l_1)), P(l_2)) = Q(r, \text{cons}(P(l_1), P(l_2))) =$$

$$Q(r, P(l)).$$

□

EXERCISE:

Define L , the collection of linear lists w.r.t. a base set A of atoms as the smallest collection L s.t.: (i) $A \subseteq L$, (ii) $a \in A, l \in L$ then $\text{cons}(a, l) \in L$, where cons is a total function of type $A \times L \xrightarrow[1-1]{\text{onto}} (L - A)$, and let car, cdr be defined as the usual one-sided inverses to cons .

Declare procedure $\text{conc}(l_1, l_2)$, concatenating l_1 to l_2 by

$$\text{conc}(l_1, l_2) \leftarrow \underline{\text{if}} \text{ at}(l_1) \underline{\text{then}} \text{cons}(l_1, l_2) \\ \underline{\text{else}} \text{cons}(\text{car}(l_1), \text{conc}(\text{cdr}(l_1), l_2)) \underline{\text{fi}}.$$

Then conc is associative, i.e.,

$$\text{conc}(\text{conc}(l_1, l_2), l_3) = \text{conc}(l_1, \text{conc}(l_2, l_3)).$$

Prove this property by structural induction upon l_1 .

□

Symbolic execution

In the correctness proof given above we "executed" Q (in a certain fashion) in the sense that we consulted their procedure bodies in order to determine the next computation step, e.g.,

$$Q(\text{cons}(l_1, l_2), r) = Q(l_1, \text{cons}(l_2, r)).$$

These "executions" are different from the ones made on a machine in that a machine operates on actual values. I.e., in that case l_1, l_2, r denote actual values, while in the former case - that of the proof - we used the observation that for each l of the form $\text{cons}(l_1, l_2)$, $Q(\text{cons}(l_1, l_2), r) = Q(l_1, \text{cons}(l_2, r))$, i.e., we executed $Q(\text{cons}(l_1, l_2), r)$ symbolically in that we described a computation step which will be made on a machine for every l of the form $\text{cons}(l_1, l_2)$ and which therefore holds for all nonatomic binary trees l . Moreover, the case that l denotes an atom is treated as a special case.

As a result, our proofs apply to every computation sequence of $Q(l, r)$ on a machine.

This notion of executing a computation step symbolically belongs to the foundation of program proving and is called symbolic execution. Although he didn't coin the terminology, this technique is already used by McCarthy in his "Towards a mathematical theory of computation" (1961), the article founding program correctness as a subject worthy of academic pursuit.

Critique of the above proof

Let us first recapitulate the technique introduced above by investigating the answer to the exercise of proving associativity of conc , in order to underline that the contents of the present section apply in general, and do not apply only to the proof previously given.

PROOF: $\text{conc}(\text{conc}(l_1, l_2), l_3) = \text{conc}(l_1, \text{conc}(l_2, l_3)), l_i \in L, i = 1, 2, 3.$

By structural induction upon l_1 :

$$(i) \text{ at}(l_1): \text{conc}(l, \text{conc}(l_2, l_3)) = \text{cons}(l, \text{conc}(l_2, l_3)) = \\ \text{conc}(\text{cons}(l, l_2), l_3) = \text{conc}(\text{conc}(l, l_2), l_3).$$

(ii) $l_1 = \text{cons}(a, l)$. Assume the result by hypothesis for l, l_2, l_3 .

$$\begin{aligned} \text{conc}(l, \text{conc}(l_2, l_3)) &= \text{conc}(\text{cons}(a, l), \text{conc}(l_2, l_3)) = \\ \text{cons}(a, \text{conc}(l, \text{conc}(l_2, l_3))) &= (\text{hyp.}) \\ \text{cons}(a, \text{conc}(\text{conc}(l, l_2), l_3)) &= \\ \text{conc}(\text{cons}(a, \text{conc}(l, l_2)), l_3) &= \\ \text{conc}(\text{conc}(\text{cons}(a, l), l_2), l_3) &= \\ \text{conc}(\text{conc}(l_1, l_2), l_3). \end{aligned}$$

□

Concatenate for the moment on a collection of finite and infinite binary trees \mathcal{B}' , i.e., we do not restrict ourselves to the smallest collection of binary trees \mathcal{B} closed w.r.t. the operation cons , but consider the largest one, \mathcal{B}' . Clearly $\mathcal{B} \subseteq \mathcal{B}'$.

The principle of structural induction as enounced above is only valid when one restricts one's inputs to trees l contained in \mathcal{B} . However, if $\text{tm} \notin l$ then $Q(l, r) = Q(r, P(l))$ still holds for infinite trees l and r , in that one can extend one's notion of equality by considering two expressions also equal in value if both do not terminate (are not defined); for $Q(l, r)$ doesn't terminate (is not defined) in that case, and neither is $P(l)$, and therefore $Q(r, P(l))$, provided one adheres to the convention that cons doesn't terminate (is not defined), in case one of its operands does not terminate (is not defined).

Since the principle of structural induction (as formulated above) doesn't extend to the whole of \mathcal{B}' , while

$$\text{tm} \notin l \rightarrow Q(l, r) = Q(r, P(l)) \text{ does hold for } l, r \in \mathcal{B}' \dots \quad (3.3)$$

we shall give another proof of this assertion using a different induction principle, which does extend to \mathcal{B}' .

Similar observations recently emerged in (Lehmann & Smyth 1977) in their remarks about structural induction.

The same reasoning holds *mutatis mutandis* for our proof of associativity of conc .

4. An informal encounter with mathematical semantics

In this section we make an important transition: from operational semantics to mathematical semantics, or, from body-replacement semantics to a different kind of semantics involving the notion of approximation.

First we introduce a formal value, denoted by \perp and pronounced as (Scott's) "bottom", or "undefined", to express nontermination of the computation of a value of an expression, or its undefinedness.

Next we define, for any recursive declaration

$$f(x_1, \dots, x_n) \rightarrow \tau[f](x_1, \dots, x_n),$$

$f^{(0)}$ as $\lambda x_1, \dots, x_n. \perp$, i.e., the function which always delivers the result \perp and $f^{(n+1)}$ as $\lambda x_1, \dots, x_n. \tau[f^{(n)}](x_1, \dots, x_n)$.

Informally, $f^{(n)}(x_1, \dots, x_n) \neq \perp$ just in case $f(x_1, \dots, x_n)$ terminates with recursion-depth $< n$, and that $f^{(n)}(x_1, \dots, x_n) = \perp$ iff computation of $f(x_1, \dots, x_n)$ requires recursion-depth $\geq n$ (assuming there occur no undefined operations in τ).

The crucial distinction between f and $f^{(n)}$ is that f is defined using body-replacement semantics, while $f^{(n)}$ is just a function defined by a very long expression (whose length increases with increasing n) of definite length for given n .

In formal approaches to the subject of mathematical semantics quite a fuss is made about this distinction, and in particular the programming language expression $\tau[f]$ is transformed into an expression belonging to a formal, rigorously defined, language. However, since for the simple programming language constructs considered in this chapter the essential difference only resides in recursion, we shall not do so; e.g., the conditional can clearly be used as a case distinction in the definition of a mathematical function.

Thirdly, for a given set D , we introduce a notion of partial order \sqsubseteq by extending D with \perp to $D \cup \{\perp\}$ and defining \sqsubseteq by: $x \sqsubseteq y$ just in case either $x = \perp$ or $x = y$.

It will be clear that the I/O behaviour of a recursive procedure defines a partial function. Now \sqsubseteq can be related as follows to such partial functions over D :

Given any partial function f over D , consider the corresponding total function f' over $D \cup \{\perp\}$ which is defined by: if $f(x)$ is defined then $f(x) = f'(x) \in D$, and $f'(x) = \perp$, otherwise. This correspondence is an isomorphism, and henceforward we shall identify f and f' .

For total functions over $D \cup \{\perp\}$ one defines $f_1 \sqsubseteq f_2$ iff $\forall x \in D \cup \{\perp\}$, $f_1(x) \sqsubseteq f_2(x)$. Viewed as a partial order between functions - that this is a partial order indeed should be proved - \sqsubseteq is called "approximation relation" in the lingo of semantics.

An important point is that we shall not consider all total functions over $D \cup \{\perp\}$ (or from any $D_1 \cup \{\perp\}$ to $D_2 \cup \{\perp\}$ for that matter):

In our model \perp denotes the value of an infinitely proceeding computation, or an undefined value such as the result of dividing by 0. Since we want our functions to model the I/O behaviour of procedures, when comparing $f(\perp)$ with $f(x)$ for $x \in D$, we either expect $f(\perp) = \perp$, or in case $f(\perp) \in D$ that $f(\perp) = f(x)$ since in that case f should denote a constant function which does not evaluate its argument. Consequently f should be monotone in its arguments. Later on we shall restrict ourselves to functions f s.t. $f(\perp) = \perp$, called strict, because we intend to model call-by-value.

This notion of approximation between functions ties neatly up with the notion of $f^{(n)}$ introduced above, for $f^{(n)} \sqsubseteq f^{(n+1)}$:

informally,

in case $f(x_1, \dots, x_n)$ terminates with value v with recursion-depth $< n$ both $f^{(n)}(x_1, \dots, x_n) = v$ and $f^{(n+1)}(x_1, \dots, x_n) = v$,

in case computation of $f(x_1, \dots, x_n)$ requires recursion-depth n , $f^{(n)}(x_1, \dots, x_n) = \perp$, and

in case computation of $f(x_1, \dots, x_n)$ requires recursion-depth $> n$, both $f^{(n)}(x_1, \dots, x_n) = \perp$ and $f^{(n+1)}(x_1, \dots, x_n) = \perp$,

and hence $f^{(n)}(x_1, \dots, x_n) \sqsubseteq f^{(n+1)}(x_1, \dots, x_n)$ in all three cases.

Our reservation "on informal grounds" is not serious, since the result can be rigorously proved within a fullfledged mathematical framework - and this we do not intend to pursue for the moment.

Observe that, since \sqsubseteq denotes a partial order, the least upper bound $\lim_{i \rightarrow \infty} x_i$ of a monotonically nondecreasing sequence $\{x_i\}$ (i.e., a sequence of values x_i s.t. $x_i \sqsubseteq x_{i+1}$, $i \in \mathbb{N}$) can be introduced.

One distinguishes three cases:

either $x_i = x_{i+1} = \perp$ for all i , and hence $\lim x_i = \perp$,

or $x_i = x_{i+1} \neq \perp$ for all i , and hence $\lim x_i = x_0$,

or $\exists n$, s.t. $x_i = \perp$ for $i < n$, and $x_n = x_{n+1} = \dots = x_{n+n} = \dots$, $n \in \mathbb{N}$, and $\lim x_i = x_n$.

Trivial as this notion may be at this level, it is useful in that it can be extended to a monotonically nondecreasing sequence of functions $\{f_n\}$:

$\lim_{i \rightarrow \infty} f_i$ is defined by $(\lim f_i)(x_1, \dots, x_n) = \lim f_i(x_1, \dots, x_n)$. Note that this definition implies an existence proof of $\lim_{i \rightarrow \infty} f_i$.

Therefore, if we restrict our attention to the I/O behaviour of a recursive procedure f (thus abstracting from any properties involving the intermediate values of a computation of its value) we obtain $f = \lim f^{(n)}$, again on informal grounds:

either $f(x_1, \dots, x_n)$ is a well-defined value v , and then there exists an index N s.t. $f^{(i)}(x_1, \dots, x_n) = \perp$, $i < N$, and $f^{(i)}(x_1, \dots, x_n) = v$, $i \geq N$, or $f(x_1, \dots, x_n)$ is undefined, and then $f^{(i)}(x_1, \dots, x_n) = \perp$, $i \in \mathbb{N}$.

These results are combined as follows with the parameter mechanism we use: We shall restrict ourselves below to call-by-value, since in programming language design call-by-name is now of historical interest only. (For call-by-name, though, a similar mathematical theory can be developed, though.)

Let D_1, \dots, D_n denote sets of well-defined values, and $D_1 \cup \{\perp_1\}, \dots, D_n \cup \{\perp_n\}$ their extensions with partial orders $\subseteq_1, \dots, \subseteq_n$.

Then one defines $\subseteq_{1, \dots, n}$ by extending $D_1 \times \dots \times D_n$, the cartesian product of D_1, \dots, D_n to $(D_1 \cup \{\perp_1\}) \times \dots \times (D_n \cup \{\perp_n\})$ and using the componentwise order over the n -tuples thus obtained:

$\langle x_1, \dots, x_n \rangle \subseteq_{1, \dots, n} \langle y_1, \dots, y_n \rangle$ iff $x_i \subseteq_i y_i$, $i = 1, \dots, n$.

Hence functions (used in) describing the I/O behaviour of recursive procedures with parameters called by value satisfy:

if $\exists j. x_j = \perp_j$, then $f(x_1, \dots, x_j, \dots, x_n) = \perp$. Such functions are called strict, and will be the only kind considered here; e.g., $\text{cons}(\perp, 1) = \text{cons}(1, \perp) = \perp$.

(This convention is not always followed in literature.)

Finally, when making the transition from operational to mathematical reasoning, we have to pinpoint the mathematical analogue of a declaration

$$P \leftarrow \tau[P]$$

of a recursive procedure P .

With our newly introduced notions, $\tau[f]$ can be conceived of as a transformation of functions f (, i.e., as $\lambda f. \tau[f]$, properly speaking).

As main result one has the famous property that the I/O behaviour $\lim P^{(n)}$ of P is the least fixed point of the equation $f = \tau[f]$. This result depends crucially on the property of τ that for any monotonically nondecreasing sequence of functions $\{f_i\}$,

$$\tau[\lim f_i] = \lim \tau[f_i],$$

i.e. that τ is continuous in f ; transformations τ which are derived from

programming language expressions are always continuous.

In the proof below we shall also mention the fixed point property, i.e. that $\lim P^{(n)} = \tau[\lim P^{(n)}]$.

Finally we tie this machinery up with a proof of (3.2).

From the remarks above it follows that, in order to prove 3.2, we should prove:

$$\forall l, r \in B'. \text{tm} \notin l \rightarrow Q(l, r) \sqsubseteq Q(r, P(l)), \dots \quad (4.1)$$

and

$$\forall l, r \in B'. \text{tm} \notin l \rightarrow Q(r, P(l)) \sqsubseteq Q(l, r), \dots \quad (4.2)$$

where Q and P denote now mathematical objects, i.e.,

least fixed points (τ , in this case), describing I/O behaviours.

It follows from the remarks above that in order to prove (4.1) we should prove:

$$\forall l, r \in B'. \text{tm} \notin l \rightarrow (\lim Q^{(n)})(l, r) \sqsubseteq (\lim Q^{(n)})(r, P(l)),$$

and this follows from proving

$$\text{for all } n: \forall l, r \in B'. \text{tm} \notin l \rightarrow Q^{(n)}(l, r) \sqsubseteq Q^{(n)}(r, P(l)):$$

PROOF: Assume $\text{tm} \notin l$. By course-of-values induction:

$n = 0$: $Q^{(0)}(l, r) = \perp$ and hence the inclusion $\perp \sqsubseteq \dots$ is trivially satisfied.

$n = 1$: $Q^{(1)}(l, r) = Q^{(0)}(\dots)$, since $\text{tm} \notin l, \dots = \perp$.

$n > 1$: Assume the result by hypothesis for $k < n$, and prove it for n :

at(1): $Q^{(n)}(l, r) = Q^{(n-1)}(r, A(l)) = (\text{fixed point property on } P)$

$$Q^{(n-1)}(r, P(l)) \sqsubseteq Q^{(n)}(r, P(l)), \text{ since } \text{tm} \notin l.$$

\neg at(1): say, $l = \text{cons}(l_1, l_2)$:

$$Q^{(n)}(\text{cons}(l_1, l_2), r) = Q^{(n-1)}(l_1, \text{cons}(l_2, r)) \sqsubseteq (\text{induction hypothesis})$$

$$Q^{(n-1)}(\text{cons}(l_2, r), P(l_1)) = Q^{(n-2)}(l_2, \text{cons}(r, P(l_1))) \sqsubseteq (\text{induction hypothesis})$$

$$Q^{(n-2)}(\text{cons}(r, P(l_1)), P(l_2)) = \text{if } n = 2 \text{ then } \perp \text{ and the result follows,}$$

$$\text{if } n > 2 \text{ then } Q^{(n-3)}(r, \text{cons}(P(l_1), P(l_2))) = (\text{fixed point property on } P)$$

$$Q^{(n-3)}(r, P(l)) \sqsubseteq Q^{(n)}(r, P(l)).$$

□

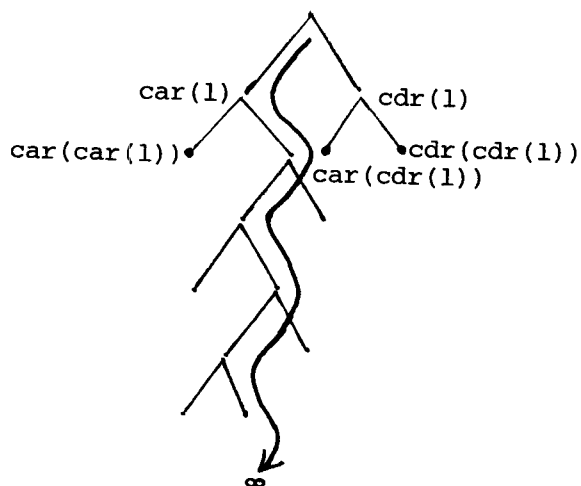
Next we prove (4.2).

{It should be noted that (4.2) would not have held, if we hadn't made the assumption that $\text{cons}(l, \perp) = \text{cons}(\perp, l) = \perp$.

This can be understood as follows:

Assume $\text{cons}(l, \perp)$ and $\text{cons}(\perp, l)$ not necessarily \perp , i.e., cons does not necessarily evaluate its arguments.

Let l denote an infinite tree:



$$P(l) = \lim P^{(n)}(l).$$

$$P^{(0)}(l) = \perp, P^{(1)}(l) = \text{cons}(P^{(0)}(\text{car}(l)), P^{(0)}(\text{cdr}(l))) = \text{cons}(\perp, \perp).$$

$$P^{(2)}(l) = \text{cons}(\text{cons}(\text{car}(\text{car}(l)), \perp), \text{cdr}(l)), \text{ which may not be necessarily } \perp.$$

Hence $Q(\text{tm}, P(l))$ not necessarily \perp .

But $Q(l, \text{tm}) = \perp$, as can be deduced from $Q = \lim Q^{(n)}$ and the fact that l contains an infinite branch.

The property of $\text{cons}(\perp, l)$ or $\text{cons}(l, \perp)$ not being necessarily \perp is made use of in special applications involving "lazy evaluations" (Morris & Henderson, Friedman & Wise) originating in work of Jean Vuillemin; lazy evaluation postpones evaluation of the arguments of cons dependent upon certain conditions.}

Let us now assume that $\text{cons}(l, \perp) = \text{cons}(\perp, l) = \perp$.

In order to prove (4.2) we shall prove:

$$\text{for all } n, \forall l, r \in \mathcal{B}. \text{tm} \notin l \rightarrow Q(r, P^{(n)}(l)) \subseteq Q(l, r).$$

By our statement that any programming language expression induces a continuous transformation (4.2) follows.

PROOF: By n-step-n+1 induction.

$n = 0$: $Q(r, P^{(0)}(l)) = Q(r, l) = l \subseteq Q(l, r)$.

$n \geq 1$: at(l): $Q(r, P^{(n)}(l)) = Q(r, A(l)) = Q(l, r)$ since $tm \notin l$.

\neg at(l): Assume the result by hypothesis for $k < n$, and prove it for n :

$l = \text{cons}(l_1, l_2)$:

$Q(r, P^{(n)}(\text{cons}(l_1, l_2))) = Q(r, \text{cons}(P^{(n-1)}(l_1), P^{(n-1)}(l_2))) =$ (fixed point prop. on Q)

$Q(\text{cons}(r, P^{(n-1)}(l_1)), P^{(n-1)}(l_2)) \subseteq$ (induction hypothesis)

$Q(l_2, \text{cons}(r, P^{(n-1)}(l_1))) =$ (fixed point property on Q)

$Q(\text{cons}(l_2, r), P^{(n-1)}(l_1)) \subseteq$ (induction hypothesis)

$Q(l_1, \text{cons}(l_2, r)) =$ (fixed point property on Q)

$Q(\text{cons}(l_1, l_2), r)$.

□

Our proof of (4.1) can also be given using n-step-n+1 induction instead of course-of-values induction, by proving

$$[\forall l, r \in \mathcal{B}^1. tm \notin l \rightarrow Q^{(n)}(l, r) \subseteq Q^{(n)}(r, P(l))] \& [Q^{(n)} \subseteq Q^{(n+1)}].$$

This can be understood as follows:

The first application of the induction hypothesis in the proof of (4.1) is followed by $\dots \subseteq Q^{(n)}(\text{cons}(l_2, r), P(l_1)) = Q^{(n-1)}(l_2, \text{cons}(r, P(l_1)))$, and now we can appeal again to the induction hypothesis for $n-1$, instead of $n-2$.

Next we do the same for our second application of the induction hypothesis.

This observation derives its importance from the fact that a proof by n-step-n+1 induction can be straightforwardly transformed into one using Scott-induction in a more formal framework. Scott-induction is a famous induction rule which unifies the other induction rules in appropriate formal frameworks; this rule will be discussed later on.

5. On correctness of backtracking (, and iterative tree-copying)

In this section we discuss correctness of that version of the backtracking technique, as described by, e.g., Floyd, in which all solutions of a problem which can be represented as a finite tree search are requested. (At the end of the section we discuss briefly that version which stops after finding any solution of the problem, instead of all solutions.)

In general, three properties are crucial if one is to speak of 'backtracking':

- (1) The problem concerns finding one or more solutions of a problem involving search in a space which can be represented as a tree.
- (2) This tree has a notion of 'partial solutions' applicable to its nodes such that
 - either such a solution is refuted because it does not approximate any 'completed solution' (- approximate in the sense of 'is an initial part of a' -)
 - or it can be improved to a more complete (partial) solution by going one step deeper into the tree-representation, until a completed (partial) solution at the tips of the tree is reached, which is either accepted and processed, or refuted because it doesn't satisfy the constraints of the particular problem.
- (3) This tree-space is to be traversed iteratively.

A direct consequence of these properties is that a correctness proof of a backtracking algorithm involving the search for all solutions to a given problem can be split into two separate parts:

- (A) A 'high-level' proof of correctness of the recursively defined search-space in that all solutions to that problem can be extracted from that tree structured search-space.
- (B) A 'lower-level' proof that the iterative algorithm traverses all of the search-space indeed.

The particular algorithm to be shortly discussed incorporates the pointer-reversal technique described previously, and hence part (B) of the correctness proof resembles the one discussed previously.

However, the general point to be stressed is that any strategy for iterative tree-traversal could be used; the pointer-reversal strategy just provides an example of such a strategy.

As a surprising bonus we shall see that our iterative solution describes in its schematic form also a tree-copying algorithm.

Assume for simplicity that the problem can be represented as searching a space which is structured as a binary tree; in case of the 8 queen's problem an octary tree is involved, and in the general case, to be discussed lateron, one has to resort to lists, which are again binary. Then a recursive search of the binary-structured-tree-space can be described in principle by a recursive procedure of the following form:

$$P(l) \leftarrow \text{if } \text{partial-solution-can-be-extended}(l) \text{ then } P(S_1(l)) \cup P(S_2(l)) \\ \text{else if } \text{partial-solution-is-dead-end}(l) \text{ then } \emptyset \\ \text{else } \text{extract-solution}(l) \text{ fi,} \quad \dots \quad (5.1)$$

where

- (i) partial-solution-can-be-extended is a total predicate, and if partial-solution-can-be-extended(l) is true, then $S_1(l)$ and $S_2(l)$ are defined,
- (ii) "U" denotes set-theoretical union, \emptyset the empty set, and $P(l)$, if defined has a set as value,
- (iii) if partial-solution-can-be-extended(l) is false, then partial-solution-is-dead-end(l) is defined,
- (iv) extract-solution(l) denotes the one-element set consisting of a solution; its existence is guaranteed by the fact that both partial-solution-can-be-extended(l) and partial-solution-is-dead-end(l) are false.

The above conditions guarantee that no undefined elementary operations (such as, e.g., division by zero in another context) cause undefinedness of computation.

For our purpose one may just as well consider (5.1) in this section as a program scheme of the form

$$P(l) \leftarrow \text{if } \text{ext}(l) \text{ then } P(S_1(l)) \cup P(S_2(l)) \text{ else } \text{Sol}(l) \text{ fi} \quad \dots \quad (5.2)$$

EXAMPLE: 4 queen's problem, recursive solution.

The 4 queen's problem requires a quaternary tree representation. Therefore its 'high-level' recursive characterization has the following schematic form:

$$P(l) \leftarrow \text{if } \text{ext}(l) \text{ then } P(S_1(l)) \cup P(S_2(l)) \cup P(S_3(l)) \cup P(S_4(l)) \\ \text{else } \text{Sol}(l) \text{ fi.}$$

The intended interpretation of $\text{ext}, S_1, S_2, S_3, S_4$, and Sol is given by:

$$P(\text{col}, s) \leftarrow \underline{\text{if}} \text{col} < 4 \wedge \text{free}(\text{col}, s) \underline{\text{then}} \bigcup_{i=1}^4 P(\text{col}+1, i.s) \\ \underline{\text{else}} \underline{\text{if}} \text{col}=4 \wedge \text{free}(\text{col}, 4) \underline{\text{then}} \{s\} \underline{\text{else}} \emptyset \underline{\text{fi}},$$

where

- (i) $\text{col} \in \{0, 1, 2, 3, 4\}$, s denotes a linear list with 1, 2, 3 or 4 as atoms, and the value of $P(\text{col}, s)$, if defined, is a set consisting of linear lists,
- (ii) free denotes a total predicate checking the particular constraints as dictated by the 4 queen's problem, of which the specific property of approximate solutions is discussed lateron,
- (iii) "U" denotes set-theoretical union, "." denotes the concatenation operation of an atom to a linear list, and $\{s\}$ the unit set with element s .

The call $P(0, \Lambda)$ with Λ denoting the empty linear list, is intended to have as value a set consisting of all solutions to the 4 queen's problem, i.e., linear lists satisfying $\text{free}(4, s)$. Its execution tree is represented in the figure on the next page.

Observe that the arrow-reversal game applies again to the tree-structural search-space of the 4 queen's problem, as indicated in the lower half of the picture.

However, one faces the following complication:

The game is not played upon a tree in situ in memory (as in the original version of the game), but upon a tree which has to be developed while the game is going on.

The solution to this complication is to realize that, in fact, there are now more phases to this game:

either

- (i) a partial solution is further developed in depth-first manner,
- or (ii) an atom in the search tree has been found,
- or the termination marked has been hit,
- or the lefthand argument does not denote a partial solution or an atom, but a stack, from which the components have to be isolated.

That is: reversing a pointer is a rule which both applies to the phase in which a partial solution is being developed - in the example by S_1, S_2, S_3, S_4 - , and to the phase in which a previously constructed tree (stack) is taken apart.

This idea is incorporated in the following iterative procedure, where for the sake of simplicity we returned to the binary tree representation:

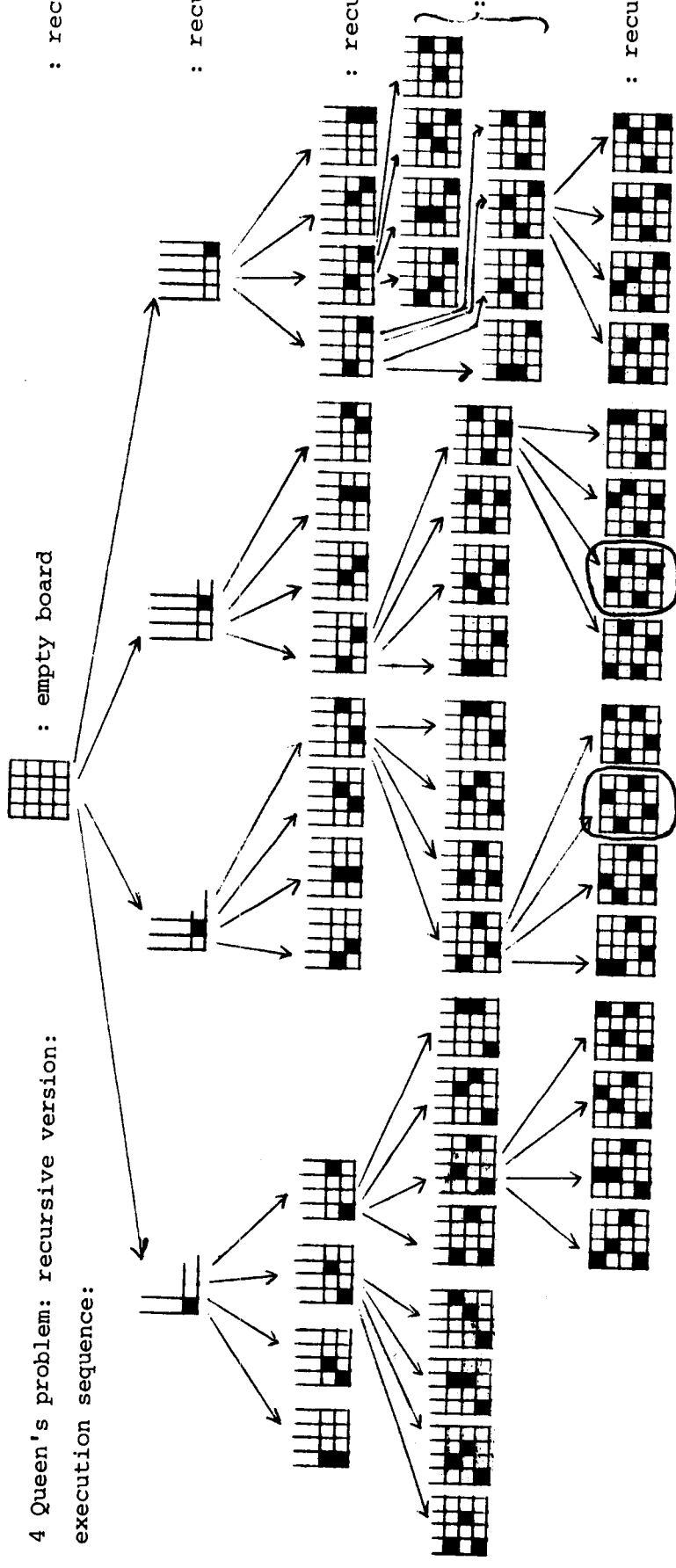
: recursiondepth 0

: recursiondepth 1

: recursiondepth 2

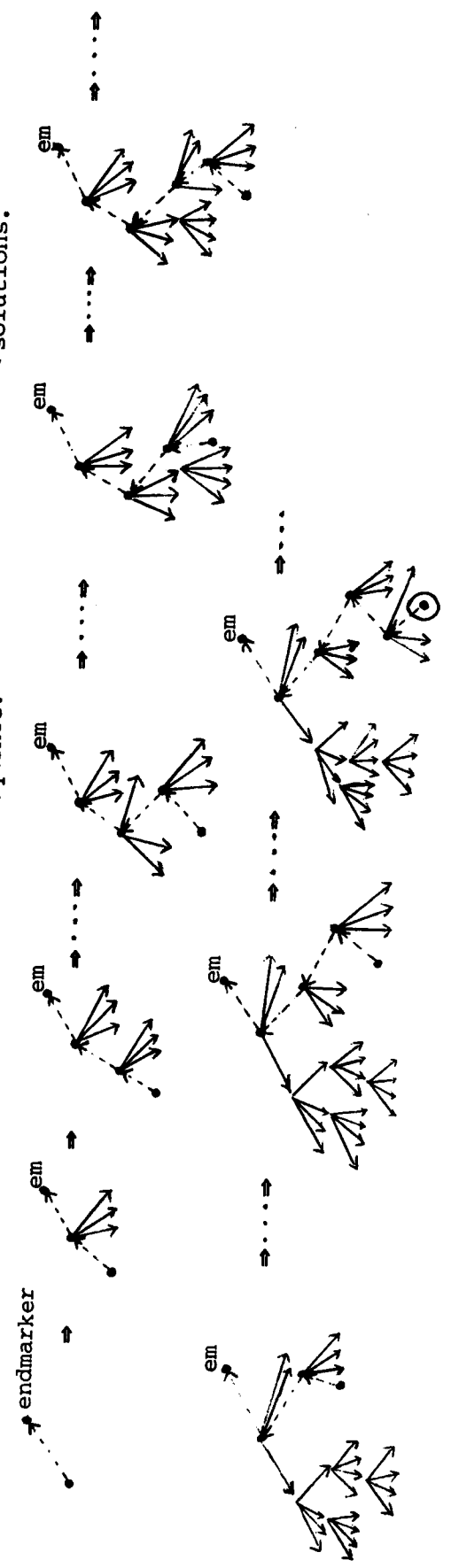
: recursion-
depth 3

: recursiondepth 4



4 Queen's problem: recursive version:
execution sequence:

4 Queen's problem: iterative version: execution sequence:
solutions!



$$\begin{aligned}
 B(l,r) \leftarrow & \text{if } \underline{\text{is-probl-repr}}(l) \text{ then} \\
 & \text{if } \underline{\text{ext}}(l) \text{ then } \underline{B}(S_1(l), \text{cons}(S_2(l), r)) \text{ else } \underline{B}(r, \text{Sol}(l)) \text{ fi} \\
 & \text{else} \\
 & \text{if } l = \underline{\text{endmarker}} \text{ then } r \text{ else } \underline{B}(\text{car}(l), \text{cons}^\nabla(\text{cdr}(l), r)) \text{ fi}, \\
 \text{cons}^\nabla(l,r) \leftarrow & \text{if } \underline{\text{is-set}}(l) \wedge \underline{\text{is-set}}(r) \text{ then } l \cup r \text{ else } \underline{\text{cons}}(l,r) \text{ fi},
 \end{aligned}$$

where l and r denote binary trees over atoms which are either sets - with characteristic predicate $\text{is-set}(l)$ - or, disjunct with these, partial solutions (problem representations) - with characteristic predicate $\text{is-probl-repr}(l)$ - or, disjunct with these and the sets, the endmarker.

Moreover, the predicates $\text{is-probl-repr}(l)$, $\text{ext}(l)$, $l = \text{endmarker}$, $\text{is-set}(l)$, and the constants S_1 , S_2 , cons , Sol , car , cdr , cons^∇ are defined in such a way that none of these predicates or constants ever gets undefined for its arguments: e.g., $\text{is-probl-repr}(l)$ is total, $\text{is-probl-repr}(l)$ implies that $\text{ext}(l)$ is defined, $\text{is-probl-repr}(l) \wedge \text{ext}(l)$ implies that $S_1(l), S_2(l)$ are defined, cons is total, $\text{is-probl-repr}(l) \wedge \neg \text{ext}(l)$ implies that $\text{Sol}(l)$ is defined etc..

The appearance of a predicate s.a. $\text{is-probl-repr}(l)$ should cause no surprise: as remarked previously, in this arrow-reversal game either a partial solution should be further developed - this situation is characterized by $\text{is-probl-repr}(l)$ - , and, if so, the partial solution is processed further, i.e., either extended - characterized by $\text{ext}(l)$ - or extracted - characterized by $\neg \text{ext}(l)$; or l denotes in fact a stack of temporarily stored partial solutions (or sets) which should be extracted - characterized by $\neg (l = \text{endmarker}) \wedge \neg \text{is-probl-repr}(l)$; or $\wedge \neg \text{is-probl-repr}(l) \wedge l = \text{endmarker}$.

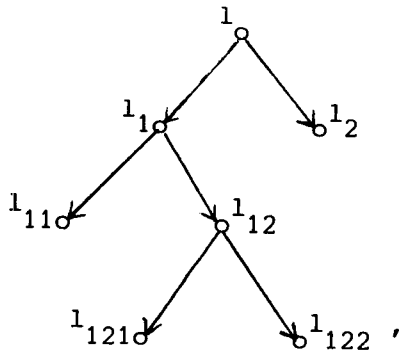
Semantically, procedure B , although compact, may be difficult to understand; yet it embodies the very notion of backtracking for all solutions, for the following assertion holds:

$$\begin{aligned}
 \forall l_1, l_2. \neg \text{is-set}(\text{cons}(l_1, l_2)) \rightarrow \\
 (\text{is-probl-repr}(l) \wedge \neg \text{is-set}(r) \rightarrow B(l,r) = B(r, P(l))), \dots
 \end{aligned} \tag{5.3}$$

which implies in particular:

$$\begin{aligned}
 \forall l_1, l_2. \neg \text{is-set}(\text{cons}(l_1, l_2)) \rightarrow \\
 (\text{is-probl-repr}(l) \rightarrow B(l, \text{endmarker}) = P(l)).
 \end{aligned}$$

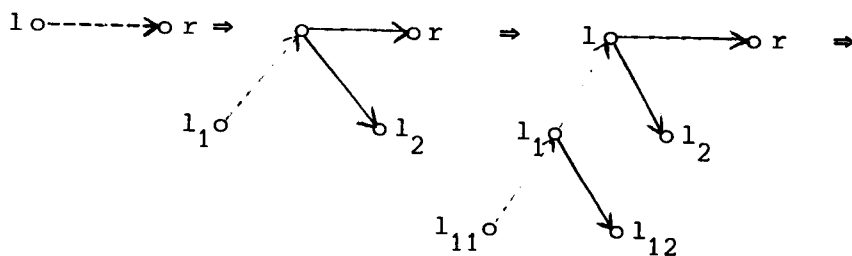
Part of a symbolic computation of $B(l,r)$ is specified below, where the search space accessible from l has the following structure:



with $l_{i_1 \dots i_k} = S_{i_k} (S_{i_{k-1}} \dots S_{i_1} (1) \dots)$.

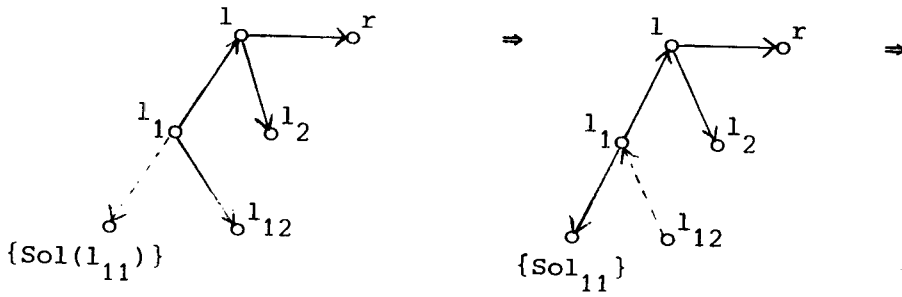
The description of each step in this sequence is accompanied by a pictorial representation of the state at that moment. Since B has two arguments, the second argument is pointed to by a dotted arrow. The search tree is built up and accessed in depth-first left-preferent order, and completed potential solutions are stored in that order, too, at the place of the corresponding problem representation. A subtree with only completed potential solutions at its tips is replaced by the set consisting of those solutions, since the tree-structure of that subtree has become redundant:

A: First the structure accessible from 1 is developed in a left-preferent manner, stacking the RHS problem descriptions (comparable with return addresses) in the right argument, until a completed problem description is met:

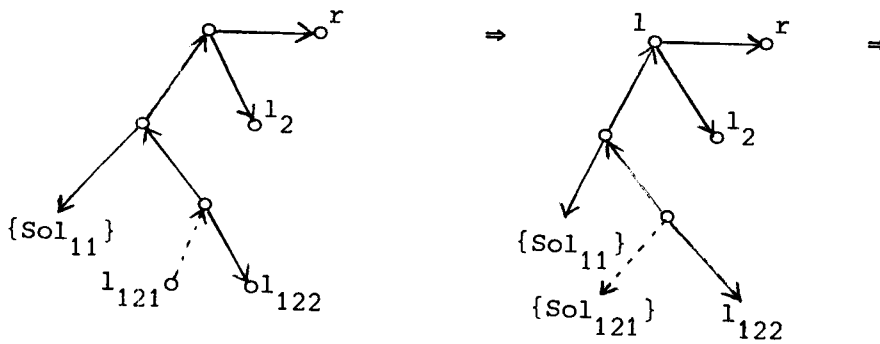


$$Q(1, r) = Q(1_1, \text{cons}(1_2, r)) = Q(1_{11}, \text{cons}(1_{12}, \text{cons}(1_2, r))) = \dots$$

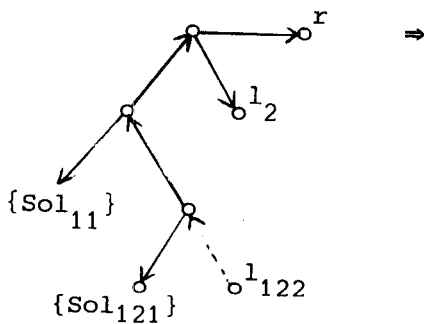
B: Then the first completed potential solution is stored as RHS argument, and the stack (the original RHS argument) becomes the new LHS argument, resulting in a last-in-first-out storage discipline:



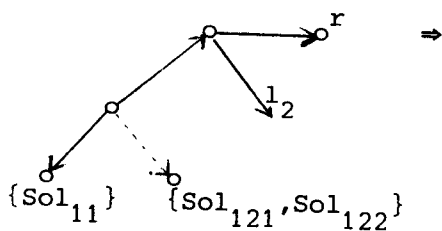
$$B(\text{cons}(l_{12}, \text{cons}(l_2, r)), \{\text{Sol}_{111}\}) = B(l_{12}, \text{cons}(\text{cons}(l_2, r), \{\text{Sol}_{11}\})) = \text{Sol}(l_{11}) = P(l_{11})$$



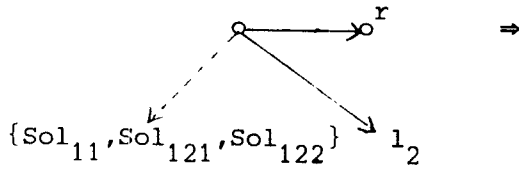
$$B(l_{121}, \text{cons}(l_{122}, \text{cons}(\text{cons}(l_2, r), \{\text{Sol}_{11}\}))) = B(\text{cons}(l_{122}, \text{cons}(\text{cons}(l_2, r), \{\text{Sol}_{11}\})), \{\text{Sol}_{121}\}) = P(l_{121})$$



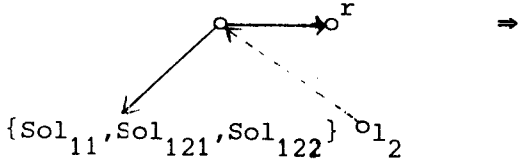
$$B(\text{cons}(\text{cons}(\text{cons}(l_2, r), \{\text{Sol}_{11}\}), \{\text{Sol}_{121}\}), \{\text{Sol}_{122}\}) = P(l_{122})$$



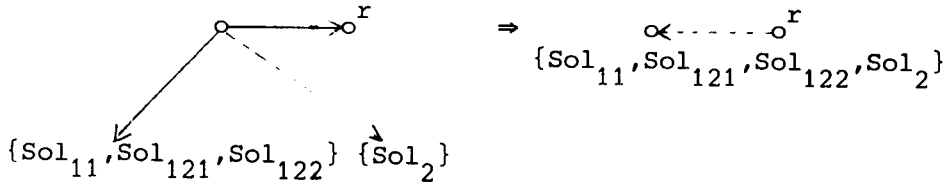
$$B(\text{cons}(\text{cons}(l_2, r), \{\text{Sol}_{11}\}), \{\text{Sol}_{121}, \text{Sol}_{122}\}) = P(l_{12})$$



$$B(\text{cons}(l_2, r), \underbrace{\{\text{Sol}_{11}, \text{Sol}_{121}, \text{Sol}_{122}\}}_{P(l_1)})$$



$$B(l_2, \text{cons}(r, \{\text{Sol}_{11}, \text{Sol}_{121}, \text{Sol}_{122}\})) =$$



$$B(\text{cons}(r, \{\text{Sol}_{11}, \text{Sol}_{121}, \text{Sol}_{122}\}), \underbrace{\{\text{Sol}_2\}}_{P(l_2)}) = B(r, \underbrace{\{\text{Sol}_{11}, \text{Sol}_{121}, \text{Sol}_{122}, \text{Sol}_2\}}_{P(l)})$$

Next assertion (.) is proved:

The proof splits into two parts: assuming $\forall l_1, l_2. \neg \text{is-set}(\text{cons}(l_1, l_2))$,

$$(1) (\text{is-probl-repr}(l) \wedge \neg \text{is-set}(r) \rightarrow B(l, r) \sqsubseteq B(r, P(l))), \dots \tag{5.4}$$

and

$$(2) (\text{is-probl-repr}(l) \wedge \neg \text{is-set}(r) \rightarrow B(r, P(l)) \sqsubseteq B(l, r), \dots \tag{5.5}$$

PROOF of (1): By course of values induction on the recursiondepth of B , i.e., we prove $(\text{is-probl-repr}(l) \wedge \neg \text{is-set}(r) \rightarrow B^{(n)}(l, r) \sqsubseteq B^{(n)}(r, P(l)))$, for all n .

We prove only the case $n > 2$; assume $\text{is-probl-repr}(r) \wedge \neg \text{is-set}(r)$.

Assume the result by hypothesis for $k < n$, and prove it for n :

(i) $\text{ext}(l)$:

$$B^{(n)}(l, r) = B^{(n-1)}(S_1 l, \text{cons}(S_2 l, r)) \quad (\text{hyp.}, \text{since } \neg \text{is-set}(\text{cons}(S_2 l, r)))$$

$$B^{(n-1)}(\text{cons}(S_2 l, r), P(S_1 l)) = B^{(n-2)}(S_2 l, \text{cons}^\nabla(r, P(S_1 l))) =$$

$$B^{(n-2)}(S_2 l, \text{cons}(r, P(S_1 l))) \sqsubseteq (\text{hyp.}) B^{(n-2)}(\text{cons}(r, P(S_1 l)), P(S_2 l)) =$$

$$B^{(n-3)}(r, \underbrace{P(S_1 l) \cup P(S_2 l)}_{P_l}) \sqsubseteq B^{(n)}(r, P_l).$$

(ii) $\neg \text{ext}(l)$:
 $B^{(n)}(l,r) = B^{(n-1)}(r, \text{Sol}(l)) = B^{(n-1)}(r, P(l)) \subseteq B^{(n)}(r, P(l)). \quad \square$

The proof of (2) is by induction on the recursion depth of $P(l)$ and is left as an exercise.

Note that equivalence (5.3) does not involve any notion of approximating complete solutions by partial solutions - the other distinctive feature of backtracking. This will be incorporated shortly into our correctness proof, by proving correctness of $P(l)$.

And now for something altogether different?: Iterative copying of binary trees!

(Due to Chris Wadsworth):

Assume we have characterised trees as being either a leaf (which is an atom, say) or a node,

which has a left-part, which is a tree, and a right-part, which is a tree.

Then the natural, recursive way to copy a binary tree is given by:

$\text{copy}(t) \leftarrow \text{if leaf}(t) \text{ then } (t)$
 $\text{else make-tree}(\text{copy}(\text{left}(t)), \text{copy}(\text{right}(t))).$

Alternatively we can copy trees iteratively by the following procedure which explicitly manipulates a stack (represented as a list):

$f(l,r) \leftarrow \text{if is-tree}(l)$
 $\text{then if leaf}(l) \text{ then } f(r,l)$
 $\text{else } f(\text{left}(l), \text{cons}(\text{right}(l), r))$
 $\text{else if } l = \text{endmarker}$
 $\text{then } r$
 $\text{else } f(\text{car}(l), \text{cons}^\nabla(\text{cdr}(l), r)),$
 where $\text{cons}^\nabla(l,f) \leftarrow \text{if is-tree}(l) \wedge \text{is-tree}(r) \text{ then make-tree}(l,r)$
 $\text{else cons}(l,r).$

I.E. tree-copying in iterative fashion is done by essentially the same scheme as B! As a result, our correctness proof of B implies that

$(\text{is-tree}(t) \rightarrow f(t, \text{endmarker}) = \text{copy}(t)),$

since the constants in the proof satisfy the same properties.

6. Correctness of the recursive characterization of backtracking for all solutions: a special case

Until now no attention has been paid to that general aspect of backtracking algorithms which involves approximating more specified solutions by means of less specified solutions. The fact that this property wasn't needed previously, constitutes the attractiveness of the proof-strategy followed in these sections: that a more complicated proof can be factored out in independent subproofs. Since proofs are always difficult enough in their own right, that is the situation we ideally strive for.

In this section we shall discuss correctness of the recursive characterization of backtracking for all solutions of the 8-queens' problem. The general case will be discussed lateron, and involves a straightforward abstraction of this example.

The recursive characterization of our version of the 8-queens' problem is:

$$P(\text{col},s) \leftarrow \text{if } \text{col} < 8 \wedge \text{free}(\text{col},s) \text{ then } \bigcup_{i=1}^8 P(\text{col}+1,i.s) \\ \text{else if } \text{col}=8 \wedge \text{free}(8,s) \text{ then } \{s\} \text{ else } \emptyset \text{ fi,}$$

with $\text{col} \in \{0, \dots, 8\}$, s denoting a linear list with $1, \dots, 8$ as atoms (possibly empty), and the value of $P(\text{col},s)$, if defined, a set consisting of linear lists; $\text{free}(\text{col},s)$ is a total predicate checking the particular constraints of the 8-queens' problem.

The call $P(0,\Lambda)$, with Λ denoting the empty linear list, is intended to have as value a set consisting of all solutions to the 8-queens' problem.

Correctness of P therefore amounts to proving that

(A1): $P(0,\Lambda)$ terminates,

(A2): $\forall s \in \text{Searchspace. } \underbrace{[(s \in P(0,\Lambda) \rightarrow \text{free}(8,s)]}_{\text{A2.1}} \wedge \underbrace{[\text{free}(8,s) \rightarrow s \in P(0,\Lambda)]}_{\text{A2.2}}]$

Condition A2.2 is necessary, since without it the empty set \emptyset as value for $P(0,\Lambda)$ would do.

The rest of this section is devoted to proving A2.1 and A2.2 in appropriate forms, using informal inductive proofs.

To formalize A2, one must describe the searchspace.

Define $R_1(\text{col},s)$ by

$$R_1(\text{col},s) \leftarrow \text{if } \text{col} < 8 \text{ then } \bigcup_{i=1}^8 R_1(\text{col}+1,i.s) \text{ else } \{s\} \text{ fi.}$$

Then the searchspace is the value of $R_1(0, \Lambda)$.

Therefore A2 is described by $\{t \mid t \in R_1(0, \Lambda) \wedge \text{free}(8, r)\} = P(0, \Lambda)$,

which is a consequence of the more general property

$$(A2.3): \{t \mid t \in R_1(\text{col}, s) \wedge \text{free}(8, r)\} = P(\text{col}, s).$$

To prove (A2.3) introduce an intermediate procedure $R_2(\text{col}, s)$:

$$R_2(\text{col}, s) \leftarrow \underline{\text{if}} \quad \text{col} < 8 \quad \underline{\text{then}} \quad \bigcup_{i=1}^8 R_2(\text{col}+1, i.s) \\ \underline{\text{else if}} \quad \text{free}(8, s) \quad \underline{\text{then}} \quad \{s\} \quad \underline{\text{else}} \quad \emptyset \quad \underline{\text{fi}},$$

satisfying $R_2(\text{col}, s) = \{t \mid t \in R_1(\text{col}, s) \wedge \text{free}(8, t)\}$, as proved below by simultaneous induction on the recursion depths of R_1 and R_2 :

We prove $R_2^{(n)}(\text{col}, s) = \{t \mid t \in R_1^{(n)}(\text{col}, s) \wedge \text{free}(8, t)\}$ for all n :

$$n=0: R_2^{(0)}(\text{col}, s) = \perp = \{t \mid t \in R_1^{(0)}(\text{col}, s) \wedge \text{free}(8, t)\}.$$

$n > 0$: assume the result by hypothesis for $n-1$, then

$$R_2^{(n)}(\text{col}, s) = \underline{\text{if}} \quad \text{col} < 8 \quad \underline{\text{then}} \quad \bigcup_{i=1}^8 R_2^{(n-1)}(\text{col}+1, i.s) \quad \underline{\text{else}} \\ \underline{\text{if}} \quad \text{free}(8, s) \quad \underline{\text{then}} \quad \{s\} \quad \underline{\text{else}} \quad \emptyset \quad \underline{\text{fi}} \\ = (\text{hyp.}) \quad \underline{\text{if}} \quad \text{col} < 8 \quad \underline{\text{then}} \quad \bigcup_{i=1}^8 \{t \mid t \in R_1^{(n-1)}(\text{col}+1, i.s) \wedge \text{free}(8, t)\} \\ \underline{\text{else if}} \quad \text{free}(8, s) \quad \underline{\text{then}} \quad \{s\} \quad \underline{\text{else}} \quad \emptyset \quad \underline{\text{fi}} \\ = \{t \mid t \in \underline{\text{if}} \quad \text{col} < 8 \quad \underline{\text{then}} \quad \bigcup_{i=1}^8 R_1^{(n-1)}(\text{col}+1, i.s) \\ \underline{\text{else}} \quad \{s\} \quad \underline{\text{fi}} \wedge \text{free}(8, t)\} \\ = \{t \mid t \in R_1^{(n)}(\text{col}, s) \wedge \text{free}(8, t)\}. \quad \square$$

Thus the proof of A2.3 reduces to:

$$(A2.4) \quad P(\text{col}, s) = R_2(\text{col}, s).$$

Now we need the 2nd characteristic of backtracking algorithms, in this case concerning the total predicate $\text{free}(\text{col}, s)$:

$$\forall i, s. \text{col} < 8 \rightarrow (\neg \text{free}(\text{col}, s) \rightarrow \neg \text{free}(\text{col}+1, i.s)), \dots \quad (6.1)$$

i.e., no more complete partial solution can be obtained from s , once $\neg \text{free}(\text{col}, s)$ holds. This property will be characterized, and proved, below as:

$$\neg \text{free}(\text{col}, s) \rightarrow R_2(\text{col}, s) \sqsubseteq \emptyset.$$

The proof of A2.4 splits into two parts:

(A2.5): if $R_2(\text{col}, s)$ terminates with value y , $P(\text{col}, s)$ also terminates and $y = P(\text{col}, s)$; this is expressed by $R_2(\text{col}, s) \sqsubseteq P(\text{col}, s)$.

(A2.6): The procedure R_2 is total, i.e., terminates always with a well defined value; then $R_2(\text{col}, s) = P(\text{col}, s)$, i.e., (A2.4) follows from (A2.5).

PROOF of (A2.5):

Applying n -step- $n+1$ induction, we prove simultaneously for all n

$\forall \text{col}, s. [R_2^{(n)}(\text{col}, s) \subseteq P^{(n)}(\text{col}, s)]. \wedge \forall \text{col}, s. [\neg \text{free}(\text{col}, s) \rightarrow R_2^{(n)}(\text{col}, s) \subseteq \emptyset].$

(i) $n = 0$: obvious.

(ii) $n > 0$: assume the result by hypothesis for $n-1$ then we prove:

(iia) $\forall \text{col}, s. \underbrace{\text{if } \text{col} < 8 \text{ then } \bigcup_{i=1}^8 R_2^{(n-1)}(\text{col}+1, i.s) \text{ else if } \text{free}(8, s) \text{ then } \{s\} \text{ else } \emptyset \text{ fi}}_{E_2}$

$\subseteq \underbrace{\text{if } \text{col} < 8 \wedge \text{free}(\text{col}, s) \text{ then } \bigcup_{i=1}^8 P^{(n-1)}(\text{col}+1, i.s) \text{ else if } \text{col} = 8 \wedge \text{free}(8, s) \text{ then } \{s\} \text{ else } \emptyset \text{ fi}}_E,$

i.e., $\forall \text{col}, s. E_2 \subseteq E,$

and

(iib) $\forall \text{col}, s. (\neg \text{free}(\text{col}, s) \rightarrow E_2 \subseteq \emptyset).$

Proof of (iia):

Of the 4 conditions which $\langle \text{col}, s \rangle$ may satisfy - $\text{col} < 8 \wedge \neg \text{free}(\text{col}, s)$, $\text{col} < 8 \wedge \text{free}(\text{col}, s)$, $\text{col} = 8 \wedge \neg \text{free}(8, s)$, $\text{col} = 8 \wedge \text{free}(8, s)$ - only the first one might give rise to different values for E_2 and E , since the second one yields by the inductive hypothesis that

$$E_2 = \bigcup_{i=1}^8 R_2^{(n-1)}(\text{col}+1, i.s) \subseteq (\text{hyp.}) \bigcup_{i=1}^8 P^{(n-1)}(\text{col}+1, i.s) = E,$$

the third one yields $E_2 = \emptyset \subseteq \emptyset = E$, and the fourth one $E_2 = \{s\} \subseteq \{s\} = E$.

By () one has:

Assume $\text{col} < 8 \wedge \neg \text{free}(\text{col}, s)$: then

$$E_2 = \bigcup_{i=1}^8 R_2^{(n)}(\text{col}+1, i.s) \subseteq \{\text{by hyp., since } \neg \text{free}(\text{col}, s) \text{ implies } \neg \text{free}(\text{col}+1, i.s)\} \bigcup_{i=1}^8 \emptyset = \emptyset \subseteq \emptyset = E. \quad \square$$

Proof of (iib): Assume $\neg \text{free}(\text{col}, s)$. For $\text{col} < 8$ one has by that $\neg \text{free}(\text{col}+1, i.s), i=1, \dots, 8$, and hence

$$E_2 = \bigcup_{i=1}^8 R_2^{(n-1)}(\text{col}+1, i.s) \subseteq (\text{hyp.}) \bigcup_{i=1}^8 \emptyset = \emptyset.$$

For $\text{col} = 8$, $E_2 = \emptyset \subseteq \emptyset.$ □

Proof of (A2.6)

Order $\{0, 1, \dots, 8\}$ by $0 > 1 > 2 > 3 > 4 > 5 > 6 > 7 > 8$.

Nontermination of R_2 originates from the following situations (for procedures with parameters called-by-value):

- (i) Infinite recursive regression: i.e., an infinitely going on computation due to an infinite number of inner recursive calls. This cannot occur since the transformation of (intermediate) values between a call and a constituent inner recursive call of the procedure body are

$$\langle \text{col}, s \rangle \rightarrow \langle \text{col}+1, i.s \rangle, \quad i=1, \dots, 8, \quad \text{i.e. } S_1, \dots, S_8,$$

and this decreases the first component of the state w.r.t. the order $>$ as defined above. Hence, since this order is finite, no infinite chain of values $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ exists of these intermediate values since the least element of $>$ is reached in finite time. Once this happens, no inner recursive call occurs any more, since, if so, this would result in decreasing the least element of $>$ w.r.t. $>$: contradiction! But this means that the recursive procedure terminates.

- (ii) At some finite recursion depth some elementary operation gets undefined: this cannot occur either, since either the predicates involved are total, or the selection made by these predicates precludes undefinedness of the elementary operations; e.g., if $\text{col} < 8$ then $S_i(\text{col}, s)$ is defined, for $i=1, \dots, 8$.

Remark: Note that in the correctness proof above, predicates and operations are split into two groups:

- (i) $\text{col} > 8, \text{col}=8, S_i$, defining the search-space, and playing already a critical rôle in proving correctness of the iterative description of the recursive search,
- (ii) $\text{free}(\text{col}, s)$, satisfying the approximation property 6.1, and playing a rôle only in correctness of the recursive search itself.

Moreover (i) involves predicates and operations which give the search-space a finite-tree structure, as follows from A2.6, above, and (ii) is used to prune this tree-structured search-space. Thus, returning to the binary version of P, schematically P has the following form:

$$P(l) \Leftarrow \underline{\text{if}} \quad \text{ext}(l) \wedge \text{free}(l) \quad \underline{\text{then}} \quad P(S_1(l)) \cup P(S_2(l)) \\ \quad \underline{\text{else if}} \quad \text{free}(l) \quad \underline{\text{then}} \quad \{l\} \quad \underline{\text{else}} \quad \emptyset \quad \underline{\text{fi.}}$$

7. The general case: correctness of a recursive problem specification

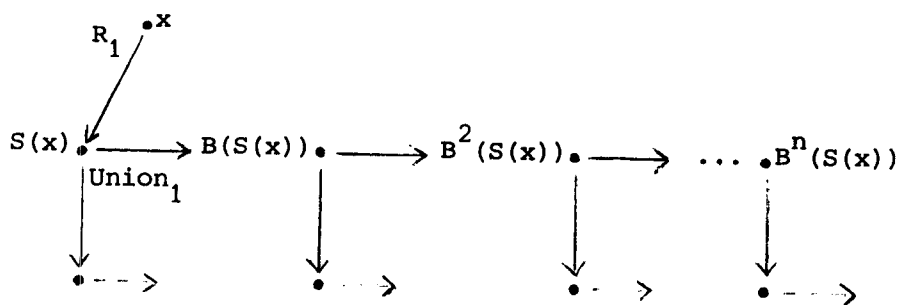
In the general case of backtracking for all solutions to a given problem, the number of sons to a father node may vary from father to father.

Therefore we introduce the following recursive characterization of the search-space:

$$\left. \begin{aligned} R_1(x) &\leftarrow \underline{\text{if}} \text{ ext}(x) \underline{\text{then}} R_1(S(x)) \cup \text{Union}_1(S(x)) \underline{\text{else}} \{x\} \underline{\text{fi}}, \\ \text{Union}_1(x) &\leftarrow \underline{\text{if}} \text{ brother}(x) \underline{\text{then}} R_1(B(x)) \cup \text{Union}_1(B(x)) \underline{\text{else}} \emptyset \underline{\text{fi}}, \end{aligned} \right\}$$

where S is a mnemonic for Son, and B for Brother.

The following picture might apply if $\text{ext}(x)$ holds



Notice that R_1 develops the search-space in-depth, and Union_1 in-breadth. We shall assume that $\text{ext}(x)$ and $\text{brother}(x)$ are total predicates which, if evaluating to true, guarantee that $S(x)$ and $B(x)$, respectively, are well-defined.

The approximation property of $\text{free}(x)$ that

$$\text{ext}(s) \wedge \neg \text{free}(x) \rightarrow \neg \text{free}(B^k(S(x)))$$

for appropriate k , is expressed using the procedure

$$\left. \begin{aligned} \text{Sons}(x) &\leftarrow \underline{\text{if}} \text{ ext}(x) \underline{\text{then}} \{S(x)\} \cup \text{Brothers}(S(x)) \underline{\text{else}} x \underline{\text{fi}}, \\ \text{Brothers}(x) &\leftarrow \underline{\text{if}} \text{ brother}(x) \underline{\text{then}} \{B(x)\} \cup \text{Brothers}(B(x)) \underline{\text{else}} \emptyset \underline{\text{fi}}, \end{aligned} \right\}$$

and amounts to:

$$\text{ext}(x) \wedge \neg \text{free}(x) \wedge y \in \text{Sons}(x) \rightarrow \neg \text{free}(y).$$

Consequently one has

$$\{y/y \in R_1(x) \wedge \text{free}(y)\} = R_2(x), \dots$$

with R_2 declared by

$$\left. \begin{aligned} R_2(x) &\leftarrow \underline{\text{if}} \text{ ext}(x) \underline{\text{then}} R_2(S(x)) \cup U_2(S(x)) \\ &\quad \underline{\text{else if}} \text{ free}(x) \underline{\text{then}} \{x\} \underline{\text{else}} \emptyset \underline{\text{fi}}, \\ U_2(x) &\leftarrow \underline{\text{if}} \text{ brother}(x) \underline{\text{then}} R_2(B(x)) \cup U_2(B(x)) \underline{\text{else}} \emptyset \underline{\text{fi}}. \end{aligned} \right\}$$

Assertion requires proving
 for all n:
$$\left. \begin{aligned} R_2^{(n)}(x) &= \{y/y \in R_1^{(n)}(x) \wedge \text{free}(y)\} \& \\ U_2^{(n)}(x) &= \{y/y \in \text{Union}_1(x) \wedge \text{free}(y)\}; \end{aligned} \right\}$$

this is done (simultaneously) using n-step-n+1 induction.

However, prior to doing so, a number of questions regarding or formalism should be answered.

If Sons(x) does not terminate, its (formal) value is (represented by) $\perp_{\text{FIN(Nodes)}}$, where FIN(Nodes) denotes the collection of finite sets of possible nodes and atoms, and $\perp_{\text{FIN(Nodes)}}$ is the formal value, expressing nontermination, added to FIN(Nodes).

Hence we should give an appropriate truth-value to e.g., $y \in \perp_{\text{FIN(Nodes)}}$.

We want the operator $\dots \in \dots$ to be monotone in both its arguments.

The reason for this is that should imply

$$\begin{aligned} \cup R_2^{(n)}(x) &= \{y/y \in \cup R_1^{(n)}(x) \wedge \text{free}(y)\} \& \\ \cup U_2^{(n)}(x) &= \{y/y \in \cup \text{Union}_1^{(n)}(x) \wedge \text{free}(y)\}. \end{aligned}$$

I.e., $\dots \in \dots$ should be continuous in its arguments. However, since the only nontrivial chain of different elements $\alpha_1 \not\subseteq \alpha_2 \not\subseteq \alpha_3 \dots \not\subseteq \alpha_k$ in $\{\perp_{\text{FIN(Nodes)}}\} \cup \text{FIN(Nodes)}$ are those with $k=2$, $\alpha_1 = \perp$, $\alpha_2 \in \text{FIN(Nodes)}$, continuity of $\dots \in \dots$ in its arguments follows from monotonicity, i.e., from $(x_1 \subseteq x_2, \alpha_1 \subseteq \alpha_2) \rightarrow (x_1 \in \alpha_1 \subseteq x_2 \in \alpha_2)$.

Hence we should give $y \in \perp_{\text{FIN(Nodes)}}$ such a truth value, that, e.g., $y \in \perp_{\text{FIN(Nodes)}} \subseteq y \in \emptyset$ and $y \in \perp_{\text{FIN(Nodes)}} \subseteq y \in \{y\}$. This implies that $(y \in \perp_{\text{FIN(Nodes)}}) = \perp_{\{\text{true}, \text{false}\}}$ where $\perp_{\{\text{true}, \text{false}\}} \subseteq \text{false}$, $\perp_{\{\text{true}, \text{false}\}} \subseteq \text{true}$, denotes the formal element added to $\{\text{true}, \text{false}\}$, and similarly that $(\perp \in \dots) = \perp_{\{\text{true}, \text{false}\}}$.

Consequently we should consider a 3-valued logic, and, in particular, define the operators $\wedge, \rightarrow, \neg$ for $\perp_{\{\text{true}, \text{false}\}}$, too.

Again, one wishes \wedge, \rightarrow , and \neg to be monotone in their arguments, for similar reasons as stated above for \in , and illustrated further on.

This implies that one has the following truth-tables for $\wedge, \rightarrow, \neg$:

\wedge	<u>true</u>	<u>false</u>	\perp	\rightarrow	<u>true</u>	<u>false</u>	\perp	\neg	<u>true</u>	<u>false</u>	\perp
<u>true</u>	<u>true</u>	<u>false</u>	\perp	<u>true</u>	<u>true</u>	<u>false</u>	\perp		<u>false</u>	<u>true</u>	\perp
<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>	<u>true</u>				
\perp	\perp	<u>false</u>	\perp	\perp	<u>true</u>	\perp	\perp				

Finally the set formation operator $\{\dots / \text{condition}(\dots)\}$ should be extended by $\{\dots / \perp_{\{\underline{\text{true}}, \underline{\text{false}}\}}\} = \perp_{\text{FIN}(\text{Nodes})}$, in order to guarantee monotonicity in its second argument,

and the union operator $\dots \cup \dots$ by $\perp_{\text{FIN}(\text{Nodes})} \cup \dots = \dots \cup \perp_{\text{FIN}(\text{Nodes})} = \perp_{\text{FIN}(\text{Nodes})}$

Now we are in a position to deduct properly from :

$$R_2^{(n)}(x) = \{y / y \in R_1^{(n)}(x) \wedge \text{free}(y)\} \text{ for all } n \Rightarrow$$

$$\sqcup R_2^{(n)}(x) = \sqcup \{y / y \in R_1^{(n)}(x) \wedge \text{free}(y)\}, \text{ since } R_2^{(n)}(x) \sqsubseteq R_2^{(n+1)}(x)$$

for all n guarantees existence of least upperbounds,

\Rightarrow by monotonicity (continuity) of $\{\dots / \dots\}$,

$$\sqcup R_2^{(n)}(x) = \{y / \sqcup (y \in R_1^{(n)}(x) \wedge \text{free}(y))\}$$

\Rightarrow by monotonicity (continuity) of \wedge ,

$$\sqcup R_2^{(n)}(x) = \{y / (\sqcup (y \in R_1^{(n)}(x))) \wedge \text{free}(y)\}$$

\Rightarrow by monotonicity (continuity) of \in ,

$$\sqcup R_2^{(n)}(x) = \{y / y \in (\sqcup R_1^{(n)}(x)) \wedge \text{free}(y)\}$$

\Rightarrow by the least fixed point characterization of recursive procedures

$$R_2(x) = \{y / y \in R_1(x) \wedge \text{free}(y)\}.$$

Next we prove

$$R_2(x) \sqsubseteq R_3(x), \dots \tag{7.1}$$

with R_3 defined by

$$\left. \begin{aligned} R_3(x) &\leftarrow \underline{\text{if}} \text{ ext}(x) \wedge \text{free}(x) \underline{\text{then}} R_3(S(x)) \cup U_3(S(x)) \\ &\quad \underline{\text{else if}} \text{ free}(x) \underline{\text{then}} \{x\} \underline{\text{else}} \emptyset \underline{\text{fi}}, \\ U_3(x) &\leftarrow \underline{\text{if}} \text{ brother}(x) \underline{\text{then}} R_3(B(x)) \cup U_3(B(x)) \underline{\text{else}} \emptyset \underline{\text{fi}} \end{aligned} \right\}$$

Note that one cannot prove $R_2(x) = R_3(x)$ since $R_2(x)$ might specify an infinite search-space, and hence $R_2(x) = \perp$, while free might prune this infinite search-space to a finite one, and hence $R_3(x) \in \text{FIN}(\text{Nodes})$.

Equality only holds when $R_2(x)$ terminates. And termination of $R_2(x)$ depends on well-foundedness of the relational union SUB of S and B in x , i.e., that there exists no infinite sequence $\{x_i\}$ with $x_i \neq \perp$ s.t. $x_{i+1} = S(x_i)$ or $x_{i+1} = B(x_i)$, starting in x ($x_0 = x$).

Assertion follows from:

$$\begin{aligned} \text{for all } n, \quad R_2^{(n)}(x) \subseteq R_3^{(n)}(x), \\ U_2^{(n)}(x) \subseteq U_3^{(n)}(x), \\ (\neg \text{free}(x) \wedge y \in \text{Sons}(x) \rightarrow R_2^{(n)}(y) \subseteq \emptyset), \\ (\neg \text{free}(x) \wedge y \in \text{Sons}(x) \rightarrow U_2^{(n)}(y) \subseteq \emptyset), \end{aligned}$$

which follows easily by (simultaneous) n -step- $n+1$ induction. The proof is analogous to that of A2.5 above.

Observe again that proving requires proving a lot more, about R_2 itself, and about U_2 and U_3 .

Thus one establishes correctness of the recursive search for all solutions, as expressed by R_3 and U_3 .

The general case: correctness of the iterative search procedure

First we define a general backtracking procedure $BT(l,r)$:

$$\begin{aligned} BT(l,r) \leftarrow & \text{if } \text{is-probl-repr}(l) \text{ then} \\ & \text{if } \text{ext}(l) \wedge \text{free}(l) \text{ then } BT(S(l), \text{CONC}(S(l), r)) \\ & \text{else } BT(r, \text{if } \text{free}(l) \text{ then } \{l\} \text{ else } \emptyset \text{ fi}) \text{ fi} \\ & \text{else if } l = \text{endmarker} \text{ then } r \\ & \text{else } BT(\text{car}(l), \text{cons}^\nabla(\text{cdr}(l), r)) \text{ fi}, \end{aligned}$$

$$\begin{aligned} \text{CONC}(l,r) \leftarrow & \text{if } \text{brother}(l) \text{ then } \text{cons}(B(l), \text{CONC}(B(l), r)) \text{ else } r \text{ fi}, \\ \text{cons}^\nabla(l,r) \leftarrow & \text{if } \text{is-set}(l) \wedge \text{is-set}(r) \text{ then } l \cup r \text{ else } \text{cons}(l,r) \text{ fi}. \end{aligned}$$

Here l and r denote binary trees over atoms which are either sets - characterized by the predicate $\text{is-set}(l)$ - or, disjunct with these, problem representations - characterized by the predicate $\text{is-probl-repr}(l)$ - or, disjunct with these and the sets, the endmarker.

We must prove that R_3 and BT satisfy

$$\text{is-probl-repr}(l) \rightarrow BT(l, \text{endmarker}) = R_3(l);$$

this follows from

$$\text{is-probl-repr}(l) \rightarrow BT(l,r) = BT(r, R_3(l)), \dots \quad (7.2)$$

By way of example we give an outline of the \subseteq -part of this proof, which bears some analogy with the proofs about Q and B given previously. In this case we must simultaneously prove and below:

$$(\text{is-probl-repr}(l) \wedge \text{is-set}(t) \rightarrow BT(\text{CONC}(l,r), t) = BT(r, t \cup U_3(l))) \dots \quad (7.3)$$

PROOF of 7.2 and 7.3: outline:

Assume $\text{is-probl-repr}(l)$.

I.a. $\text{ext}(l) \wedge \text{free}(l)$ holds:

$$\begin{aligned} \text{BT}^{(n)}(l, r) &= \text{BT}^{(n-1)}(S(l), \text{CONC}(S(l), r)) \subseteq (\text{hyp.}) \\ &\quad \text{BT}^{(n-1)}(\text{CONC}(S(l), r), R_3(S(l))) \subseteq (\text{hyp.}) \\ &\quad \text{BT}^{(n-1)}(r, R_3(S(l)) \cup U_3(S(l))) \subseteq \text{BT}^{(n)}(r, R_3(l)). \end{aligned}$$

b. $\neg \text{ext}(l) \vee \neg \text{free}(l)$ holds.

$$\text{BT}^{(n)}(l, r) = \text{BT}^{(n)}(r, \text{if free}(l) \text{ then } \{l\} \text{ else } \emptyset \text{ fi}) \subseteq \text{BT}^{(n)}(r, R_3(l)).$$

Assume $\text{is-set}(t)$, too.

II.a. $\text{brother}(l)$:

$$\begin{aligned} \text{BT}^{(n)}(\text{CONC}(l, r), t) &= \text{BT}^{(n)}(\text{cons}(B(l), \text{CONC}(B(l), r)), t) = \\ &\quad \text{BT}^{(n-1)}(B(l), \text{cons}(\text{CONC}(B(l), r), t)) \subseteq (\text{hyp.}) \\ &\quad \text{BT}^{(n-1)}(\text{cons}(\text{CONS}(B(l), r), t), R_3(B(l))) = \\ &\quad \text{BT}^{(n-2)}(\text{CONC}(B(l), r), R_3(B(l)) \cup t) \subseteq (\text{hyp.}) \\ &\quad \text{BT}^{(n-2)}(r, R_3(B(l)) \cup t \cup U_3(B(l))) = (\text{associativity of } U) \\ &\quad \text{BT}^{(n-1)}(r, t \cup R_3(B(l)) \cup U_3(B(l))) \subseteq \\ &\quad \text{BT}^{(n)}(r, t \cup R_3(l)). \end{aligned}$$

b. $\neg \text{brother}(l)$:

$$\begin{aligned} \text{BT}^{(n)}(\text{CONC}(l, r), t) &= \text{BT}^{(n)}(r, t) = \text{BT}^{(n)}(r, t \cup \emptyset) = \\ &\quad \text{BT}^{(n)}(r, t \cup U_3(l)). \end{aligned}$$

□

I would not have found the above proofs without having studied the simpler versions for Q and B , and recognizing the similarities.

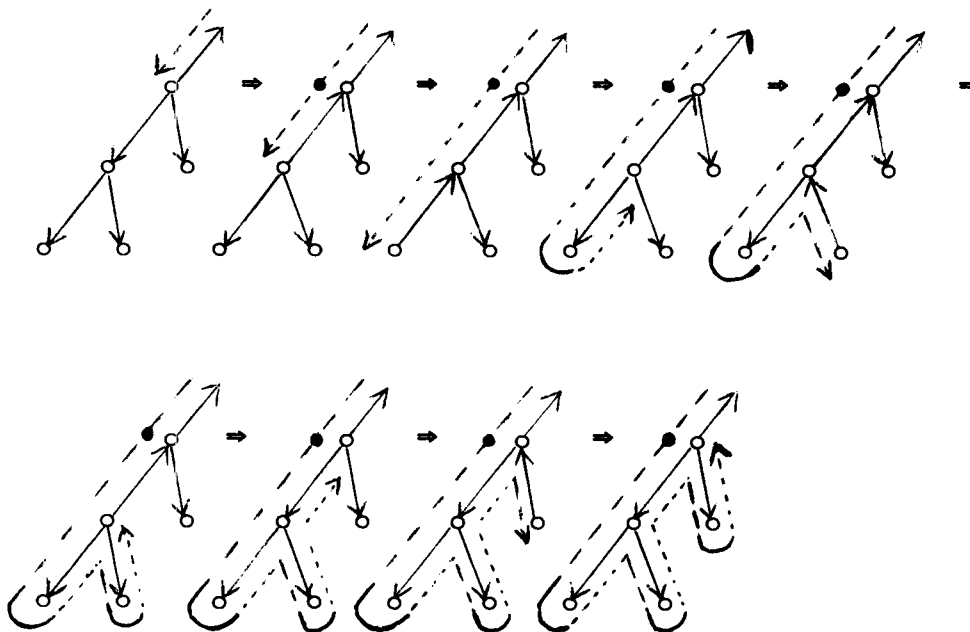
Résumé

- First I defined the search-space recursively, using R_1 .
- Then I proved correctness of an intermediate recursive problem specification R_2 using a predicate free having certain approximation properties.
- Thirdly I proved correctness of the final recursive problem specification R_3 , which pruned the recursively defined search-space using free. Also it has been argued that termination of the general version of the back tracking problem is a consequence of a well defined search-space, which guarantees termination; hence one should prove termination for particular applications.
- Fourthly, I described equivalence between R_3 and the iterative problem solution BT - the backtracking algorithm proper.
- These four steps constitute proving correctness of $BT(1, \text{endmarker})$. Notice how many additional properties, generalizations, auxiliary procedures were needed, and observe how little the perspective is on any chance of automating such a proof.

8. THE DEUTSCH-SCHORR-WAITE marking algorithm: first step

The Deutsch-Schorr-Waite marking algorithm for rooted, binary, digraphs - that binary is no restriction, really, should be clear to the reader who understood the general version of the backtracking algorithm - is obtained after two steps.

In this section the first step is described: the version for binary trees. The secret behind this algorithm is to build in a notion of direction into the arrow reversal game. I.e., one distinguishes between a leftdown, and a back-up phase, as indicated in the drawing below:



Traversal in leftdown phase is indicated by \dashrightarrow , and traversal in back-up phase by \dashleftarrow .

Why this differentiation into two phases? Well, new nodes are encountered in leftdown phase, while already encountered nodes are revisited in back-up phase.

Hence upon extending the algorithm to digraphs, one must avert the danger of infinite traversal of a cycle by checking in leftdown phase upon nodes encountered previously at the beginning of a cycle. This can be done by introducing a marking bit in each interior node, setting this bit when the node is encountered for the first time, and checking upon the value of this bit in leftdown phase (, since all nodes encountered in back-up phase have been visited already anyhow).

(The sceptical reader may observe that, although the strategy outlined above certainly results in traversal of the digraph, this fact on itself doesn't prove the necessity of introducing the two phases described above. And indeed, adaptation of algorithm Q to digraphs does not result in infinite traversal of cycles; later on we shall indicate why this adaptation of algorithm Q is ineffective in that one cannot indicate when Q has to terminate.)

Traversal in leftdown phase ends upon encountering an atom, upon which traversal in back-up phase is initiated.

Traversal in back-up phase changes into leftdown phase upon encountering a (not yet visited) righthand subtree of the original tree, as indicated above, and remains in back-up phase, otherwise.

Consequently the two phases do not alternate. So, upon reaching a node, how is one to decide whether to remain in the same phase, or to change into another phase?

In an external node (atom/leaf) there is no problem:

--->o...>...

But an internal node is visited thrice:

1st time: --->---o--->---



2nd time: ...>...o--->---

3rd time: ...>...o...>...



The solution is to introduce (again) a marking but in each interior node: Meeting an interior node for the first time in leftdown phase sets this bit to, say, 1:

1st time: --->  o ---> ---

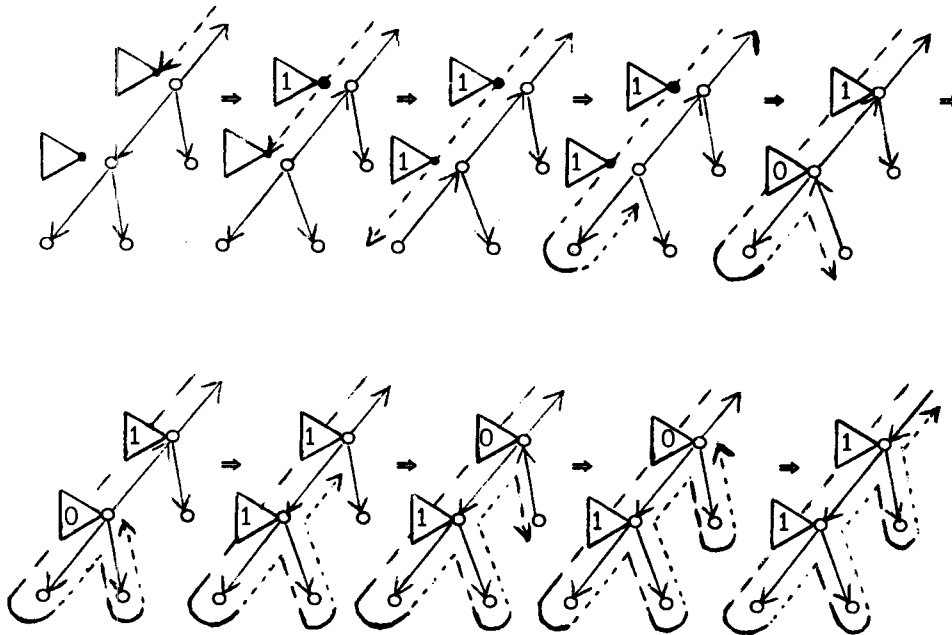
Meeting this node next in back-up phase, i.e., with marking bit set to 1, changes the marking bit into 0:

2nd time: ..>  o => ..>  o ---> ---

Meeting this node for the third time, i.e., in back-up phase with bit set to 0, indicates that the back-up phase has to continue, after which the bit has outlived its purpose of phase-marking, and can be safely used as marking bit, setting it (uniformly) to either 0 or 1:

3rd time: ..>  o => ..>  o ...> ..

This is indicated in the following figure:



This idea is reflected in the following marking algorithm for binary trees (with one bit in each interior node since no danger of cycle-traversal is present, as yet):

```

Left(l,r) ← if at(l) then Back(l,r)
             else Left(car(l),cons(cdr(l),r,1)) fi,
Back(l,r) ← if r = NIL then l else
             if bitfield(r) = 1 then Left(car(r),cons(cdr(r),l,0))
             else Back(cons(cdr(r),l,1),car(r)) fi,

```

where NIL serves as endmarker, and bitfield(r) isolates the marking bit of r.

Let $M(l) \leftarrow \text{if } \text{at}(l) \text{ then } l \text{ else } \text{cons}(M(\text{car}(l)), M(\text{cdr}(l)), 1) \text{ fi}$
denote the obvious recursive version of the marking algorithm.
Then the following assertion holds both for finite and infinite trees l
(- cons calls its arguments by value):

$$\text{Left}(l,r) = \text{Back}(M(l),r).$$

These algorithms have been investigated independently by Burstall, and by de Roever, and their correctness proof is similar to that of algorithm Q (cfr. pages 18 and 20).

An alternative way of arriving at Left and Back is by specifying Left and Back by the assertion given above, and deducing ("synthesizing") the text of Left and Back from this specification.

THE DEUTSCH-SCHORR-WAITE MARKING ALGORITHM: second step

In the above algorithm new nodes of the tree are encountered for the first time in leftdown phase, and are then submitted to a test of the form $at(l)$. Therefore, in case of traversal of a cycle of a binary digraph, that test is the spot in the algorithm where newly encountered nodes have to be distinguished from already visited ones in order to prevent infinite repetition in traversal of that cycle.

This distinction is made by introducing a second marking bit which enables marking a node upon encountering it for the first time, and by replacing the test $at(l)$ by a test $at(l) \vee \text{already visited}(l)$.

Next the underlying formalism is changed, for the following reason: We intend to traverse a binary digraph in situ: I.e., by making the changes suggested above at precisely those locations which are encountered while traversing that digraph.

This is not reflected by the $at, car, cdr, cons$ -based formalism used above: E.g., a perfectly valid implementation for $cons$ is one in which every new application of $cons(l,r)$ results in fetching an address of a new node from some free list, and encoding l and r inside the car - and cdr -fields of that new node.

In the present section, upon encountering an interior node with address α we intend to overwrite the contents of that node at α destructively, i.e., execution of our algorithm results in a side-effect upon the memory.

In an, e.g., PASCAL-based notation this would result in the introduction of records, and pointers l and r to those records; the algorithm would have two such pointers as parameters, and execution would be expressed as a side-effect upon the "memory" (set) consisting of these records. This is left to the reader.

We intend to prove correctness of the algorithm, and hence have to quantify the side-effect upon the memory explicitly by introducing the memory as parameter to the procedures. Such is the objective of the formalism introduced below (cf. Topor):

Let At and Loc denote two disjoint sets, and at denote a total predicate over these sets satisfying $at(\alpha) = \underline{\text{true}}$ iff $\alpha \in At$.

A memory for representing binary directed graphs with two marking bits is a total function $\alpha: Loc \rightarrow \{0,1\}^2 \times (Loc \cup At)^2$.

Changes of such a memory are described as follows:

Let $a_1, a_2 \in \{0,1\}$, and $a_3, a_4 \in (\text{Loc} \cup \text{At})$, then $\sigma[\alpha \rightarrow a_1, a_2, a_3, a_4]$ is for $\alpha \in \text{Loc}$ defined by

$$\lambda \beta \in \text{Loc}. \underline{\text{if}} \alpha = \beta \underline{\text{then}} \langle a_1, a_2, a_3, a_4 \rangle \underline{\text{else}} \sigma(\beta) \underline{\text{fi}}.$$

For $\alpha \in \text{Loc}$, elements of the quadruple $\sigma(\alpha)$ are accessed by the (total) functions

already visited, $f: \text{Loc} \times \text{Mem} \rightarrow \{0,1\}$ and

$\text{hd}, \text{tl}: \text{Loc} \times \text{Mem} \rightarrow \text{Loc} \cup \text{At}$, with Mem denoting the collection of memories as defined above, and already visited, f, hd, tl defined by:

If $\alpha \in \text{Loc}, \sigma \in \text{Mem}$, and $\sigma(\alpha) = \langle a_1, a_2, a_3, a_4 \rangle$, then

already visited(α, σ) = a_1 , $f(\alpha, \sigma) = a_2$, $\text{hd}(\alpha, \sigma) = a_3$, $\text{tl}(\alpha, \sigma) = a_4$.

In view of the above, the Deutsch-Schorr-Waite marking algorithm of binary directed graphs, expressed by $\text{LEFT}(\alpha, \beta, \sigma)$ and $\text{BACK}(\alpha, \beta, \sigma)$ below, should now be obvious:

$$\begin{aligned} \text{LEFT}(\alpha, \beta, \sigma) &\leq \underline{\text{if}} \text{at}(\alpha) \vee \text{already visited}(\alpha, \sigma) = 1 \underline{\text{then}} \text{BACK}(\alpha, \beta, \sigma) \\ &\quad \underline{\text{else}} \\ &\quad \text{LEFT}(\text{hd}(\alpha, \sigma), \alpha, \sigma[\alpha \rightarrow 1, 1, \text{tl}(\alpha, \sigma), \beta]) \underline{\text{fi}}, \end{aligned}$$

$$\begin{aligned} \text{BACK}(\alpha, \beta, \sigma) &\leq \underline{\text{if}} \beta = \text{NIL} \underline{\text{then}} \langle \alpha, \sigma \rangle \underline{\text{else}} \\ &\quad \underline{\text{if}} f(\beta, \sigma) = 1 \underline{\text{then}} \\ &\quad \text{LEFT}(\text{hd}(\beta, \sigma), \beta, \sigma[\beta \rightarrow \text{m}(\beta, \sigma), 0, \text{tl}(\beta, \sigma), \alpha]) \\ &\quad \underline{\text{else}} \\ &\quad \text{BACK}(\beta, \text{hd}(\beta, \sigma), \sigma[\beta \rightarrow 1, 1, \text{tl}(\beta, \sigma), \alpha]) \underline{\text{fi}}. \end{aligned}$$

Let

$$\begin{aligned} \text{M}(\alpha, \sigma) &\leq \underline{\text{if}} \text{at}(\alpha) \vee \text{already visited}(\alpha, \sigma) = 1 \underline{\text{then}} \sigma \underline{\text{else}} \\ &\quad \text{M}(\text{tl}(\alpha, \sigma), \text{M}(\text{hd}(\alpha, \sigma), \sigma[\alpha \rightarrow 1, f(\alpha, \sigma), \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)])) \underline{\text{fi}}, \end{aligned}$$

denote the obvious recursive version of the marking procedure.

Then correctness of the Deutsch-Schorr-Waite marking algorithm for binary directed graphs is expressed by

$$\text{LEFT}(\alpha, \text{NIL}, \sigma) = \langle \alpha, \text{M}(\alpha, \sigma) \rangle, \dots$$

a special case of

$$\text{LEFT}(\alpha, \beta, \sigma) = \text{BACK}(\alpha, \beta, \text{M}(\alpha, \beta)).$$

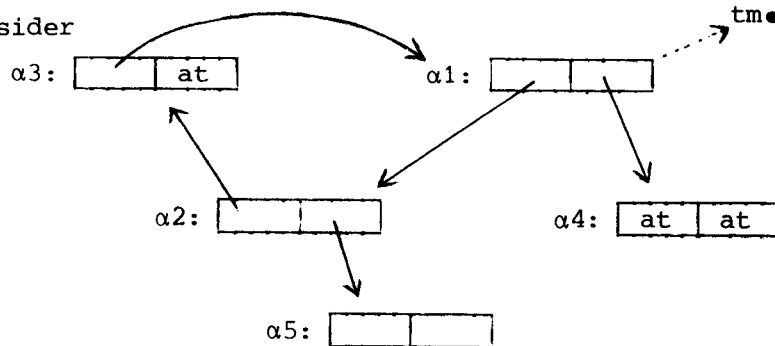
Insert: why algorithm Q cannot be adapted to traversal of binary digraphs:
one does not know when to terminate.

Consider

$$Q(\alpha, \beta, \sigma) \leftarrow \begin{array}{l} \text{if } \neg \text{at}(\alpha) \text{ then } Q(\text{hd}(\alpha, \sigma), \alpha, \sigma[\alpha:\text{tl}(\alpha, \sigma), \beta]) \\ \text{else if } \neg \text{tm}(\alpha) \text{ then } Q(\beta, A(\alpha), \sigma) \text{ else ? fi} \end{array}$$

We did not specify what Q does when it hits the termination marker; this has the following reason:

Consider



$Q(\alpha 1, \text{tm}, \sigma)$ leads to the following sequence of intermediate results:

$\langle \alpha 1, \text{tm}, \sigma \rangle \Rightarrow \langle \alpha 2, \alpha 1, \sigma[\alpha 1:\alpha 4, \text{tm}] \rangle \Rightarrow \langle \alpha 3, \alpha 2, \sigma[\alpha 1:\alpha 4, \text{tm}][\alpha 2:\alpha 5, \alpha 1] \rangle \Rightarrow$

$\langle \alpha 1, \alpha 3, \sigma[\alpha 1:\alpha 4, \text{tm}][\alpha 2:\alpha 5, \alpha 1][\alpha 3:\text{at}, \alpha 2] \rangle \Rightarrow$

$\langle \alpha 4, \alpha 1, \sigma[\alpha 1:\text{tm}, \alpha 3][\alpha 2:\dots][\alpha 3:\dots] \rangle \Rightarrow \dots \Rightarrow$

$\langle \alpha 1, \alpha 4, \sigma[\alpha 1:\text{tm}, \alpha 3][\alpha 2:\dots][\alpha 3:\dots][\alpha 4:A(\text{at}), A(\text{at})] \rangle \Rightarrow$

$\langle \text{tm}, \alpha 1, \sigma[\alpha 1:\alpha 3, \alpha 4] \rangle$... clearly Q is not yet finished with its backtracking: it still has to track back to $\alpha 3$, and then to $\alpha 2$ in order to visit $\alpha 4$.

I.e., Q should not terminate when hitting tm, but go on backtracking. Eventually a moment will occur in the computation after Q has hit tm repeatedly, when the full graph is restored and everywhere visited. ... But Q will not know this, since it cannot predict how often tm has to be passed prior to stopping.

End of insert

As we shall see later on, a considerably more interesting observation concerning the termination of M can be made, which holds, a fortiori, for $\text{LEFT}(\alpha, \text{NIL}, \sigma)$.

Proof of the above assertion splits as usual into two cases:

\sqsupseteq : By induction on the recursiondepth of M . Omitted.

\sqsubseteq : By (simultaneous n -step- $n+1$) induction on the recursiondepths of LEFT and BACK .

We prove the latter. To this end two auxiliary lemma's, and an important observation are needed.

LEMMA: $M(\alpha, \sigma[\beta:1, a_1, a_2, a_3])[\beta:1, b_1, b_2, b_3] = M(\alpha, \sigma[\beta:1, b_1, b_2, b_3])$
with a_i, b_i of the correct types ... (A)

Proof: By straightforward induction on recursiondepth M .

LEMMA: $\text{hd}(\gamma, M(\delta, \rho)) \sqsubseteq \text{hd}(\gamma, \rho)$
 $\text{tl}(\gamma, M(\delta, \rho)) \sqsubseteq \text{tl}(\gamma, \rho) \quad \dots$ (B)
 $f(\gamma, M(\delta, \rho)) \sqsubseteq f(\gamma, \rho)$
 $\text{already visited}(\gamma, M(\delta, \rho[\gamma:1, \dots])) \sqsubseteq 1$

Proof: By straightforward induction on recursiondepth M .

OBSERVATION: The latter lemma will be used in conjunction with the call-by-value parameter mechanism. Let P denote a procedure with parameters called-by-value. Then $P(\dots, \text{hd}(\gamma, M(\delta, \rho)), \dots, M(\delta, \rho), \dots) = P(\dots, \text{hd}(\gamma, \rho), \dots, M(\delta, \rho), \dots)$, and similarly for $\text{tl}, f, \text{already visited} \dots$ (C)

Insert

Observation (C) can be more succinctly formulated by:

Suppose we are considering functions on flat lattices, to flat lattices, only. And let $g(\alpha, f(x, y)) \sqsubseteq g(\alpha, y)$.

If $h(x_1, x_2)$ is strict in its arguments, and we know that $h(\alpha, f(x, y)) = h(g(\alpha, f(x, y)), f(x, y))$ then also $h(\alpha, f(x, y)) = h(g(\alpha, y), f(x, y))$.

end of insert

This has the following reason:

In case $M(\delta, \rho)$ does not terminate, $P(\dots, \text{hd}(\gamma, M(\delta, \rho)), \dots, M(\delta, \rho)) = \perp$ anyhow, and otherwise, $\text{hd}(\gamma, M(\delta, \rho)) = \text{hd}(\gamma, \rho)$, and hence, $(P(\dots, \text{hd}(\gamma, M(\delta, \rho)), \dots, M(\delta, \rho), \dots) = P(\dots, \text{hd}(\gamma, \rho), \dots, M(\delta, \rho), \dots))$.

An axiomatic theory on which this observation can be based is developed in my thesis (Recursive procedures: semantics & prooftheory, Mathematisch Centrum 1975) and is closely linked with predicate transformers.

PROOF: \sqsubseteq part:

(i) $\neg \text{at}(\alpha) \wedge \neg \text{already visited}(\alpha, \sigma) = 1$:

$\text{LEFT}^{n+1}(\alpha, \beta, \sigma) = \text{LEFT}^n(\text{hd}(\alpha, \sigma), \alpha, \sigma[\alpha:1, 1, \text{tl}(\alpha, \sigma), \beta]) \sqsubseteq$ (hypothesis)

$\text{BACK}^n(\dots) \sqsubseteq \text{BACK}^{n+1}(\text{hd}(\alpha, \sigma), \alpha, M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, 1, \text{tl}(\alpha, \sigma), \beta])) = (B)$

$\text{LEFT}^n(\text{tl}(\alpha, \sigma), \alpha, M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, 1, \text{tl}(\alpha, \sigma), \beta])[\alpha:1, 0, \beta, \text{hd}(\alpha, \sigma)]) \sqsubseteq$ (hyp.)

$\text{BACK}^n(\dots) \sqsubseteq \text{BACK}^{n+1}(\text{tl}(\alpha, \sigma), \alpha, M(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, 1, \text{tl}(\alpha, \sigma), \beta]))$

$[\alpha:1, 0, \beta, \text{hd}(\alpha, \sigma)]) = (B)$

$\text{BACK}^n(\alpha, \beta, M(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, 1, \text{tl}(\alpha, \sigma), \beta])[\alpha:1, 0, \dots]))$

$[\alpha:1, 0, \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)] \sqsubseteq$

{By (A): $M(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, 1, \text{tl}(\alpha, \sigma), \beta])[\alpha:1, 0, \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)]) =$

(by (A), again) $M(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, 0, \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)])) = M(\alpha, \sigma)$ }

$\text{BACK}^{n+1}(\alpha, \beta, M(\alpha, \sigma)).$

(ii) $\text{at}(\alpha) \vee \text{already visited}(\alpha, \sigma) = 1$:

$\text{LEFT}^{n+1}(\alpha, \beta, \sigma) = \text{BACK}^{n+1}(\alpha, \beta, \sigma) = \text{BACK}^{n+1}(\alpha, \beta, M(\alpha, \sigma))$

\square

9. TERMINATION OF $M(\alpha, \sigma)$ - and, hence, of the DEUTSCH-SCHORR-WAITE marking algorithm.

The binary digraphs which we are manipulating are not arbitrary at all: they are finite! Moreover, since σ has been introduced as a partial mapping, σ has to satisfy certain consistency criteria in that, in case $\langle \alpha, \sigma \rangle$ represents a binary digraph and $\neg \text{at}(\alpha)$, one has $\alpha \in \text{domain}(\sigma)$, and hence $\text{already visited}(\alpha, \sigma)$ is well-defined so that the test $\text{at}(\alpha) \vee \text{already visited}(\alpha, \sigma) = 1$ is also well-defined and either true or false. In case the latter is false, since $\alpha \in \text{domain}(\sigma)$, also $\text{hd}(\alpha, \sigma)$ and $\text{tl}(\alpha, \sigma)$ are well-defined, and both $\text{hd}(\alpha, \sigma)$ and $\text{tl}(\alpha, \sigma)$ are either atoms or belong to $\text{domain}(\sigma)$. In case they belong both to $\text{domain}(\sigma)$, therefore, $\langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle$ and $\langle \text{tl}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle$ represent again binary digraphs to which the same reasoning applies, until, after repeated execution of the hd - and/or tl -operations, one arrives at locations α^∇ s.t. $\text{at}(\alpha^\nabla) \vee \text{already visited}(\alpha^\nabla, \sigma^\nabla) = 1$ does hold.

The formulation of this property requires a more sophisticated induction principle than the principles previously used to characterize binary trees and linear lists (cfr. pages 8, 9, 12), primarily because one has

to assert both that $\alpha \in \text{dom}(\sigma)$ - this is a mathematical, and in general not programmable statement since it amounts in its full generality to solving the halting problem - and the essential observation of FINITENESS mentioned above - which we shall formulate in terms of well-foundedness (for the case of markable binary digraphs).

To clarify the situation, let us return to the axiomatic characterization of binary trees discussed previously. We did prove that $P(x) \sqsubseteq x$ where

$$P(x) \Leftarrow \text{if } \text{at}(x) \text{ then } x \text{ else } \text{cons}(P(\text{car}(x)), P(\text{cdr}(x))) \text{ fi}$$

using n-step-n+1 induction.

But termination of $P(x)$ had to be asserted, since $P(x) \sqsubseteq x$ and nontermination of $P(x)$ were comparable - just use infinite trees as model.

What did this assertion amount to? That, given a binary tree, one could apply car or cdr only a finite number of times "before hitting an atom" - i.e. computation of $P(x)$ can never be infinite -, and moreover, for the successively generated values x , $\text{car}(x)$ and $\text{cdr}(x)$ are always defined, until one hits an atom. Now infinite computation is ruled out by asserting that $\lambda x. \text{if } \neg \text{at}(x) \text{ then } \text{car}(x) \text{ or } \text{cdr}(x) \text{ else } \perp \text{ fi}$ is well-founded, where or denotes nondeterministic choice, i.e., there does not exist an infinite sequence $\langle x_i \rangle_{i=0}^{\infty}$ s.t. $x_{i+1} = \text{car}(x_i)$ or $x_{i+1} = \text{cdr}(x_i)$, and $x_j = \perp$ for some $j \in \mathbb{N}$. Undefinedness of car and cdr is ruled out by asserting that $\neg \text{at}(x)$ implies that car and cdr are defined in x .

The analogy with this situation for binary digraphs is that occurrence of locations α s.t. $(\neg \text{at}(\alpha) \wedge) \alpha \notin \text{dom}(\sigma)$ must be forbidden. This can be formulated in terms of well-foundedness by extending well-foundedness of $\lambda \alpha, \sigma. \text{if}$ already visited $(\alpha, \sigma) = 0$ then $\langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle$ or $\langle \text{tl}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle$ else ... (- comparable with well-foundedness of $\text{car}(x)$ or $\text{cdr}(x)$ -) by also requiring well-foundedness of $\lambda \alpha, \sigma. \text{if } \neg \text{at}(\alpha) \wedge \alpha \notin \text{dom}(\sigma) \text{ then } \langle \alpha, \sigma \rangle$ else ... since this extension enforces ill-foundedness on all $\langle \alpha, \sigma \rangle$ s.t. $\neg \text{at}(\alpha) \wedge \alpha \notin \text{dom}(\sigma)$.

Notation: If relation R is well-founded in x , this will be abbreviated by $x \in \text{wf}(R)$. Also already visited will be abbreviated to av , and $\text{domain}(\sigma)$ to $\text{dom}(\sigma)$ - as already done above.

Let I be defined by

$$I = \lambda \alpha, \sigma. \text{if } \alpha \in \text{dom}(\sigma) \wedge \neg \text{at}(\alpha) \text{ then } \langle \alpha, \sigma \rangle \text{ else} \\ \text{DEF } \text{if } \text{av}(\alpha, \sigma) = 0 \text{ then } \langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, f(\alpha, \sigma), \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)] \rangle \text{ or} \\ \langle \text{tl}(\alpha, \sigma), \sigma[\alpha:1, f(\alpha, \sigma), \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)] \rangle \text{ else } \perp \text{ fi.}$$

Then our induction principle for binary digraphs with two (at least one) marking bit is:

J: $\alpha \in \text{dom}(\sigma) \Rightarrow \langle \alpha, \sigma \rangle \in \perp(I)$

We assume that if α denotes an atom, $I(\alpha, \sigma) = \perp$ since $\text{av}(\alpha, \sigma) = \perp$ in that case. Notice also that well-foundedness is defined in such a way as to exclude sequences $\langle x_i \rangle_{i=0}^{\infty}$ in which \perp occurs: well-foundedness of I in $\langle \alpha, \sigma \rangle$ implies the absence of an infinite sequence $\langle x_i \rangle_{i=0}^{\infty}$ s.t. $x_i \neq \perp$, and $I(x_i) = x_{i+1}$, for $i \in \mathbb{N}$.

Amongst others, J asserts no infinite looping of cycles (on account of marking of, and testing on, the av -bit), but does model repeated traversal of shared subgraphs, if such sharing occurs. {The emphasis is here on modelling: the occurrences of $\sigma[\alpha:1, \dots]$ in I , i.e. in $\langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle$ or $\langle \text{tl}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle$ concern separate copies of the same memory. I.e., since no "side-effect" of one copy on another one is expressible, shared data, which occur necessarily in as many separate copies of the memory (as the degree of sharing) are traversed in each of these copies in which they exist.}

Now in case of $M(\alpha, \sigma)$ the situation is different:

Repeated traversal of shared substructures is excluded by the recursive marking procedure $M(\alpha, \sigma)$ (declared by

$$M(\alpha, \sigma) \leftarrow \text{if } \text{at}(\alpha) \vee \text{av}(\alpha, \sigma) = \perp \text{ then } \sigma \text{ else } \\ M(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, f(\alpha, \sigma), \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)])) \text{ fi},$$

since the call-by-value parameter mechanism enforces that, recursively, first $\langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle$ is traversed and marked as already visited, and secondly $\langle \text{tl}(\alpha, \sigma), \sigma^\nabla \rangle$ is traversed with σ^∇ only differing from $\sigma[\alpha:1, \dots]$ in that previously visited nodes have been marked already, and are therefore excluded from traversal (, as is not the case with J).

{Thus, the order of the computations in the recursive case (originating in $M(\alpha, \sigma)$) and the iterative case (originating in $\text{LEFT}(\alpha, \text{NIL}, \sigma)$) are the same, but the repetition of execution of I starting in $\langle \alpha, \sigma \rangle$, imposed by well-foundedness of I in $\langle \alpha, \sigma \rangle$, leads in general not at all to the same computations as those originating from $M(\alpha, \sigma)$, and even, if they are the same (because there is no sharing) the respective orderings may be totally different.}

Consequently, infinite computation of $M(\alpha, \sigma)$ implies ill-foundedness of I in $\langle \alpha, \sigma \rangle$: The possibility of infinite traversal without (possibility for) repeated traversal of shared subgraphs - as is the case with $M(\alpha, \sigma)$ - implies trivially the possibility of infinite traversal with (possibility

for) finite, repeated, traversal of shared substructures.

Hence J implies the following by contraposition of the above observation:

THEOREM: $\alpha \in \text{dom}(\sigma) \Rightarrow \langle \alpha, \sigma \rangle \in \text{domain } M$

Intuitively, one is faced with the following situation:

What does it mean that $M(\alpha, \sigma)$ has no well-defined value?

It means that the computation sequence for $M(\alpha, \sigma)$

either (1) leads to an infinite number of inner recursive calls of M ,

or (2) after a finite number (possibly none) of inner recursive calls

of M an elementary statement of M is undefined (in the inter-

mediate state presented as input to this elementary statement).

What does case(1) imply?:

It implies that there exists an infinite sequence $\langle x_i \rangle_{i=0}^{\infty}$, with $x_0 = \langle \alpha, \sigma \rangle$, $x_i \neq \perp$, s.t. the state-transformations mapping x_i to x_{i+1} describe state-transformations in between successive inner calls of M .

What are the state-transformations in between successive inner calls of M ?

They are described by

$\lambda \alpha, \sigma. \text{ if } \text{av}(\alpha, \sigma) = 0 \text{ then}$
 $\langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle \text{ or } \langle \text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) \rangle \text{ else } \perp$

- since there are two possibilities for inner recursive calls of M we need to express the (nondeterministic) choice between these two possibilities:

$\langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle \text{ or } \langle \text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) \rangle$.

What does case(2) imply?

After a finite number (possible none) of inner recursive calls of M one faces the situation that for some x_i , the next-to-be-computed-elementary statement of M is undefined in x_i . Let $x_i = \langle \alpha_i, \sigma_i \rangle$. Then inspection of M 's procedure body reveals that this situation can only arise in case $\alpha_i \notin \text{dom}(\sigma_i) \wedge \neg \text{at}(\alpha_i)$, i.e., α_i is a location in which σ_i is not defined.

Reduction of case(2) to case(1):

As remarked above, case(2) can be reduced to case(1) by introducing the following relation as an extra possibility for transformation between successive x_i 's:

$\lambda\alpha, \sigma. \underline{\text{if } \alpha \notin \text{dom}(\sigma) \wedge \neg \text{at}(\alpha) \text{ then } \langle \alpha, \sigma \rangle \text{ else } \dots}$,

since, once $\alpha \notin \text{dom}(\sigma) \wedge \neg \text{at}(\alpha)$ is satisfied in $\langle \alpha, \sigma \rangle$, it remains satisfied under this transformation ... ad infinitum.

Thus $M(\alpha, \sigma)$ does not terminate iff there exists an infinite sequence

$\langle x_i \rangle_{i=0}^{\infty}, x_0 = \langle \alpha, \sigma \rangle$, s.t. for $i \in \mathbb{N}, x_i \neq \perp, x_i R x_{i+1}$ with R defined by

$\lambda\alpha, \sigma. \underline{\text{if } \alpha \notin \text{dom}(\sigma) \wedge \neg \text{at}(\alpha) \text{ then } \langle \alpha, \sigma \rangle \text{ else}}$
 $\quad \underline{\text{if } \text{av}(\alpha, \sigma) = 0 \text{ then}}$
 $\quad \langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots] \rangle \text{ or } \langle \text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) \rangle \text{ else } \perp \text{ fi.}$

(These observations relate to the paper by Hitchcock & Park; personally I think they are a more convincing example for application of their theory than these authors supply themselves.)

Next we prove that the existence of such an infinite sequence for R which originates in $\langle \alpha, \sigma \rangle$ implies the existence of such an infinite sequence for I . Hence, well-foundedness of I in $\langle \alpha, \sigma \rangle$ implies well-foundedness of R in $\langle \alpha, \sigma \rangle$, i.e., termination of M in $\langle \alpha, \sigma \rangle$.

We shall prove the following:

Let $\langle \langle \alpha_i, \sigma_i \rangle \rangle_{i=0}^{\infty}$ satisfy $\langle \alpha_i, \sigma_i \rangle R \langle \alpha_{i+1}, \sigma_{i+1} \rangle, \langle \alpha_i, \sigma_i \rangle \neq \perp$.

Then we can replace the memories σ_i in this sequence by other memories

τ_i s.t. $\langle \alpha_i, \tau_i \rangle I \langle \alpha_{i+1}, \tau_{i+1} \rangle, \langle \alpha_i, \tau_i \rangle \neq \perp, \sigma_0 = \tau_0$; τ_i will be obtained from σ_i by "deleting the side-effect on the memory σ_i due to executing $M(\text{hd}(\alpha_i, \sigma_i), \sigma_i[\alpha_i:1, \dots])$ ".

I.e., assuming inductively that well-defined pairs $\langle \alpha_0, \tau_0 \rangle, \dots, \langle \alpha_n, \tau_n \rangle$ have been obtained s.t. $\sigma_0 = \tau_0$,

$\langle \alpha_i, \tau_i \rangle I \langle \alpha_{i+1}, \tau_{i+1} \rangle, i = 1, \dots, n-1$, then a well-defined pair

$\langle \alpha_{n+1}, \tau_{n+1} \rangle$ s.t. $\langle \alpha_n, \tau_n \rangle I \langle \alpha_{n+1}, \tau_{n+1} \rangle$ is obtained using the following inductively defined construction process:

(1) Either: If

• $\langle \alpha_0, \sigma_0 \rangle$
 ⋮
 • $\langle \alpha_n, \sigma_n \rangle$ s.t. $\alpha_n \notin \text{dom}(\sigma_n) \wedge \neg \text{at}(\alpha_n)$, and hence
 $\langle \alpha_{n+1}, \sigma_{n+1} \rangle = \langle \alpha_n, \sigma_n \rangle$,
 then
 • $\langle \alpha_0, \tau_0 \rangle (= \langle \alpha_0, \sigma_0 \rangle)$
 ⋮
 • $\langle \alpha_n, \tau_n \rangle$ s.t. $\alpha_n \notin \text{dom}(\tau_n) \wedge \neg \text{at}(\alpha_n)$, and chose
 and choose $\langle \alpha_n, \tau_n \rangle$ itself for $\langle \alpha_{n+1}, \tau_{n+1} \rangle$.

(2) Or: If

- $\langle \alpha_0, \sigma_0 \rangle$
- $\langle \alpha_n, \sigma_n \rangle$ s.t. $av(\alpha_n, \sigma_n) = 0$ and
- $\langle \alpha_{n+1}, \sigma_{n+1} \rangle = \langle hd(\alpha_n, \sigma_n), \sigma_n[\alpha_n:1, \dots] \rangle$

then

- $\langle \alpha_0, \tau_0 \rangle (= \langle \alpha_0, \sigma_0 \rangle)$
- $\langle \alpha_n, \tau_n \rangle$ s.t. $av(\alpha_n, \tau_n) = 0$, and chose
- $\langle hd(\alpha_n, \tau_n), \tau_n[\alpha_n:1, \dots] \rangle$ for $\langle \alpha_{n+1}, \tau_{n+1} \rangle$

(3) Or: THE INTERESTING CASE:

if

- $\langle \alpha_0, \sigma_0 \rangle$
- $\langle \alpha_n, \sigma_n \rangle$ s.t. $av(\alpha_n, \sigma_n) = 0$ and

$hd(\alpha_n, \sigma_n)$:

- $\langle \alpha_{n+1}, \sigma_{n+1} \rangle = \langle tl(\alpha_n, \sigma_n), M(hd(\alpha_n, \sigma_n), \sigma_n[\alpha_n:1, \dots]) \rangle$

then

- $\langle \alpha_0, \tau_0 \rangle (= \langle \alpha_0, \sigma_0 \rangle)$
- $\langle \alpha_n, \tau_n \rangle$ s.t. $av(\alpha_n, \tau_n) = 0$ and
- $\langle tl(\alpha_n, \tau_n), \tau_n[\alpha_n:1, \dots] \rangle = \langle \alpha_{n+1}, \tau_{n+1} \rangle$

notice that the side effect
due to •
has been deleted.

Only the third case is note-worthy:

At the LHS of the picture we have suggested the side-effect on $\sigma_n[\alpha_n:1, \dots]$ due to execution of $M(hd(\alpha_n, \sigma_n), \sigma_n[\alpha_n:1, \dots])$ instead of just having to deal with $\sigma_n[\alpha_n:1, \dots]$.

This side-effect is absent in our definition of τ_{n+1} .

Intuitively these side-effects do not matter, since existence of the path $\langle \langle \alpha_i, \sigma_i \rangle \rangle_{i=0}^\infty$ implies that α_i is never already visited, i.e. $av(\alpha_i, \sigma_i) = 0$, along that path. Thus we might just as well delete the marking of other parts of the memory by $M(hd(\alpha_i, \sigma_i), \sigma_i[\alpha_i:1, \dots])$, since this did not result in marking $\alpha_{i+1}, \alpha_{i+2}, \dots$.

Next we formalize this process, and prove that for τ_i so-obtained,

$\langle \alpha_i, \tau_i \rangle \vdash \langle \alpha_{i+1}, \tau_{i+1} \rangle$, indeed.

Insert

In the proof presented on the following pages we prove, assuming existence of one infinite sequence, the existence of another infinite sequence. Hence it is not clear how to express this proof using n-step-n+1 induction only. This connection between proofs using n-step-n+1 induction and infinite sequences is elucidated in my paper "on backtracking and greatest fixed-points" (ICALP '77, Turku, Arto Salomaa (ed.), Springer Lecture Notes in Computer Science); hence the proof can be considered as one using n-step-n+1 induction, only.

end of insert

Let $p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle)$ be defined by

$$\begin{aligned} \forall \gamma \in \text{Loc. } [& (\gamma \in \text{dom}(\sigma) \leftrightarrow \gamma \in \text{dom}(\tau)) \wedge \\ & (\text{av}(\gamma, \sigma) = 0 \rightarrow \text{av}(\gamma, \tau) = 0) \wedge \\ & f(\gamma, \sigma) = f(\gamma, \tau) \wedge \text{hd}(\gamma, \sigma) = \text{hd}(\gamma, \tau) \wedge \text{tl}(\gamma, \sigma) = \text{tl}(\gamma, \tau)]. \end{aligned}$$

Then we have:

$$p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle) \Rightarrow p(\langle \alpha', \sigma' \rangle, \langle \alpha', \tau' \rangle) ,$$

Where α', σ' is such that

- a) if $\alpha \notin \text{dom}(\sigma) \wedge \neg \text{at}(\alpha)$ then $\langle \alpha', \sigma' \rangle = \langle \alpha, \sigma \rangle$,
- b) if $\alpha \in \text{dom}(\sigma)$ then
 - either (i) $\langle \alpha', \sigma' \rangle = \langle \text{hd}(\alpha, \sigma), \sigma[\alpha:1, f(\alpha, \sigma), \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)] \rangle$
 - or (ii) $\langle \alpha', \sigma' \rangle = \langle \text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, f(\alpha, \sigma), \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)]) \rangle$
in case $M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots])$ is well-defined,*)

and τ' is obtained from τ as follows:

in case(a) above, $\tau' = \tau$,

and in both subcases of (b) above, $\tau' = \tau[\alpha:1, f(\alpha, \sigma), \text{hd}(\alpha, \sigma), \text{tl}(\alpha, \sigma)]$

*) This can be safely assumed, since, otherwise, one would have chosen $\alpha' = \text{hd}(\alpha, \sigma)$, i.e. case(b(i)) in order to prolong the sequence of α_i 's towards infinity.

PROOF:

Case a): Since $\langle \alpha, \sigma \rangle = \langle \alpha', \sigma' \rangle$ and $\tau = \tau'$, trivially $p(\langle \alpha', \sigma' \rangle, \langle \alpha', \tau' \rangle) \equiv p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle)$.

Case b) (i): $\gamma \in \text{dom}(\sigma') \leftrightarrow \gamma \in \text{dom}(\tau')$:

$\text{dom}(\tau[\alpha:1, \dots]) = \{\text{since } \alpha \in \text{dom}(\sigma) \text{ one has } \alpha \in \text{dom}(\tau) \text{ by } p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle)\}$
 $\text{dom}(\tau) = \{\text{by } p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle)\} \text{dom}(\sigma) = \text{dom}(\sigma')$.

$\text{av}(\gamma, \sigma') = 0 \rightarrow \text{av}(\gamma, \tau') = 0$.

Assume $\text{av}(\gamma, \sigma') = 0$:

(i) $\gamma = \alpha$ leads to contradiction since $\text{av}(\alpha, \sigma') = 1$. Hence $\gamma \neq \alpha$:

(ii) $\gamma \neq \alpha$: then $\text{av}(\gamma, \tau') = 1$ leads to contradiction since

$1 = \text{av}(\gamma, \tau') = \text{av}(\gamma, \tau) = (\text{by } p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle) \text{ and contraposition})$
 $\text{av}(\gamma, \sigma) = \text{av}(\gamma, \sigma') = 0 \text{ †.}$

Hence $\text{av}(\gamma, \tau') = 0$.

$f(\gamma, \sigma') = f(\gamma, \tau) \wedge \text{hd}(\gamma, \sigma') = \text{hd}(\gamma, \tau') \wedge \text{tl}(\gamma, \sigma') = \text{tl}(\gamma, \tau')$: Obvious.

Case b) (ii): In addition to properties (B) and (C) on page , one needs the following auxiliary lemma:

$(\langle \delta, \rho \rangle \in \text{dom}(M) \Rightarrow \forall \gamma \in \text{Loc}, \tau \in \text{Mem}. [M(\delta, \rho) = \langle \delta, \tau \rangle \Rightarrow \text{dom}(\delta) = \text{dom}(\tau)])$,

the proof of which is an exercise in induction on recursion depth.

Assume $M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots])$ is well-defined (cfr. the note on the bottom of page 46):

$\gamma \in \text{dom}(\sigma') \leftrightarrow \gamma \in \text{dom}(\tau')$

Since $M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots])$ is well-defined, one has by the auxiliary lemma:

$\gamma \in \text{dom}(\sigma') \leftrightarrow \gamma \in \text{dom}(\sigma[\alpha:1, \dots])$ and $\text{dom}(\sigma[\alpha:1, \dots]) = \text{dom}(\sigma) =$
 $(\text{by } p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle)) \text{dom}(\tau) = \text{dom}(\tau')$

$\text{av}(\gamma, \sigma') = 0 \rightarrow \text{av}(\gamma, \tau') = 0$

Assume $\text{av}(\gamma, \sigma') = 0$.

(i) $\gamma = \alpha$ leads to contradiction since $\text{av}(\alpha, \sigma') =$ (by properties B and C on page) 1. †. Hence $\gamma \neq \alpha$:

(ii) $\gamma \neq \alpha$: Then $\text{av}(\gamma, \tau') = 1$ leads to contradiction:

$1 = \text{av}(\gamma, \tau') = \text{av}(\gamma, \tau) = (\text{by } p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle) \text{ and contraposition})$

$\text{av}(\gamma, \sigma) = \text{av}(\gamma, \sigma') = 0 \text{ †. Hence } \text{av}(\gamma, \tau') = 0$

□

10. An exercise in Scott induction

1. In the proofs using n -step- $n+1$ induction discussed above, one notices that these proofs are uniform in n :

For in these proofs one only observes the case distinction between $n=0$ and $n \neq 0$; one is not interested whether $n=7,13,217$.

E.g., let the recursive procedure P be defined by

$$P(x_1, \dots, x_n) \leftarrow \tau[P](x_1, \dots, x_n),$$

and suppose one proved "something about" P using n -step- $n+1$ induction.

In case $n=0$, $P^{(0)} = \lambda x_1, \dots, x_n. \perp$, and hence one has to prove "something about" the nowhere-defined function Ω .

In case $n \neq 0$, "something about" $P^{(n)}$ has to imply "something about" $P^{(n+1)} = \tau[P^{(n)}]$.

Now Scott induction requires the introduction of partial function variables X in order to express the case $n \neq 0$ uniformly in n , i.e., without mentioning n at all:

One has to prove that an assertion about a free variable X implies the same assertion about $\tau[X]$ (, where substitution of X inside τ is defined in such a way that no free-bound clashes of occurrences of X inside τ can arise).

The other case, $n=0$, is covered by proving that this assertion also holds for the nowhere defined partial function Ω .

Obviously, once one has proved assertion A for Ω , and also that if A holds for (any value of the free variable) X , this implies A for $\tau[X]$, induction on i yields that A holds for $\tau^i[\Omega]$, $i \geq 0$, where $\tau^0[\Omega] \stackrel{\text{DEF}}{=} \Omega$, $\tau^{i+1}[\Omega] \stackrel{\text{DEF}}{=} \tau[\tau^i[\Omega]]$. Since $P = \lim_{i=0}^{\infty} \tau^i[\Omega]$, one therefore needs the property that A commutes with the limit of $\langle \tau^i[\Omega] \rangle_{i=0}^{\infty}$ in order to obtain that A holds for P ; this is by definition the case if A is continuous in X . Since continuity of A in X can often be guaranteed by simple syntactic restrictions on A , observing these syntactic restrictions justifies:

Scott's induction rule: (1) $\Phi \vdash A(\Omega)$

(2) $\Phi, A(X) \vdash A(\tau[X])$

$\Phi \vdash A(\mu X[\tau[X]]);$

here X denotes a free variable;

Φ a list of assumptions in which X does not occur free;

substitution inside τ is suitably defined as to evade clashes between bound

occurrences of a variable inside τ , and substituted free occurrences of that variable;

$\mu X[\tau[X]]$ denotes the least fixed point of the transformation $X \mapsto \tau[X]$, occurrences of X inside $\mu X[\tau[X]]$ are bound, and $\tau[X]$ is continuous in X (in order to that $\mu X[\tau[X]] = \lim \tau^i[\Omega]$); specifying the form of Φ, A would entail formalizing a language: since we do not intend to do so, we refer for examples to previous proofs.

This rule can be extended to mutually recursive procedures as:

Simultaneous Scott induction:

$$\begin{array}{l} (1) \quad \Phi \vdash A(\Omega_1, \dots, \Omega_n) \\ (2) \quad \Phi, A(X_1, \dots, X_n) \vdash A(\tau_1[X_1, \dots, X_n], \dots, \tau_n[X_1, \dots, X_n]) \\ \hline \Phi \vdash A(\mu_1 X_1 \dots X_n [\tau_1, \dots, \tau_n], \dots, \mu_n X_1 \dots X_n [\tau_1, \dots, \tau_n]); \end{array}$$

here the stipulations mentioned above should be generalized;

Ω_i and X_i are of the same, appropriate, type;

$\mu_i X_1 \dots X_n [\tau_1[X_1, \dots, X_n], \dots, \tau_n[X_1, \dots, X_n]]$ denotes the i^{th} component of the least fixed point of the transformation

$$\langle X_1, \dots, X_n \rangle \mapsto \langle \tau_1[X_1, \dots, X_n], \dots, \tau_n[X_1, \dots, X_n] \rangle, \quad i:1 \dots n.$$

In case P_1, \dots, P_n are mutually recursive, and declared by

$$\left. \begin{array}{l} P_1 \leftarrow \tau_1[P_1, \dots, P_n] \\ \vdots \\ P_n \leftarrow \tau_n[P_1, \dots, P_n] \end{array} \right\} ,$$

$\mu_i X_1 \dots X_n [\tau_1, \dots, \tau_n]$ expresses the I/O behaviour of P_j .

The simultaneous version of Scott induction can be justified by defining $\tau_i^0[\Omega_1, \dots, \Omega_n] = \Omega_i, \tau_i^{k+1}[\Omega_1, \dots, \Omega_n] = \tau_i[\tau_1^k[\Omega_1, \dots, \Omega_n], \dots, \tau_n^k[\Omega_1, \dots, \Omega_n]]$, and observing that $P_i = \lim_{k \rightarrow \infty} \tau_i^k$.

2. Our intention is to formulate the termination proof of M using Scott's induction rule.

This leads to the difficulty that the notion of well-foundedness cannot in general be appropriately defined within the framework of continuous partial functions; in fact, well-foundedness requires in general the introduction of certain non-continuous, monotone, transformations, cfr. [Hitchcock & Park]. The non-continuous aspect of these transformations arises from the fact that

a relation S , of which one intends to express well-foundedness, may be unbounded in its nondeterminism, i.e., for certain x , $\{y \mid \langle x, h \rangle \in S\}$ is infinite.

However, if this is not the case, i.e., S is finitely bounded in its non-determinism, then one can prove that well-foundedness of S can be expressed using continuous functions.

It can be proved, cfr. [de Bakker], that if such is the case, one may just as well express well-foundedness of S by requiring termination of a certain "boolean procedure" whose declaration depends upon the definition of S .

Specifically, the paper by de Bakker describes a general technique which implies, e.g., that M terminates in $\langle \alpha, \sigma \rangle$ iff the boolean procedure p_R , defined below, evaluates to true in $\langle \alpha, \sigma \rangle$.

Observe that the relations I and R defined previously are bounded in their non-determinism as only a choice between two possibilities is introduced.

Example:

p_I is defined by:

$$p_I(\alpha, \sigma) \leftarrow \begin{array}{l} \text{if } \alpha \notin \text{dom}(\sigma) \wedge \neg \text{at}(\alpha) \text{ then } p_I(\alpha, \sigma) \\ \text{else if } \text{at}(\alpha) \vee \text{av}(\alpha, \sigma) = 1 \text{ then } \text{true} \\ \text{else } p_I(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) \wedge p_I(\text{tl}(\alpha, \sigma), \sigma[\alpha:1, \dots]) \text{ fi.} \end{array}$$

Similarly, p_R is defined by:

$$p_R(\alpha, \sigma) \leftarrow \begin{array}{l} \text{if } \alpha \notin \text{dom}(\sigma) \wedge \neg \text{at}(\alpha) \text{ then } p_R(\alpha, \sigma) \\ \text{else if } \text{at}(\alpha) \vee \text{av}(\alpha, \sigma) = 1 \text{ then } \text{true} \\ \text{else } p_R(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) \wedge p_R(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots])) \text{ fi.} \end{array}$$

Now our induction principle J can be reformulated as

$$\boxed{p_I(\alpha, \sigma) \neq \perp}$$

And our theorem that J implies termination of M (in view of the remarks about de Bakker's paper) by

$$\boxed{p_R(\alpha, \sigma) = \perp \Rightarrow p_I(\alpha, \sigma) = \perp}$$

3. We shall prove by Scott induction that

$$p_R(\alpha, \sigma) = \perp \Rightarrow p_I(\alpha, \sigma) = \perp, \dots \quad (*)$$

As usual this assertion cannot be proved directly, since the inductive nature of constructing the infinite computation sequence for p_I requires a more general assertion.

The essential insight is here to prove

$$\forall \alpha, \sigma, \tau. ((p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle) \wedge p_R(\alpha, \sigma) = \perp) \Rightarrow p_I(\alpha, \tau) = \perp), \dots \quad (**)$$

with p defined as on page 53; (**) makes the inductive nature explicit of the termination argument given in that section; assertion (*) follows from (**), since $p(\langle \alpha, \sigma \rangle, \langle \alpha, \sigma \rangle)$ holds, by taking $\sigma = \tau$. (Finding such a more general assertion as (**) requires an effort which sums up to at least a month of work.)

PROOF of (**): By Scott induction on p_I , i.e., $\mu X_I[\tau[X_I]]$, where τ_I denotes the procedure body of p_I .

Hence we have to prove:

- (i) $\forall \alpha, \sigma, \tau. [p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle) \wedge p_R \langle \alpha, \sigma \rangle = \perp \Rightarrow \Omega \langle \alpha, \tau \rangle = \perp]$: trivial.
- (ii) $\forall \alpha, \sigma, \tau. [p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle) \wedge p_R \langle \alpha, \sigma \rangle = \perp \Rightarrow X_I \langle \alpha, \tau \rangle = \perp] \vdash$
 $\forall \alpha, \sigma, \tau. [p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle) \wedge p_R \langle \alpha, \sigma \rangle = \perp \Rightarrow \tau_I[X_I] \langle \alpha, \tau \rangle = \perp].$

(After having proved (i) & (ii), Scott induction implies (**) by the least fixed point characterization of recursive procedures.)

PROOF of (ii):

Assume the inductive assumption, assume $p(\langle \alpha, \sigma \rangle, \langle \alpha, \tau \rangle)$ and assume $p_R(\alpha, \sigma) = \perp$. Then one has to prove $\tau_I[X_I](\alpha, \tau) = \perp$.

There are three cases:

- $\text{at}(\alpha) \wedge \alpha \notin \text{dom}(\sigma)$: then $\text{at}(\alpha) \wedge \alpha \in \text{dom}(\tau)$ follows from p , by assumption $X_I \langle \alpha, \tau \rangle$ and $\tau_I[X_I](\alpha, \tau) = X_I(\alpha, \sigma)$, hence the result follows from the inductive assumption.
- $\text{at}(\alpha) \vee \text{av}(\alpha, \sigma) = 1$: then $p_R(\alpha, \sigma) = \text{true}$, which invalidates the antecedent of the implication, and hence the implication is trivially satisfied.
- $\alpha \in \text{dom}(\sigma)$: $p_R(\alpha, \sigma) = p_R(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) \wedge p_R(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]))$, and $\tau_I[X_I](\alpha, \tau) = X_I(\text{hd}(\alpha, \tau), \tau[\alpha:1, \dots]) \wedge X_I(\text{tl}(\alpha, \tau), \tau[\alpha:1, \dots])$.

$$(i) \quad p_R(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) = \perp.$$

Chose $\alpha' = \text{hd}(\alpha, \sigma)$, $\sigma' = \sigma[\alpha:1, \dots]$, $\tau' = \tau[\alpha:1, \dots]$.

It follows from p that $\alpha' = \text{hd}(\alpha, \tau)$.

By the result on page one has $p(\langle \alpha', \sigma' \rangle, \langle \alpha', \tau' \rangle)$.

We already assumed $p_R(\alpha', \sigma') = \perp$.

Therefore it follows from the inductive assumption that $X_I(\alpha', \tau') = \perp$.

Thus the result follows since $X_I(\alpha', \tau') = \perp$ implies $\tau_I[X_I](\alpha, \tau) = \perp$.

(ii) $p_R(\text{tl}(\alpha, \sigma), M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots])) = \perp$.

One can safely assume $M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots])$ to be well-defined, for, if not, $p_R(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots]) = \perp$ and the previous case applies.

Chose $\alpha' = \text{tl}(\alpha, \sigma)$, $\sigma' = M(\text{hd}(\alpha, \sigma), \sigma[\alpha:1, \dots])$, $\tau' = \tau[\alpha:1, \dots]$.

It follows from p that $\alpha' = \text{tl}(\alpha, \tau)$.

by the result on page $p(\langle \alpha', \sigma' \rangle, \langle \alpha', \tau' \rangle)$ holds. We already assumed

$p_R(\alpha', \sigma') = \perp$. Therefore it follows from the inductive assumption that

$X_I(\alpha', \tau') = \perp$.

Thus the result follows since $X_I(\alpha', \tau') = \perp$ implies $\tau_I[X_I](\alpha, \tau) = \perp$.

□

This completes our correctness-proof of DEUTSCH-SCHORR-WAITE with a termination proof.

Note that we separated the problem of proving

$$\text{LEFT}(\alpha, \beta, \sigma) = \text{BACK}(\alpha, \beta, M(\alpha, \sigma))$$

by n-step-n+1 induction

from the problem of proving termination of $\text{LEFT}(\alpha, \beta, \sigma)$ by assuming the simplest induction principle available. This separation of proofs and techniques agrees with FLOYD's observations concerning the separation of a (total) correctness-proof into a partial correctness-proof (not involving the assertion of termination) and a termination proof. We stuck to this principle throughout these notes.

Well, dear reader, where from here? A correctness proof of Clark's list-copying algorithm ("Copying list structures without auxiliary storage", Dept. of Comp. Sc., C.M.U., oct. '75) is in preparation. Au revoir!

References

- de Bakker, Termination of nondeterministic programs, in 3rd colloquium on Automata, languages & programming, Michaelson & Milner (eds.), Edinburgh University Press, 1976.
- Burstall, Proving properties of programs by structural induction, Comput. J. 12, 41-48, 1969.
- Burstall, Program proving as hand simulation with a little induction, IFIP 74, North-Holland.
- Burstall & Darlington, A transformation system for developing recursive programs, J.ACM 24 (Jan. 1977), pp. 44-67.
- Clark, D.W., A fast algorithm for copying list structures, C.ACM 21 (May '78), pp. 351-357.
- Dwyer, Simple algorithms for traversing a tree without auxiliary stack, Inf. Proc. Lett. 2, 143-145, 1973.
- Dijkstra, Edsger W., A personal summary of the Gries-Owicki theory, EWD 559, Burroughs, Nuenen, the Netherlands, 1976.
- Dijkstra, E.W., et al., On the fly garbage collection - an exercise in cooperation, in Springer Lecture Notes on Computer Science No. 46, Springer Verlag (Berlin etc.), 1976.
- Fisher, D.A., Copying cyclic list structures in linear time using bounded workspace, C.ACM 18 (May '75), pp. 251-252.
- Floyd, Non-deterministic algorithms, J.ACM 14, 4, 1967.
- Floyd, Assigning meanings to programs, in: proc. of a symp. in appl. math., vol. 19, Mathematical aspects of Computer Science, J.P. Schwartz (ed.), AMS, Providence, R.I., 1967.
- Friedman & Wise, Cons should not evaluate its arguments, in Automata, Languages and Programming, Michaelson & Milner (eds.), Edinb. Univ. Press, Edinburgh, 1976.
- Gries, An exercise in proving programs correct, C.ACM 20, pp. 921-930, 1977.
- Gerhart, Correctness-preserving program transformations, Proc. 2nd POPL Symp. Palo Alto (1975).
- Gerhart, Proof theory and partial correctness of verification systems, SIAM J. Comp. 5 (Sept. 1976), pp. 355-377.

- Henderson & Morris, A lazy evaluator, Proc. 3rd POPL Symp. Atlanta, Georgia, 1976.
- Hitchcock & Park, Induction rules and proofs of termination, in Proc. IRIA symp. on automata, formal languages, and programming, M. Nivat (ed.), North-Holland, 1972.
- Hoare, An axiomatic basis for computer programming, C.ACM 12, 576-583, 1969.
- Lee, de Roever & Gerhart, The evolution of list-copying algorithms, Proc. 6th POPL Symp., San Antonio, 1979.
- Manna, Ness & Vuillemin, Inductive methods for proving properties of programs, in: Proc. of an ACM Conference on proving assertions about programs, Las Cruces, 1972.
- McCarthy, A basis for a mathematical theory of computation, in: Computer Programming and Formal Systems, pp. 33-70, Braffort & Hirschberg (eds.), North-Holland, 1963.
- Owicki & Gries, An axiomatic proof technique for parallel programs, Acta Inf. 6, 319-340, 1976.
- Robson, A bounded storage algorithm for copying cyclic structures, C.ACM 20 (June '77), pp. 431-433.
- de Roever, Recursive program schemes: semantics & proof theory, Math. Centre Tracts 70, Math. Centre, Amsterdam, 1976.
- de Roever, On backtracking and greatest fixpoints, in Formal Description of programming concepts, E.J. Neuhold (ed.), North-Holland Publ. Comp., 1978.
- Schorr-Waite, An efficient machine-independent procedure for garbage collection in various list structures, C.ACM 10 (Aug. 1967), pp. 501-506.
- Topor, Correctness of the Schorr-Waite list marking algorithm, Memo MIP-R-104, School of Artificial Intelligence, Univ. of Edinburgh, 1974. (Published in Acta Informatica?)
- Vuillemin, Correct and optimal implementations of recursion in a simple programming language, JCSS 9, no. 3, 1974.

