On a systematic derivation of attribute evaluation algorithms
Marcelis, A.J.J.M.

Published: 01/01/1988

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. Jan. 2019
On a systematic derivation of attribute evaluation algorithms

by
A.J.J.M. Marcelis
EUT-report 88-02

AMS subject classification: 68Q50, 68Q25, 68N20.

February 1988
Preface

This note finds its origin in my Master's thesis, which was prepared at the Eindhoven University of Technology between September 1986 and May 1987 (and which carries the same title, actually).

The major part of this thesis is included here without change; only Sections 3.2 through 3.5 have been modified in order to obtain a much clearer termination argument for the algorithm(s) under consideration, and to improve the correctness proof for the circularity detection method introduced in Section 3.3 (which is actually the same method as used - but not justified - by Jalili in [Ja183]).

Also one section containing some preliminary ideas about the improvements mentioned above (originally included in Chapter 4) could then be omitted.

Finally I would like to mention that my working (at my Master's thesis as well as at the adaptation) was supervised by Prof.dr. F.E.J. Kruseman Aretz and Dr.ir. C. Hemerik, to whom I am indebted for that.

Eindhoven, February 1988

Jos Marcelis
CONTENTS

0. Introduction 1
   0.0. Motivation 1
   0.1. Notational conventions 3
1. Starting points 5
   1.0. The transformational method for classifying algorithms 5
   1.1 Attribute grammars and attribute evaluation 10
2. Basis problem and basic solutions 20
   2.0. Basic problem 20
   2.1. Basic solution 24
   2.2. Derived solutions 30
3. Derivation of some demand driven algorithms 37
   3.0. Demand driven evaluation 37
   3.1. Towards a basic demand driven algorithm 40
   3.2. Restricting the visiting of objects 47
   3.3. Termination and detection of circularities 51
   3.4. Restricting the size of $W$ 59
   3.5. Restricting the tracing of objects 60
   3.6. Removing the final nondeterminism 62
   3.7. Changing the representation of $S$: the DSW-idea 65
   3.8. Relaxing the size of $W$ 72
4. Miscellaneous 73
   4.0. On the method of the algorithm transformation 73
   4.1. On related work concerning attribute evaluation 74
References 76
0. Introduction

0.0. Motivation

The concept of an attribute grammar as it is commonly used nowadays was introduced by D.E. Knuth in [Knu68]. In broad outline, an attribute grammar (AG) is obtained from a context-free grammar (CFG) by associating a number of attributes with each of the nonterminal symbols, and associating a number of so-called semantic rules with each of the production rules of the CFG.

Attributes are either inherited or synthesized, and for every production rule $p$ the semantic rules of $p$ define (the value of) all and only the synthesized attributes of the $p$'s left-hand side nonterminal and the inherited ones of $p$'s right-hand side nonterminals in terms of (the values of) other attributes of $p$.

Thus, if you imagine a derivation tree of such an AG, composed of a number of production rules, the information contained in inherited and synthesized attributes flows downward respectively upward through the tree.

The idea of appending attributes to nonterminal symbols of a CFG was not entirely novel at the time; earlier versions, called syntax-directed translation schemes (SDTS), provided each nonterminal symbol with one synthesized attribute and each production rule with one semantic rule defining the attribute of the left-hand side nonterminal in terms of the right-hand side attributes.

In this way translations could be realized from strings in the language generated by the grammar to some desired target language; the attribute of the root of a derivation tree then contains the translation of the string generated by that tree.

The gain in providing inherited attributes together with synthesized ones undoubtedly is, that it makes context-sensitive features of programming languages (in particular matters of type-checking and scope rules; "only declared variables may be used") easier to describe, but the price to be paid for it is considerable also: As opposed to SDTS, the order in which attribute-values can be determined (by applying semantic rules) is not fixed beforehand anymore for a given derivation tree, and this evaluation process can often no longer be combined with parsing in a straightforward way. In addition, the possibility of circular dependent attributes in a tree is introduced, which may exclude evaluation altogether.

Starting in the early seventies, many people have been busy studying and trying to improve on the situation sketched above. This often resulted in the definition of certain (sub-)classes of AG for which attribute evaluation can be done in a reasonable amount of time and space, and/or in the introduction of actual strategies (i.e. attribute evaluation algorithms) to do the job for such classes. For an extensive bibliography concerning such inventions (and more) we refer to
As such investigations are (almost necessarily?) sometimes rather self-contained, a great diversity now exists among the overwhelming amount of evaluation algorithms published thus far, and it is not clear at all whether certain approaches are really that much different from each other or actually boil down to the same idea.

This thesis, now, should be regarded as a first (?), modest step to bring some structure into the great variety of existing algorithms.

The method to be followed is the so-called transformational method, the idea behind which is the following: Starting from a very abstract (i.e. little detailed) basic algorithm that is a correct (but intolerably inefficient) solution to the attribute evaluation problem, transform this algorithm into more detailed (and more efficient) ones by "adding details" to it; of course in such a way that the correctness of the solution is preserved.

Continuing this process (starting again from newly obtained solutions) a hierarchy of ever more detailed algorithms is obtained, which hopefully contains the major algorithms known from literature (and possibly even some new ones (!)).

A particular way to add details to an algorithm and to guarantee the correctness preservation of the transformation process is adopted from the work of H.B.M. Jonkers, who, in his Ph.D. thesis [Jon82], applied the transformational method to structure the major garbage collection algorithms known from literature.

One may well conclude that this thesis is based on two starting-points: the notion of AG and attribute evaluation on the one hand and the application of the transformational method (especially according to Jonkers) on the other. In fact, during the time I spent working on this paper, the emphasis seems to have shifted gradually from the former to the latter point.

The rest of this thesis is organized as follows:

In Chapter 1 an elaborate description is given of the two starting-points: First the transformational method is dealt with, after which AG and the attribute evaluation problem are discussed similar to the way this is usually done in literature. As these latter definitions turn out to be too specific to serve as a sound basis for the formulation of a sufficiently abstract evaluation algorithm, they are "lifted" a little in Chapter 2, resulting in the more abstract notion of "object evaluation". Also the basic algorithm is presented then, together with a couple of algorithms that can immediately be derived from it.

In Chapter 3 the transformational method is applied in its full extent so as to derive the beginning of a particular branch in the hierarchy of algorithms, namely that of the so-called demand driven algorithms. Finally, in Chapter 4 a couple of miscellaneous matters are discussed (in particular a short evaluation of the transformational method is presented there).
0.1. Notational conventions
Two notational matters are discussed that appear through the entire paper; additional conventions will be drawn when used for the first time.

0.1.0. On sequences
A sequence is essentially a bag in which the order of the elements is relevant. A sequence of "things" of an arbitrary kind may be denoted by listing its elements in order, enclosed by angle brackets. For example:

\[<1, 2, 1, 3>\]

denotes the sequence, call it \(R\), of integers with elements 1, 2, 1 and 3 (in that order). Notation \(<>\) is used to denote the empty sequence, and \(\oplus\) is used as a binary infix operator denoting the concatenation of sequences. \(\oplus\) is associative.

For any sequence \(S\):

i) \(\text{rng}(S)\) denotes the set of all elements of \(S\) (e.g. \(\text{rng}(R) = \{1, 2, 3\}\), \(\text{rng}(<>) = \emptyset\)).

ii) \(\text{length}(S)\) denotes the number of elements of \(S\) for instance \(\text{length}(R) = 4\), \(\text{length}(<>) = 0\).

iii) The elements of \(S\) are numbered in order from 1 through \(\text{length}(S)\), and the \(i\)-th element of \(S\) (\(i:1\leq i\leq \text{length}(S)\)) is denoted by \(S \cdot i\) (e.g. \(R \cdot 3 = 1\)).

iv) Provided that \(\text{length}(S) > 0\) holds, \(\text{first}(S)\) and \(\text{rest}(S)\) denote the first element of \(S\) respectively the sequence obtained from \(S\) by leaving out the first element, i.e. \(S = <\text{first}(S) > \oplus \text{rest}(S)\).

A stack \(S\) is a sequence on which only two operations are allowed: appending an element to respectively removing an element from the right-hand side of \(S\) (if any). As usual, these operations are called push- respectively pop-operation, where the latter is allowed only if \(\text{length}(S) > 0\). Under this same assumption \(\text{TOP}(S)\) denotes the right-most element of \(S\), and \(\text{POP}(S)\) denotes the remainder of \(S\) after removing \(\text{TOP}(S)\) from it.

0.1.1. On algorithms
For the denotation of algorithms in this thesis we shall adhere to the guarded command language of Dijkstra [Dijk76] as much as possible; be it that we allow algorithms to operate freely on such complex data-structures as sets and sequences.

Also two nondeterministic language constructs have been added to the language: the let-statement and the for-statement, to be described informally hereafter:
i) The let-statement has the form

\[
\text{Let } x : P(x) \\
\]

where \( x \) must be a fresh name in the context and \( P(x) \) is a condition in terms of \( x \). The type of \( x \) must follow from \( P(x) \), and its scope will run to the end of the language construct in which the statement occurs (e.g. do-loop, if-clause). An execution of the above statement associates a value to the name \( x \) such that \( P(x) \) holds, after which \( x \) may be used as a constant in the program. If no value for \( x \) satisfying \( P(x) \) exists then the execution of the statement is illegal and if more than one value exists then one of them is chosen nondeterministically.

ii) The for-statement has the form

\[
\text{for } x : P(x) \rightarrow S \text{ rof ,} \\
\]

where \( x \) is a fresh name, \( P(x) \) is a condition in terms of \( x \) and \( S \) is a statement sequence (in which name \( x \) need not necessarily occur). \( P(x) \) must be such that the type of \( x \) follows from it, and the scope of \( x \) is bounded by the for - rof - pair. Execution of the for-statement amounts to repeated execution of \( S \), where before each of these executions \( x \) is given a value satisfying \( P(x) \) and such that it is not given the same value twice. The value given then serves as a constant value for the execution of \( S \) in question. Thus, \( S \) is executed exactly as many times as there exist values for \( x \) satisfying \( P(x) \).

If no such value exists then the for-statement amounts to a skip, and if more than one value exists and still may be chosen then one is selected nondeterministically.

The main algorithms in this thesis (which are algorithms marking the end of a transformation-step) are given by listing successively the program-variables they make use of, the postcondition of the algorithm, its invariants and its statement sequence. For intermediate algorithms (obtained within a transformation-step) only the statement sequence will be given. As invariants and postcondition belong exclusively to a certain algorithm, their numbering will reflect the identification of the algorithm they belong to. (The identification of an algorithm will be discussed in Section 1.0)
1. Starting points

1.0. The transformational method for classifying algorithms

(Although the method is described in terms of attribute evaluation algorithms here, it applies equally well to algorithms dealing with other problems.)

In order to apply the transformational method, first of all a proper specification of the problem in question - here: attribute evaluation - should be formulated.

In view of the fact that (in the end) our research should comprise as many algorithms (known from literature) as possible, this specification should be kept as general as possible, i.e., it should not contain details that are not essential to the description of the attribute evaluation problem. (For that would exclude certain algorithms altogether.) A specification meeting this requirement is said to define the basic problem of attribute evaluation.

Next an algorithm must be presented that is a solution to the basic problem. Such an algorithm is called a basic solution and it will serve as a starting-point for the derivation of all other algorithms.

Of such a basic solution it is required that

i) it is "abstract", i.e., it contains little algorithmic detail. For example, decisions about the order of inspecting variables should be postponed as far as such an order is not of crucial importance. (Clearly this leads to nondeterministic statements.)

   Closely related to this is of course the choice of data-types with random access (such as bags and sets) to operate on.

   (Note: A sufficiently abstract formulation of the basic problem may also prevent the basic solution from containing a large amount of detail.)

ii) it can easily be proved to meet the specification, although it may be intolerably inefficient in its operation; not in the least because it need not terminate (mainly due to nondeterministic actions contained in it). Thus, we allow of partial correctness at this high level.

   As the basic solution is usually quite close to the specification itself, proving its (partial) correctness will not pose too many problems.

   (Note: of course, one should not take advantage of the allowance of partial correctness by presenting
do true → skip od

as a basic solution; this being a partially correct solution to any problem. The basic solution must be such that a totally correct algorithm can be derived from it by the method to be described hereafter.)

Then an iterative process begins of deriving more detailed (and more efficient) algorithms, starting from the basic solution.
(The process is iterative in the sense that each newly obtained algorithm may be used as a starting-point for further transformation.)

An algorithm \( B \) is derived from algorithm \( A \) by adding a so-called \textit{algorithm detail} to the latter. Such an algorithm detail is typically a requirement about the run-time behavior of algorithms (for instance - in the case of attribute evaluation - : "no attribute may be recomputed").

It is obvious that the addition of an algorithm detail may cause a certain overhead, e.g. in the above example the set of not-yet-computed attributes may have to be kept to select a candidate for evaluation.

It is required that the transformation from \( A \) to \( B \) is \textit{correctness preserving}, which means that if \( A \) is a (possibly partially) correct solution to a certain problem, then so is \( B \) (\( B \) can be totally correct, whereas \( A \) is not).

Thus \( B \) performs the same task as \( A \), be it that it is more restricted in its operation. A particular method (adopted from \cite{Jon82}) for transforming algorithms in such a way that their correctness preservation is guaranteed is presented in subsection 1.0.0.

Clearly, following the lines sketched above we can obtain more detailed solutions to the basic problem (for remember that we started from the basic solution which is a solution to this problem, and the transformations are correctness preserving).

Though, this is not enough, for most attribute evaluation algorithms known from literature (and especially the more efficient ones) are not a solution to the (very general) basic problem, but only to a restricted version of the attribute evaluation problem (simply called: a restricted problem).

That is, for their correct operation they require a certain amount of additional knowledge about the evaluation problem at hand. This observation motivates the introduction of another form of detail, called \textit{problem details}.

A problem detail should be regarded as a restriction on the data constituting the problem (e.g. the attribute grammar concerned) which is known to hold as a precondition to the attribute evaluation process. Such a restriction may allow of a more efficient solution to the problem.

Note that an algorithm that is a solution to a problem \( P \) containing certain problem details is also a solution to any problem \( P' \), containing all details of \( P \) (and maybe some more).

To summarize the foregoing, an hierarchy of attribute evaluation algorithms is obtained as follows:

Start from an abstract algorithm (the basic solution) that is a solution to the basic problem. Add
algorithm details (i.e. apply correctness preserving transformations) to this algorithm - and again to resulting ones - a number of times to obtain more detailed solutions to the basic problem.

At an appropriate moment, add a problem detail to the basic problem, so as to obtain a restricted problem, and choose one (or more) of the previously derived algorithms to serve as a starting-point for the derivation of new algorithms (via correctness preserving transformations); these new algorithms will probably make use of the problem detail given.

Then, add new problem details (to either of the existing problems), and so on.

(Here we note again, that if \( P \) is a problem and \( P' \) is a problem obtained from \( P \) by adding a problem detail, then every solution to \( P \) is also a solution to \( P' \), and hence a correct starting-point for the derivation of other solutions to \( P' \).

**Notational convention.** In an attempt to display the connection between the algorithms on the one hand and the problem- /algorithm- details they make use of on the other, the following convention is adopted (with a slight modification) from [Jon82].

Observe that an algorithm is derived as a solution to a problem with certain problem details (none in the case of the basic problem) and, in addition, certain algorithm details apply to an algorithm (again possibly none).

Now first of all a unique capital letter is associated with each problem - and algorithm - detail.

Next, every (main) algorithm we come across will be identified as

\[
\text{algorithm } i [\delta \cdot \alpha] ,
\]

where \( i \) is a natural number, \( \delta \) is a string of capital letters which denote the problem details of the problem to which algorithm \( i \) is a solutions and \( \alpha \) is a string of capital letters which denote the algorithm details that apply to algorithm \( i \).

Number \( i \) denotes the relative place of the algorithm in the sequence of algorithms in this paper and strings \( \alpha \) and \( \delta \) denote the logical place in the hierarchy of algorithms, that is, subject to the following remark:

**Remark.** We mentioned a few times already a hierarchy of algorithms to be obtained, and \([\alpha \cdot \delta]\) (belonging to an algorithm \( i \)) seems to determine the relative place of algorithm \( i \) in this hierarchy, but does it do that in a unique way? First of all, if \( \delta' \) is such that it contains all letters in \( \delta \) and even some more, then algorithm \( i \) is also a solution to the problem characterized by string \( \delta' \), which means that, in order to find the place of \( i \) in the hierarchy, string \( \delta \) must be as small as possible (i.e. it must not contain problem-details that are not actually used by algorithm \( i \)).

But then again, is \([\delta \cdot \alpha]\) uniquely identifying? On the one hand, we shall arrive a couple of times at two algorithms that have quite different statement-sequences but the same identification \([\delta \cdot \alpha]\), and on the other hand we can imagine two algorithms \( i[\delta \cdot \alpha] \) and \( j[\delta \cdot \alpha'] \) with \( \alpha \neq \alpha' \) that we would like to consider "the same", because they perform the same job; for instance think of the (iterative) stack-version and the corresponding recursive version of an algorithm.

Thus, there seems to be a two-level hierarchy: the (minimized) strings \( \delta \) impose a partial order on groups of algorithms, and within such a group the hierarchy's nature is still food for further
1.0.0. *A particular strategy for correctness preserving transformations*

Recall that the objective of transforming an algorithm $A$ into an algorithm $B$ is that $B$ be a more detailed (and more efficient) version of $A$. (This being inherent in our working from an abstract to a concrete level.)

Two characteristic goals to be pursued in this context are

i) reduction of the nondeterminism in $A$

ii) replacing variables in $A$ by variables with a more efficient representation in $B$;

examples of both will be met.

Both goals (i) and (ii) can be achieved via the replacement of a set $X$ of variables in $A$ by a set $Y$ of variables in $B$ (where in case of goal (i) $Y$ contains a variable with limited access as a substitute for a variable in $X$ with more free access - for instance think of a stack vs. a set -).

This now is exactly the policy of the correctness preserving technique according to Jonkers [Jon82], i.e. the technique essentially embodies a way to change the representation of data (in a correctness preserving manner).

At the basis of a transformation-step (from algorithm $A$ to algorithm $B$, say) is of course the formulation of the objective to be reached with this step, i.e. the formulation of the algorithm detail to be imposed. The transformation from $A$ to $B$ is then accomplished in four phases:

1. Choose fresh variables and express the relation between these new variables and the old ones (particularly those to be replaced) in a number of assertions (to be inserted in algorithm $A$). Of course the guideline for the choice of new variables must be that they allow of imposition of the algorithm detail, whereas the variables to be replaced did not.

2. Add assignments to the newly introduced variables in algorithm $A$ so as to make the assertions inserted during phase 1 hold. As it turns out, the validity of the new assertions can sometimes not be established by adding assignments alone; some replacements on the old variables must also be performed. This typically occurs in the case of a reduction of nondeterminism, where the operations on the old variables contain too much freedom, and hence do not allow of the new, stronger assertions to be made valid. It is important to note here that - as shown by Jonkers - these replacements may be based on the new assertions already, even if their validity is not fully established yet. (Which is very fortunate of course as the new assertions embody the only place where statements about the (desired) limited behaviour of the algorithm are made.)
3. Make replacements in the algorithm so as to achieve the objective of the transformation step, viz. by replacing operations on old variables by operations on new ones (in particular, one may think of the replacement of a guard in terms of the old variables by a guard in terms of new variables); thereby exploiting the relations between these variables as laid down in the assertions.

Due to these replacements the variables that were meant to be removed should be redundant now, which means that they are only used in assignments to themselves. The assignments to such redundant variables can now be removed from the program-text.

4. As the variables that were made redundant in phase 3 do not appear in the statement sequence anymore, the assertions containing them do no longer hold. These assertions must therefore be rewritten so as to remove the redundant variables from them. To do so, it is possible to put a big existential quantifier in front of the assertions, quantifying over all redundant variables, but often (not always, as we shall see) the rewriting is possible using the relations between the redundant variables and the new variables.

Four remarks.

i) As (almost) all of our algorithms consist of a single loop, we shall use invariants instead of assertions to express the relations between variables, where an invariant corresponds to four assertions: one after initialization (before the loop), one at the beginning of the loop-body, one at the end of this body and one after the loop.

ii) Above the process has been sketched in its most general form. Sometimes a transformation step does for instance not bring about the existence of redundant variables. Of course in such a case all actions concerning redundant variables must be omitted.

iii) Entirely outside the scope of the above transformation process is the guarantee of termination preservation for the newly obtained algorithm. One should still convince oneself of the fact that for instance the restriction of nondeterminism does not at the same time cause the inability to terminate. Aggravated to our situation (remember that we shall allow of partial correctness to start with), the above means that we must make sure that a transformation step does not exclude the derivation of a properly terminating algorithm later on.

iv) As was pointed out earlier, the above method only works if the objective can be achieved by changing the representation of variables; it does not cover such transformations as loop-fusion or exchanging recursion and repetition. This is the reason that we have to resort to another kind of transformation at a particular point in this thesis.
1.1. Attribute grammars and attribute evaluation.

The concept of an attribute grammar (AG), as we shall use it, was originally introduced by D.E. Knuth [Knu68] as a tool for specifying the semantics of (programming-) languages whose syntax is defined by means of a context-free grammar (CFG).

An AG $G$ is obtained from a CFG $G_0$ by associating a finite set of attributes with each nonterminal symbol of $G_0$, and associating a number of so-called semantic rules with each of the productions rules of $G_0$. Such a semantic rule (associated with production rule $p$) specifies the value of some attribute occurring in $p$ in terms of the values of certain other attributes occurring in $p$ (here an attribute is loosely said to occur in $p$ if it is associated with a nonterminal symbol that occurs in $p$).

1.1.0. Attribute grammars and related concepts.

Below we present a definition of an attribute grammar which resembles the one given in [Fil83] to a large extent. (In fact, it should be noted here that in literature there is no such thing as a uniform definition of an attribute grammar; various definitions can be encountered, although mostly the differences between them are not essential and only serve to facilitate the description of a certain aspect.)

Definition 1.0 An attribute grammar (AG) $G$ is described in the following six points.

0) {underlying context-free grammar}.

$G$ has a context-free grammar (CFG) $G_0 = (V_N, V_T, P, S)$, called the underlying CFG of $G$. As usual, $V_N$ and $V_T$ stand for the disjoint, finite sets of nonterminal respectively terminal symbols of $G_0$ (or: of $G$), $P$ denotes the finite set of production rules and $S, S \in V_N$, is the start symbol of $G_0$ (or: of $G$).

We assume that $G_0$ is a reduced CFG in the sense that each nonterminal symbol

i) can be derived from $S$, and

ii) can produce at least one string $w \in V_T^*$

by zero or more applications of rules in $P$.

It is also assumed that the start symbol $S$ does not occur at the right-hand side of any production rule in $P$.

Production rule $p \in P$ is denoted as

$$p : X_0 \rightarrow w_0 X_1 w_1 \cdots w_{np-1} X_{np} w_{np} ,$$

where $np \geq 0$ and, for each $i : 0 \leq i \leq np$, $X_i \in V_N$ and $w_i \in V_T^*$. This is called the usual form of a production rule of $G$. (This form will be used in the sequel to denote an arbitrary production rule of $G$.)

When considering a derivation tree of $G_0$ (also called a derivation tree of $G$) we assume it to be a complete derivation tree, that is, its leaf nodes are labeled with a terminal symbol and its root is
labeled with the start symbol $S$.

The set of all complete derivation trees is denoted by $CDT(G)$.

A leaf node is also called a terminal node (as opposed to a nonterminal node) and, for each tree $t$, $\text{yield}(t)$ denotes the string of labels of the leaf nodes of $t$, concatenated from left to right.

An occurrence of production rule $p$ (of the usual form) in $t$ is a portion of $t$ consisting of a node $u$ labeled with $X_0$ together with its direct descendants in $t$ whose labels, concatenated from left to right, form the string

$$w_0 X_1 w_1 \cdots w_{np-1} X_{np} w_{np}. $$

We say that $p$ is applied at node $u$ and we denote with $u\cdot 1$, $u\cdot 2$, \ldots, $u\cdot np$ the direct descendants of $u$ in $t$ whose labels are $X_1$, $X_2$, \ldots, $X_{np}$, respectively. Thus, $u$, $u\cdot 1$, \ldots, $u\cdot np$ are the nonterminal nodes of the occurrence of $p$.

(N.b.: For ease of description, node $u$ is often also addressed as $u\cdot 0$ in this context, making $u\cdot 0$, $u\cdot 1$, \ldots, $u\cdot np$ the nonterminal nodes of the occurrence of $p$.)

1) \{semantic domain\}.

$G$ has a semantic domain $D = (\Omega, \Phi)$, where $\Omega$ is a set of sets, called sets of attribute values and $\Phi$ is a collection of functions of type

$$V_1 \times \cdots \times V_m \rightarrow V_0,$$

where $m \geq 0$ and, for each $i : 0 \leq i \leq m$, $V_i \in \Omega$. The elements of $\Phi$ are called semantic functions. For each function $f \in \Phi$ we require that $f$ is a total function.

2) \{attribute names\}.

$G$ has a set, $\text{ATT-NAMES}(G)$, of attribute names. This set is partitioned into two subsets, $\text{IN-NAMES}(G)$ and $\text{SYN-NAMES}(G)$, of inherited and synthesized attribute names, respectively. Each name $a \in \text{ATT-NAMES}(G)$ has associated with it an element of $\Omega$, denoted by $V(a)$.

3) \{attributes of a nonterminal\}.

To each nonterminal symbol $X \in V_N$ a subset of $\text{ATT-NAMES}(G)$, denoted $\text{ATT-NAMES}(X, G)$, is attached. If $a \in \text{ATT-NAMES}(X, G)$ then $a(X)$ is an attribute of nonterminal symbol $X$.

$a(X)$ is called inherited or synthesized (shortly: i- or s-attribute) if $a \in \text{IN-NAMES}(G)$ of $a \in \text{SYN-NAMES}(G)$, respectively.

We denote with $\text{ATT}(X, G)$ the set of all attributes of $X$, and with $\text{IN-ATT}(X, G)$ and $\text{SYN-ATT}(X, G)$ the (disjoint) sets of its i- and s-attributes, respectively. (Note that, for different nonterminal symbols $X$ and $Y$, $\text{ATT}(X, G) \cap \text{ATT}(Y, G) = \emptyset$, whereas $\text{ATT-NAMES}(X, G)$ and $\text{ATT-NAMES}(Y, G)$ may overlap.)

For the start symbol $S$ of $G$ it is required that $\text{IN-ATT}(S, G) = \emptyset$. Finally, the set of all attributes of $G$, $\text{ATT}(G)$, is defined as
\[ \text{ATT}(G) = \left( \bigcup X : X \in V_N : \text{ATT}(X, G) \right). \]

Attribute \( a(X) \) can take on any value from \( V(a) \), i.e. the element of \( \Omega \) associated with attribute name \( a \).

4) \{ attributes of a production rule \}.

Let \( p \) be the production rule of the usual form
\[ p : X_0 \rightarrow w_0 X_1 w_1 \cdots w_{np-1} X_{np} w_{np} \]
and let \( a(X_i) \) be an attribute of \( X_i \) for some \( 0 \leq i \leq np \). Then the pair \( \langle a(X_i), i \rangle \) is an attribute of production rule \( p \). The set of all attributes of \( p \) is denoted by \( \text{ATT}(p, G) \):
\[ \text{ATT}(p, G) = \{ \langle a(X_i), i \rangle | 0 \leq i \leq np \land a(X_i) \in \text{ATT}(X_i, G) \} \]

An attribute \( \langle a(X_i), i \rangle \) of \( p \) is either a defined attribute of \( p \) or a used attribute of \( p \); it is a defined attribute if
\[ (i = 0 \land a(X_i) \in \text{SYN-ATT}(X_i, G)) \lor (i \neq 0 \land a(X_i) \in \text{IN-ATT}(X_i, G)), \]
whereas it is a used one if
\[ (i = 0 \land a(X_i) \in \text{IN-ATT}(X_i, G)) \lor (i \neq 0 \land a(X_i) \in \text{SYN-ATT}(X_i, G)) \]

An attribute \( \langle a(X_i), i \rangle \) of \( p \) is simply denoted \( \langle a, i \rangle \), when this causes no confusion.

5) \{ Semantic rules of a production \}.

Associated with production rule \( p \) of the usual form is a set of semantic rules, denoted by \( \text{RULES}(p, G) \). Each semantic rule of \( p \) is specified by a semantic function \( f \in \Phi \), say of type \( V_1 \times \cdots \times V_m \rightarrow V_0 \) for some \( m : m \geq 0 \), and a sequence of \( m + 1 \) attributes of \( p \)
\[ \langle a_0(X_{j_0}), j_0 \rangle, \langle a_1(X_{j_1}), j_1 \rangle, \ldots, \langle a_m(X_{j_m}), j_m \rangle \]
where, for all \( k : 0 \leq k \leq m, 0 \leq j_k \leq np \) and \( V(a_k) = V_k \).

Such a semantic rule is written as
\[ \langle a_0(X_{j_0}), j_0 \rangle = f(\langle a_1(X_{j_1}), j_1 \rangle, \ldots, \langle a_m(X_{j_m}), j_m \rangle) \]
and we say that it defines the attribute \( \langle a_0(X_{j_0}), j_0 \rangle \) using the attributes \( \langle a_1(X_{j_1}), j_1 \rangle, \ldots, \langle a_m(X_{j_m}), j_m \rangle \) as arguments, or that \( \langle a_0(X_{j_0}), j_0 \rangle \) depends in \( p \) on the attributes \( \langle a_1(X_{j_1}), j_1 \rangle, \ldots, \langle a_m(X_{j_m}), j_m \rangle \). When no confusion arises, this semantic rule is simply denoted \( \langle a_0, j_0 \rangle = f(\langle a_1, j_1 \rangle, \ldots, \langle a_m, j_m \rangle) \).

For the set \( \text{RULES}(p, G) \) it is required that

i) The semantic rules in \( \text{RULES}(p, G) \) define only defined attributes of \( p \) and for each defined attribute of \( p \) there is exactly one semantic rule defining it.
ii) The semantic rules in \( \text{RULES}(p, G) \) use only used attributes of \( p \) as arguments.

(An AG that meets this requirement is said to be in Bochmann normal form [Boc76].)

The semantic rules of production rule \( p \) create dependencies among the latter's attributes that can be visualized by means of a directed graph. Therefore consider the following definition:

**Definition 1.1** Let \( G \) be an AG and \( p \) a production rule (of the usual form) of \( G \). The directed graph \( pg(p, G) \), called the *production graph* of \( p \), has as nodes the attributes of \( p \) and an edge runs from attribute \( <a, j> \) to attribute \( <b, k> \) if and only if there is a semantic rule in \( \text{RULES}(p, G) \) that defines \( <b, k> \) using \( <a, j> \) as an argument.

Just like attributes of a production rule of \( G \) were defined in definition 1.0 (4), attributes of a derivation tree of \( G \) can be defined in a fairly straightforward manner:

**Definition 1.2** Let \( G \) be an AG and \( t \in CDT(G) \). Let node \( u \) of \( t \) be labeled with symbol \( X \in V_N \). If \( a(X) \in \text{ATT}(u, G) \) then \( <a(X), u> \) is an *attribute of node \( u \).* We denote with \( \text{ATT}(u, G) \) the set of all attributes of node \( u \):

\[
\text{ATT}(u, G) = \{ <a(X), u> | a(X) \in \text{ATT}(X, G) \}
\]

and with \( \text{ATT}(t, G) \) the set of all attributes of all nodes of \( t \):

\[
\text{ATT}(t, G) = (\bigcup u : u \text{ is a nonterminal node of } t : \text{ATT}(U, G)).
\]

An element \( <a(X), u> \) of \( \text{ATT}(t, G) \) is called an *attribute of derivation tree \( t \)* and it is denoted simply by \( <a, u> \) when this causes no confusion.

If production rule \( p \) occurs somewhere in derivation tree \( t \) of \( G \) (see definition 1.0 (0) for this notion) then each attribute of \( p \) gives rise to an attribute of \( t \) and, moreover, the dependencies that exist between the corresponding attributes of \( t \). Finally, semantic rules of \( p \) can be translated into semantic rules of \( t \):

**Definition 1.3** Let \( G \) be an AG and \( t \in CDT(G) \).

a) The directed graph \( dtg(t, G) \), called the *derivation tree graph* of \( t \), has as nodes the elements of \( \text{ATT}(t, G) \) and edges as follows: Let production rule \( p \) (of the usual form) occur in \( t \) and such, that \( u-0, u-1, \ldots, u-np \) are the nonterminal nodes of this occurrence (cf. definition 1.0. (0)). Then an edge runs from attribute \( <a, u-i> \) to attribute \( <b, u-j> \) in \( dtg(t, G) \) if and only if an edge runs from attribute \( <a, i> \) to attribute \( <b, j> \) in \( pg(p, G), \) for \( i, j : 0 \leq i \leq np \land 0 \leq j \leq np \).
b) Let production rule $p$ occur in $t$ (as in part (a)) and let $<a_0, j_0> = f(<a_1, j_1>, \ldots, <a_m, j_m>)$ be in $\text{RULES}(p, G)$. Then $<a_0, w j_0> = f(<a_1, w j_1>, \ldots, <a_m, w j_m>)$ is a semantic rule of derivation tree $t$ and analogous to definition 1.0. (5) we say that it defines the attribute $<a_0, w j_0>$ using the attributes $<a_1, w j_1>, \ldots, <a_m, w j_m>$ as arguments.

$\text{RULES}(t, G)$ denotes the set of all semantic rules of $t$.

Two remarks should be made in connection with definition 1.3.
Firstly, it is easy to see that for each attribute of $t$ there is exactly one semantic rule in $\text{RULES}(t, G)$ defining it (use the fact that $G$ is in Bochmann normal form). In view of the definition of attribute evaluation (still to come) this is a very fortunate circumstance.

Secondly, graph $\text{dtg}(t, G)$ can be thought of as the result of "pasting together" the graphs $\text{pg}(\cdot, G)$ of the production rules that $t$ is composed of. Note however, that in this way graph $\text{dtg}(t, G)$ may contain circularly dependent attributes, as opposed to the constituent graphs $\text{pg}(\cdot, G)$ (Bochmann normal form!): 

Definition 1.4 An AG $G$ is noncircular (called "well-formed" in [Knu68]) if for each derivation tree $t \in \text{CDT}(G)$ the graph $\text{dtg}(t, G)$ contains no circularly dependent attributes.

1.1.1. An example

In order to illustrate the notions introduced above, and as a proposal for the denotation of graphs $\text{pg}$ and $\text{dtg}$, we give the following example (which is actually a part of Knuth's original example, presented in [Knu68]):

A string consisting of symbols zero and one (0 and 1) can be understood to denote a natural number in binary notation. Using a well-known conversion rule, such a string can be transformed into a string consisting of symbols 0 through 9, denoting the same natural number in decimal notation.

Based on a CFG $G_0$ that can produce any string consisting of zero's and one's, we present below an AG $G$ that gives the decimal "meaning" of such a string.

i) $G_0 = (V_N, V_T, P, S),$ where

$V_N = \{ S, L, B \} \quad V_T = \{ 0, 1 \}$,

$P = \{ S \rightarrow L, L \rightarrow LB, L \rightarrow B, B \rightarrow 0, B \rightarrow 1 \}$
ii) $G$ has semantic domain $D = (\Omega, \Phi)$, where $\Omega$ has as its only element the set $\mathbb{N}$ of natural numbers and $\Phi$ consists of five functions:
- the constant function zero which always delivers the value $0 \in \mathbb{N}$.
- functions $id, f_1, f_2$, all of type $\mathbb{N} \to \mathbb{N}$ with
  \[
  id(n) = n \\
  f_1(n) = n + 1 \\
  f_2(n) = 2^n.
  \]
- function $f_3: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ with
  \[
  f_3(n, m) = n + m.
  \]

iii) $\text{IN-NAMES}(G) = \{r\}$, $\text{SYN-NAMES}(G) = \{v\}$, and (of course) $IN$ is the element of $\Omega$ associated both with $r$ and $v$.

iv) The nonterminal symbols of $G$ have attributes as follows:
- $\text{ATT}(S, G) = \{v(S)\}$,
- $\text{ATT}(L, G) = \{v(L), r(L)\}$,
- $\text{ATT}(B, G) = \{v(B), r(B)\}$.
Observe, using (iii), that the attribute $v(\cdot)$ ("value") is synthesized and the attribute $r(\cdot)$ ("rank") is inherited. Observe also that the start symbol $S$ has no inherited attributes.

v) Below we give the semantic rules for each of the production rules $p \in P$, accompanied with the corresponding production graphs $pg(p, G)$.

\[
\text{rule } S \rightarrow L : \\
S \quad r \quad v \\
L \quad r \quad v \\
< v, 0 > \not\in < v, 1 > \\
< r, 1 > \not\in 0
\]

\[
\text{rule } L \rightarrow LB : \\
L \quad r \quad v \\
< v, 0 > \not\in < v, 1 > + < v, 2 > \\
< r, 1 > \not\in < r, 0 > + 1 \\
< r, 2 > \not\in < r, 0 >
\]

\[
\text{rule } L \rightarrow B : \\
L \quad r \quad v \\
B \quad r \quad v \\
< v, 0 > \not\in < v, 1 > \\
< r, 1 > \not\in < r, 0 >
\]
Observe that $G$ is in Bochmann normal form (cf. definition 1.0. (5)).

As an example, let us consider the graph $dig(t, G)$ where $t$ is the derivation tree for string 0101. For each nonterminal node $u$ of $t$ the correct values (i.e. in accordance with the semantic rules) of $u$'s attributes are also given. (Here we rely on the reader having an intuitive idea what it means for an attribute to "have a value", and even "in accordance with the semantic rules", as these notions still have to be defined (Section 1.1.2).)
It is easy to see that AG $G$ is noncircular.
For any derivation tree $t$, the value of the synthesized attribute of the root of $t$ plays a special role: it contains the meaning of the string $\text{yield}(t)$ (sometimes simply called the meaning of $t$). Now observe from the above example that there can be attributes of $t$ that do not "contribute" to the meaning of $t$, as no directed path runs from these attributes to the attribute of $t$'s root. Therefore the meaning of $t$ can be determined without computing the value of these attributes.

1.1.2. Attribute evaluation

This section deals with the notion of attribute evaluation for an AG $G$ (with semantic domain $D = (\Omega, \Phi)$).
Informally speaking the idea is, given a derivation tree $t \in \text{CDT}(G)$, to assign a value to "sufficiently many" attributes of $t$ (i.e. elements of $\text{ATT}(t, G)$) in such a way that for each attribute $<a, u>$ - whenever it has a value assigned to it - this value

i) is of the correct type, i.e. is an element of $V(a)$, the set in $\Omega$ associated with attribute name $a$.

ii) is in accordance with the values of certain other attributes of $t$, such as prescribed by the unique semantic rule in $\text{RULES}(t, G)$ defining $<a, u>$.

To account for the recording of values a (variable) partial function

$$\text{val} \in \text{ATT}(t, G) \rightarrow (\bigcup_{p} V : V \subseteq \Omega : V)$$

is introduced. If $<a, u> \in \text{dom(val)}$ then we say that attribute $<a, u>$ is evaluated and $\text{val}(<a, u>)$ should be interpreted as the value of $<a, u>$. Also, to decide whether sufficiently many attributes have been evaluated we assume a set $W$, $W \subseteq \text{ATT}(t, G)$, to be given on forehand, i.e. $W$ is a part of the specification for a concrete attribute evaluation problem. $W$ should be interpreted as the set of so-called wanted attributes, i.e. attributes of $t$ that must be evaluated in any case. Thus, in view of the fact that $\text{dom(val)}$ reflects the set of evaluated attributes, sufficiently many attributes have been evaluated if $W \subseteq \text{dom(val)}$.

Note. Here we note that in most practical situations the set $W$ is chosen equal to $\text{ATT}(t, G)$, i.e. all attributes of $t$ must be evaluated, or $W$ equals the synthesized attribute(s) of the root of $t$, in which case these attribute(s) is (are) said to contain the meaning of derivation tree $t$; see for instance the example in the preceding section.
In the latter case it is often not necessary to evaluate all attributes of $t$, as some of them do not contribute to the meaning of $t$, i.e. there is no directed path in $\text{dtg}(t, G)$ running from these attributes to an attribute of the root. Again the preceding section provides an example of this.

In order to express this informal description into more formal terms, the following definition will be helpful.
Definition 1.5 Let \( G \) be an AG and \( t \in \text{CDT}(G) \).
For every \( U \subseteq \text{ATT}(t, G) \), and partial function \( \text{val} \in \text{ATT}(t, G) \xrightarrow{p} \bigcup V : V \in \Omega : V \) we define
(a) \( U \) is self-contained if for all \( <a_o, u_o> \in U \) with
\[<a_o, u_o> = f(<a_1, u_1>, \ldots, <a_m, u_m>)\]
as defining semantic rule in \( \text{RULES}(t, G) \):
\[(A_i : 1 \leq i \leq m : <a_i, u_i> \in U)\]
b) \( U \) is consistent with respect to \( \text{val} \) if \( U \) is self-contained and if, in addition:
\[i) \quad U \subseteq \text{dom(\text{val})} \]
and for all \( <a_o, u_o> \in U \) as in part (a):
\[ii) \quad \text{val}(<a_o, u_o>) \in V(a_0) \]
\[iii) \quad \text{val}(<a_o, u_o>) = f(\text{val}(<a_1, u_1>), \ldots, \text{val}(<a_m, u_m>))\]

Using definition 1.5., the attribute evaluation problem for given \( G, t, W \) now consists in determining a set \( U \subseteq \text{ATT}(t, G) \) and a mapping \( \text{val} \in (\text{ATT}(t, G) \xrightarrow{p} \bigcup V : V \in \Omega : V) \) such that
\[U \supseteq W \land \text{U consistent w.r.t. \text{val}}\]

If we let
\[Q_o = G \land t \in \text{CDT}(G) \land W \subseteq \text{ATT}(t, G)\]
\[R = U \supseteq W \land \text{U consistent w.r.t. \text{val}}\]
then the following specifies the attribute evaluation problem:

\[S : \begin{align*}
&| G, t, W \{Q_o\} \\
&\quad ; | U : \text{set of attributes of } t \\
&\quad ; \text{val} : \text{ATT}(t, G) \xrightarrow{p} \bigcup V : V \in \Omega : V \\
&\quad ; S \{R\} \\
&\end{align*}\]

By adding problem details we obtain more restricted versions of this problem, specified by

\[S : \begin{align*}
&| G, t, W \{Q_o \land Q_1\} \\
&\quad ; | U : \text{set of attributes of } t \\
&\quad ; \text{val} : \text{ATT}(t, G) \xrightarrow{p} \bigcup V : V \in \Omega : V \\
&\quad ; S \{R\} \\
&\end{align*}\]

where \( Q_1 \) is condition in terms of \( G, t \) and/or \( W \) which denotes the conjunction of all problem details that apply to the situation.
(For the notion of a problem detail see Section 1.0.)
However, it is believed that specification (0) in its own right already contains a considerable amount of problem detail (hidden in $Q_0$), and therefore (0) is not a suitable specification for the basic problem according to section 1.0.
Thus, first of all a more abstract formulation of the problem will be presented in the next chapter.
2. Basic problem and basic solutions

2.0. Basic problem.

For an attribute grammar $G$ the semantic rules are defined within the scope of the single production rules, which is commonly regarded as an attractive property of AG, because it makes the semantics of a language construct fairly easy to read.

As a consequence, for a derivation tree $t \in \text{CDT}(G)$ only "local" dependencies exist between the attributes of $t$, i.e. only attributes of neighboring nodes in $t$ are related to each other via semantic rules.

This property is a useful one because it gives rise to a particular class of attribute evaluation algorithms (the so-called "tree-walking evaluators"), but from a conceptual point of view the "locality" of attributes in a semantic rule constitutes irrelevant information: It really is a problem detail and therefore the formulation of the basic problem can do without it.

An obvious way to abstract from this detail is to turn attributes (which are pairs $<a, u>$) into more abstract "objects" (an object being a not further specified primitive concept), thereby removing the entire underlying tree-structure from the discussion; What remains is basically the graph $\text{dg}(t, G)$ (see definition 1.3.) in abstract disguise: it has objects instead of attributes as its nodes and tree $t$ can no longer be recognized in it. (Also the distinction between inherited and synthesized attributes has disappeared, as this information was hidden in the first component of the pair $<a, u>$.)

In subsection 2.0.0 we present the notion of an 'object evaluation structure' and the object evaluation problem is defined. Next, in subsection 2.0.1 it is shown how a triplet $(G, t, W)$ (constituting the date for a concrete attribute evaluation problem) can be transformed into an object evaluation structure by applying an abstraction function to it. Since, via this abstraction function, every solution to the object evaluation problem corresponds to a solution to the attribute evaluation problem of section 1.1, it then follows that the object evaluation problem can be taken as definition of the basic problem (of attribute evaluation).
2.0.0. Object evaluation

Definition 2.0. An object evaluation structure $Q$ is a 7-tuple

$$Q = (OB, \Omega, \Phi, \text{type}, \text{func}, \text{args}, W),$$

where

- $OB$ is a finite set
- $\Omega$ is a finite set of sets
- $\Phi \subseteq (\bigcup m, V_0, V_1 \cdots V_m : m \geq 0 \land (A i : 0 \leq i \leq m : V_i \in \Omega) : V_1 \times \cdots \times V_m \rightarrow V_0)$
- $\text{type} \in (OB \rightarrow \Omega)$
- $\text{func} \in (OB \rightarrow \Phi)$
- $\text{args} \in (OB \rightarrow OB^*)$
- $W \subseteq OB$.

Here, and in the sequel, functions denoted by $\rightarrow$ are total functions. The various components of $Q$ are subject to the following restriction:

$$\begin{align*}
(A x : x \in OB : \text{func}(x) &\in \\
(X i : 1 \leq i \leq \text{length} (\text{args}(x)) : \text{type}(\text{args}(x) \cdot i)) &\rightarrow \text{type}(x)
\end{align*}$$

For an object evaluation structure $Q$ as above, the elements of $OB$ are called objects, elements of $\Omega$ are called sets of object values, and for each object $x \in OB \ \text{type}(x)$ is the type of $x$, $\text{func}(x)$ is the function defining $x$ and $\text{args}(x)$ contains the arguments of $x$.

Finally, $W$ is called the set of wanted objects.

Associated with object evaluation structures are definition 2.1, definition 2.2 and property 2.3:

Definition 2.1. Let $Q = (OB, \Omega, \Phi, \text{type}, \text{func}, \text{args}, W)$ be an object evaluation structure. For every $U \subseteq OB$ and partial function $\text{val} \in (OB \xrightarrow{p} (\bigcup V : V \in \Omega : V))$

a) $U$ is self-contained if for all $x \in U$

$$\text{rng}(\text{args}(x)) \subseteq U.$$

b) $U$ is consistent w.r.t. $\text{val}$ if $U$ is self-contained and if, in addition

i) $U \subseteq \text{dom}(\text{val})$

and for all $x \in U$

ii) $\text{val}(x) \in \text{type}(x)$

iii) $\text{val}(x) = \text{func}(x) (\text{VAL}(\text{args}(x)))$
(Nb 0: Here, and in the sequel, \((\text{VAL(args}(x)))\) is a shorthand notation for \((\text{val(args}(x) \cdot 1), \cdots, \text{val(args}(x) \cdot \text{length(args}(x)))\))\), i.e., the tuple containing (in order) the results of the expressions \(\text{val(args}(x) \cdot i)\), for \(i : 1 \leq i \leq \text{length(args}(x))\).

Nb 1: The fact that we use the same terms (such as 'consistent') for related concepts both in connection with attribute evaluation and object evaluation will cause no problems, as we shall pretty soon forget about attribute evaluation altogether and turn exclusively to object evaluation.

Nb 2: As could be expected, we shall refer to \(\text{val}(x)\) as the \text{value} of object \(x\), whenever \(x \in \text{dom(val)}\) holds.

**Definition 2.2** Let \(Q = (\text{OB}, \Omega, \Phi, \text{type}, \text{func}, \text{args}, W)\) be an object evaluation structure, and let \(x \in \text{OB}\) and \(y \in \text{OB}\).

a) \(x\) \emph{directly depends on} \(y\), denoted \(y \rightarrow x\), if \(y \in \text{rng(args}(x))\).

b) relations \(\rightarrow^+\) ('depends on') and \(\rightarrow^*\) denote the transitive respectively the reflexive and transitive closure of \(\rightarrow\).

c) \(Q\) is \emph{acyclic} (or: noncircular) if \((\forall x : x \in \text{OB} : \neg (x \rightarrow^+ x))\); otherwise \(Q\) is \emph{cyclic} (circular).

**Property 2.3** Let \(Q\) as above and let \(U \subseteq \text{OB}\). Then:

\(U\) \emph{self-contained} \iff \((\forall y : y \in \text{OB} : (\exists x : x \in U : y \rightarrow^* x) : y \in U)\)

For a given object evaluation structure \(Q\) (as above) the object evaluation problem consists in determining a set \(U \subseteq \text{OB}\) and a mapping \(\text{val} \in (\text{OB} \xrightarrow{p} (\bigcup V : V \in \Omega : V))\) such that

\(U \supseteq W \land U\) \text{ consistent w.r.t. val}.

That is, the following is a specification for this problem:

\[ S : \llbracket \text{OB}, \Omega, \Phi, \text{type}, \text{func}, \text{args}, W \rrbracket \{P\} \]
\[
\llbracket U : \text{set of objects} \rrbracket
; \text{val} : \text{OB} \xrightarrow{p} (\bigcup V : V \in \Omega : V) \quad (1)
; S \{R\}
\]
\]

where \(P\) and \(R\) stand for
\[
P \equiv (\text{OB}, \Omega, \Phi, \text{type}, \text{func}, \text{args}, W)\) is an object evaluation structure
\[
R \equiv U \supseteq W \land U\) \text{ consistent w.r.t. val} \]
2.0.1. Relating attribute evaluation to object evaluation

Let \((G, t, W)\) be such that \(G\) is an AG (with semantic domain \(D = (\Omega, \Phi)\), \(t \in CDT(G)\) and \(W \subseteq ATT(t, G)\). Thus, the triplet \((G, t, W)\) constitutes the data for a concrete attribute evaluation problem.

\((G, t, W)\) can be transformed into an object evaluation structure \(Q = (OB, \Omega, \Phi, \text{type, func, args, } W')\) using an abstraction function

\[ \alpha : ATT(t, G) \to OB \]

where \(\alpha\) is one-to-one and surjective, and such that the following relations hold between \((G, t, W)\) and the components of \(Q\):

(i) \(W' = \{ \alpha(<a, u>) \mid <a, u> \in W \}\)

and for all \(<a_0, u_0> \in ATT(t, G)\) such that \(<a_0, u_0> = f(<a_1, u_1>, \ldots, <a_m, u_m>)\)

is the semantic rule in RULES\((t, G)\) defining \(<a_0, u_0>\):

(ii) type \((\alpha(<a_0, u_0>)) = V(a_0)\)

(iii) func \((\alpha(<a_0, u_0>)) = f\)

(iv) args \((\alpha(<a_0, u_0>)) = \alpha(<a_1, u_1>), \ldots, \alpha(<a_m, u_m>)\)

and, of course, \(\Omega\) and \(\Phi\) are equal to the corresponding elements of the semantic domain of \(G\).

For a concrete \((G, t, W)\) and \(Q\), related to each other via an \(\alpha\) as described above, the evaluation problems (respectively attribute- and object-) can be easily seen to coincide in the sense that for each attribute in \(ATT(t, G)\) its type, arguments and defining function are preserved by \(\alpha\) and moreover, the sets \(W\) and \(W'\) of wanted attributes/objects and the notion of consistency for attributes/objects are in agreement with each other.

Now suppose that \(S\) is a solution to the object evaluation problem, i.e. it meets specification (1) above. Then \(S\) can (appropriately interpreted) also be regarded as a solution to the attribute evaluation problem, specified by (0) in section 1.1.2. Namely: A concrete \((G, t, W)\) can be turned into an object evaluation structure \(Q = (OB, \Omega, \Phi, \text{type, func, args, } W')\), using a certain function \(\alpha_0\). For this \(Q\) \(S\) then comes up with an \(U'\) and \(val'\) such that

\[ U' \supseteq W' \wedge U' \text{ is (object--) consistent w.r.t. } val' \]

Interpreting the objects operated on by \(S\) as attributes again (via the "interpretation function" \(\alpha_0^{-1}\); note that it exists) yields \(U\) and \(val\) satisfying

\[ U = \{ \alpha_0^{-1}(x) \mid x \in U' \}\]
\[ \text{dom}(\text{val}) = \{ \alpha_0^{-1}(x) \mid x \in \text{dom}(\text{val}') \}\]

such that

\[ (A <a, u> : <a, u> \in \text{dom}(\text{val}) \mid \text{val}(<a, u>) = \text{val}'(\alpha_0(<a, u>))) \]
Thus, each solution to the object evaluation problem yields a solution to the attribute evaluation problem, or, stated differently, the latter problem is a special case of the former one (containing some problem details). Therefore the object evaluation problem may be taken as the basic problem (for attribute evaluation).

2.1. Basic solution

Convention 2.4. To save ourselves the trouble of inserting everywhere the phrase "For every object evaluation structure \( Q = \ldots \)" we assume in the remainder of this thesis that there is only one such structure

\[ Q = (OB, \Omega, \Phi, \text{type}, \text{func}, \text{args}, W) \]

in accordance with Definition 2.0. \( Q \) should be regarded as a prototype of all object evaluation structures. Also, the problem is to construct a set \( U : U \subseteq OB \) and a mapping \( \text{val} : \text{val} \in (OB \rightarrow (\bigcup V : V \in \Omega : V)) \) such that (according to Definition 2.1):

\[ U \supseteq W \land U \text{ consistent w.r.t. val}. \]

To derive the basic solution to the (object) evaluation problem, consider the desired postcondition

\[ R_0 : U \supseteq W \land U \text{ consistent w.r.t. val} \]

We propose to relax the first conjunct of \( R_0 \) and to keep the second one as an invariant to our algorithm (extended with a self-evident bound for \( U \)):

\[ \text{P0.0} : U \subseteq OB \land U \text{ consistent w.r.t. val} \]

On account of the definition of consistency, \( \text{P0.0} \) implies \( U \subseteq \text{dom(val)} \). It makes sense, however, to maintain the stronger condition

\[ \text{P0.1} : U = \text{dom(val)} \]

as an additional invariant; that is, \( U \) now has the following interpretation

\( U \) contains the evaluated objects, i.e. the objects that have already been assigned a value to.

(N.b.: Throughout, it is assumed that an assignment to \( \text{val}(x) \), for some object \( x \), automatically establishes the condition \( x \in \text{dom(val)} \). Thus, no statements of the form "\( \text{dom(val)} := \text{dom(val)} \cup \{x\} \)" are necessary.)

The initialization \( U := \emptyset \) establishes \( \text{P0.0} \land \text{P0.1} \), and our intention is to add objects to \( U \) under invariance of \( \text{P0.0} \land \text{P0.1} \), until condition \( U \supseteq W \) is satisfied.
Under what conditions is an object $x$ *evaluatable*, i.e. can it be put in $U$ without violating the invariants?

The invariance of P0.1 causes no problems; it merely states that each addition to $U$ should be accompanied by an assignment to $\text{val}$. Now consider P0.0. Since consistency implies self-containedness, $\text{rng}(\text{args}(x)) \subseteq U$ should hold as a precondition to the addition of $x$ to $U$. This condition also turns out to be sufficient, because the assignment $\text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x)))$ then preserves the consistency of $U$ w.r.t. $\text{val}$ (cf. Definition 2.1):

i) $U \subseteq \text{dom}(\text{val})$ is implied by P0.1.

ii) $\text{val}(x) \in \text{type}(x)$ is also established, namely:

   $\text{rng}(\text{args}(x)) \subseteq U \land U \text{ consistent w.r.t. } \text{val}$

   $\Rightarrow \{\text{definition 2.1(b), part ii}\}$

   $(\forall y : y \in \text{rng}(\text{args}(x)) : \text{val}(y) \in \text{type}(y))$

   $\Rightarrow \{\text{type conformity of } \text{func}(x), \text{see def. 2.0 ; application of } \text{func}(x) \text{ is allowed} \}$

   $\text{val}(x) \in \text{type}(x)$.

iii) $\text{val}(x) = \text{func}(x) (\text{VAL}(\text{args}(x)))$ is obvious now.

Apparently two tasks can now be discerned (to be taken care of by our algorithm):

- $t_0$ : selecting evaluatable objects
- $t_1$ : the actual evaluation of objects

(It may seem obvious to combine these tasks right away; from the viewpoint of separation of concerns we prefer to do not.)

Variable $U$ is already concerned with task $t_1$. To model the progress of task $t_0$ we introduce variable $V$, a sequence of objects, with the following interpretation:

$V$ contains the objects which - although not yet evaluated - are known to be evaluatable in the order of their appearance in $V$.

(Thus, $V$ represents an *evaluation order* for $\text{rng}(V)$). On account of the preceding paragraph (and the interpretation of $U$), this interpretation can be translated immediately into the following additional invariants:

P0.2: $\text{rng}(V) \subseteq \text{OB}$

P0.3: $(\forall i, j : 1 \leq i < j \leq \text{length}(V) : V.i \neq V.j)$

P0.4: $(\forall V_0, V_1, x : V = V_0 \oplus x > \oplus V_1 : \text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V_0))$

P0.5: $U \cap \text{rng}(V) = \emptyset$

(Note that, for every $V_0, V_1$ such that $V = V_0 \oplus V_1$, $P0.0 \land P0.4 \Rightarrow U \cup \text{rng}(V_0)$ is self-contained.

However, taking
P0.4 alt: \((A V_0, V_1 : V = V_0 \oplus V_1 : U \cup \text{rng}(V_0))\) is self-contained)

as an alternative formulation for P0.4 would not be sufficient to formalize the interpretation of \(V\): Consider the case that \(V_0 = <x>\) for some object \(x\) such that \(x \rightarrow x\) holds. Although \(U \cup V_0\) can be self-contained, \(x\) may not be evaluable.

The alternative formulation would suffice, however, if \(\neg (x \rightarrow x)\) holds for all \(x : x \in OB.\)

All in all we arrive at the following algorithm as a first approach to the basic solution. With each iteration-step, the algorithm will be able to choose nondeterministically between contributing to task t0 (by selecting an evaluable object; or at least trying to do so) and contributing to task t1 (by evaluating an object).

On account of P0.4 the object \(\text{first}(V)\) is a candidate for evaluation, provided it exists (i.e. \(V \neq < >\)), and by P0.3 \& P0.5 only objects outside \(U \cup \text{rng}(V)\) may be added to \(V\). For the rest the algorithm reflects the ideas developed above in a straightforward manner, therefore its partial correctness should be obvious.

\textbf{Naive algorithm.}

\begin{verbatim}
var: \(U : \text{set of objects}, V : \text{sequence of objects}
val: OB \rightarrow \bigcup V : V \in \Omega : V
post: RO: U, W \wedge U \text{ consistent w.r.t. val}
inv: P0 = P0.0 \& P0.1 \& \ldots \& P0.5 (see above)
action:
    U, V := \emptyset, <>; \{P0\}
    do \neg (U \supseteq W) \{ \neg (U \cup \text{rng}(V) = OB) \lor \neg (V = <>), see note \}
        \rightarrow if \neg (U \cup \text{rng}(V) = OB)
            \rightarrow Let x : x \in OB \setminus (U \cup \text{rng}(V));
            if \text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V) \rightarrow V := V \oplus <x>
                \lbrack \neg (\text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V)) \rightarrow \text{skip} \rbrack
            fi
        \lbrack \neg (V = <>)
            \rightarrow Let x : x = \text{first}(V);
            val(x) := \text{func}(x) (\text{VAL}(\text{args}(x)));
            U, V := U \cup \{x\}, \text{rest}(V)
            fi \{P0\}
    od \{R0\}
\end{verbatim}
Note: \[ U \cup \text{rng}(V) = \text{OB} \land V = <> \]
\[ \Rightarrow \{ \} \]
\[ U = \text{OB} \]
\[ \Rightarrow \{ W \subseteq \text{OB} \} \]
\[ U \supseteq W \]

This assertion guarantees that there is no danger of abortion.

This algorithm is naive in the sense that the selection of evaluable objects may be terminated as soon as \( U \cup \text{rng}(V) \supseteq W \), rather than \( U \cup \text{rng}(V) = \text{OB} \). That is, the following restriction is imposed:

**Restriction.** Selection of evaluable objects is terminated as soon as \( W \) is included in the set of all thus far selected objects.

As will be shown below, a consequence of this restriction is that, if it terminates, the improved algorithm will end up with \( V = <> \), as opposed to the naive algorithm. The enforcement of the above restriction is achieved simply by replacing

\[
\text{if} \neg (U \cup \text{rng}(V) = \text{OB}) \rightarrow \text{if} \neg (U \cup \text{rng}(V) \supseteq W). 
\]

(Note that, analogous to the note above, we still have absence of abortion:

\[
\neg (U \supseteq W) \Rightarrow \neg (U \cup \text{rng}(V) \supseteq W) \lor \neg (V = < >). 
\]

This replacement could be considered to constitute the whole transformation. However, we can go one step further; namely for the modified algorithm the additional invariant

\[
P0.6 : \neg (U \supseteq W) \lor V = < >
\]

can be shown to hold: First, initialization establishes P0.6. Next, statement \( V := V \ominus <> \) is now guarded by \( \neg (U \cup \text{rng}(V) \supseteq W) \); hence \( \neg (U \supseteq W) \) holds, which is not falsified by this statement. For the extension of \( U \) consider two cases:

Statement \( U, V := U \cup \{x\}, \text{rest}(V) \) either establishes \( V = <> \) or it does not. In the former case we are done, and in the latter case the addition of \( x \) to \( V \) apparently did not establish \( U \cup \text{rng}(V) \supseteq W \) (as objects were added to \( V \) afterwards), hence its addition to \( U \) does not establish \( U \supseteq W \) either (use the fact that only objects from \( V \) are added to \( U \) and \( V \) is first-in-first-out).

The validity of P0.6 now enables us to replace the guard \( \neg (U \supseteq W) \) by the equivalent expression \( \neg (U \cup \text{rng}(V) \supseteq W) \lor \neg (V = < >) \):

\[
\text{do } \neg (U \supseteq W) \rightarrow \text{do } \neg (U \cup \text{rng}(V) \supseteq W) \lor (V = < >)
\]

(Nb: This equivalence can be shown as follows:...
The resulting algorithm, called improved algorithm above, has the loop-structure

\[
\text{do } B0 \lor B1 \\
\quad \rightarrow \text{if } B0 \rightarrow \ldots.. \\
\quad \quad [B1 \rightarrow \ldots..] \\
\quad \quad \quad \text{fi} \\
\quad \text{od},
\]

where \( B0 \equiv \neg(U \cup \text{rng}(V) \supset W) \) and \( B1 \equiv \neg(V = <>) \). Changing this structure into the semantically equivalent one

\[
\text{do } B0 \rightarrow \ldots.. \\
\quad [B1 \rightarrow \ldots..] \\
\quad \text{od}
\]

we obtain as resulting algorithm from this transformation:

\textbf{Algorithm 0} \[\cdot\]

\textbf{var:} \hspace{1em} \( U \): set of objects, \( V \): sequence of objects

\textbf{val:} \hspace{1em} \( \text{OB} \rightarrow (\bigcup V : V \in \Omega : V) \)

\textbf{post:} \hspace{1em} \( U \supset W \land U \text{ consistent w.r.t. val} \)

\textbf{inv:} \hspace{1em} \( P0 \equiv P0.0 \land P0.1 \land \cdots \land P0.6 \)

\textbf{action:} \hspace{1em} \( U, V := \emptyset, < > ; \{ P0 \} \)

\[
\text{do } \neg(U \cup \text{rng}(V) \supset W) \\
\quad \rightarrow \text{Let } x : x \in \text{OB} \setminus (U \cup \text{rng}(V)); \\
\quad \quad \text{if } \text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V) \rightarrow V := V \oplus < > \\
\quad \quad \quad [\neg(\text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V)) \rightarrow \text{skip}] \\
\quad \quad \text{fi} \quad \{ P0 \} \\
\quad \neg(V = < >) \\
\quad \rightarrow \text{Let } x : x = \text{first} (V); \\
\quad \quad \text{val}(x) := \text{func}(x) \text{ (VAL(args}(x))) ; \\
\quad \quad U, V := U \cup \{x\}, \text{rest} (V) \quad \{ P0 \} \\
\quad \text{od} \quad \{ P0 \land U \cup \text{rng}(V) \supset W \land V = < > , \text{hence R0} \} \\
\]

\[\]

\[\]

\[\]

\[\]
As Algorithm 0 will be used as a starting-point for the derivation of all other algorithms, let us have a closer look at it.

Although its detail-list is depicted empty, the algorithm can be regarded to satisfy a number of details already, e.g.

i) "a semantic function uses all of its arguments".

This problem-detail is reflected in the definition of consistency and, as a consequence, in the fact that the evaluation of an object is allowed only if all of its arguments have previously been evaluated. Weakening this restriction, one may think of semantic functions like

\[ \lambda x_0 \cdot \lambda x_1 \cdot \lambda x_2 \cdot (\text{if } C(x_0) \text{ then } f_0(x_1) \text{ else } f_1(x_2)) \]

becoming of interest, where \( C(x_0) \) is a boolean expression in terms of \( x_0 \) and \( f_0 \) and \( f_1 \) are functions in terms of \( x_1 \) respectively \( x_2 \). Dependent on the result of \( C(x_0) \), the outcome can be determined without having evaluated either \( x_1 \) or \( x_2 \).

ii) "no object may be evaluated more than once".

This algorithm detail is hidden in invariants P0.3 and P0.5; it can be regarded as an "optimizing" detail, although efficiency is only a relative notion in the context of nondeterminism and partial correctness.

iii) "objects are selected/evaluated in isolation".

By this we mean that objects cannot be evaluated "simultaneously" in groups (using some iterative process), which would be especially interesting in case the evaluation structure \( Q \) is circular. Under certain condition it would then be possible to assign a value to mutually dependent objects in a consistency-preserving way.

The weakest condition (problem detail) that guarantees the existence of a solution to the evaluation problem probably is, that objects should not be mutually dependent via conflicting semantic rules.

Wanting to use algorithm 0 (and algorithms derived from it) with one-by-one evaluation, we shall have to settle for a stronger condition, e.g. that objects should not be mutually dependent at all, i.e. \( Q \) is noncircular.

(Namely, it is easy to see that algorithm 0 cannot possibly select, let alone evaluate, an object \( x \) that entangled in a circular dependency (i.e. \( x \rightarrow^+ x \) : The precondition to the addition of \( x \) to \( V \) implies

\[ (Ay :: y \rightarrow x \Rightarrow y \in U \cup \text{rng}(V)) \].

Continuing this argument we find that in the end \( x \in U \cup \text{rng}(V) \) should hold prior to the addition of \( x \) to \( V \), i.e. \( x \) must be selected (and may even be evaluated) before it can be selected.)

From the above it follows that Algorithm 0 cannot be the most general solution to the evaluation problem, however for the time being let us regard it as general enough (as it promises to be a starting-point for the derivation of a fair amount of evaluation algorithms).
Finally note that, even if $Q$ is noncircular, Algorithm 0 still need not terminate, viz. if the mechanism ("demon") which is responsible for making the nondeterministic choices stubbornly keeps making wrong choices. (For instance if it tries to select the same not-evaluatable object over and over again.) Thus, there are two reasons for non-terminating behaviour, and since we want to arrive at totally correct algorithms our future efforts (Chapter 3) should, among other things, be directed towards the removal of both of them.

2.2 Derived solutions

Looking at Algorithm 0, two ways to proceed immediately strike the eye: The selection and evaluation of objects could be separated in the sense that evaluation of objects may not start until sufficiently many objects have been selected first, or these activities could be merged by making sure that an object is evaluated immediately after it is selected. The corresponding transformations will be carried out in subsections 2.2.0 respectively 2.2.1. Neither of these transformations fits into the framework described in subsection 1.0.0, yet we feel sufficiently confident about the (partial) correctness of the resulting algorithms. (Of course, an independent correctness-proof could also be given, if desired.)

2.2.0 Separating the selection and evaluating of objects

Impose the following restriction:

Restriction A : Evaluation of objects is postponed until sufficiently many objects have been selected.

Observing that $U$ reflects the set of evaluated objects and $U \cup \text{rng}(V)$ equals the set of all thus far selected objects, restriction A can immediately be translated into the following invariant for the desired algorithm:

$$P0.7 : U = \emptyset \lor U \cup \text{rng}(V) \supseteq W .$$

The initialization $U, V := \emptyset$, $<>$ establishes $P0.7$. Execution of the first alternative of the loop in Algorithm 0 cannot falsify $P0.7$ as it merely brings about an extension of $\text{rng}(V)$; execution of the second alternative, however, violates assertion $U = \emptyset$ and therefore we better make sure that $U \cup \text{rng}(V) \supseteq W$ holds beforehand (this being in invariant condition to the second alternative).

The way to achieve this is by replacing
First of all check that this replacement (strengthening of the guard) is indeed allowed in the sense that the invariant-plus-negation-of-the-guards still implies postcondition RO. (The fact that it does should come as no surprise, as $U \cup \text{rng}(V) \supseteq W$ is simply the negation of the guard of the first alternative of the do-statement.)

The above replacement ensures the invariance of P0.7.

In addition, as a textual improvement we can now leave out variable $U$ from formulae "..... $\cup U$" everywhere inside the first loop-alternative; namely the validity of

$$\text{P0.7} \land \neg(U \cup \text{rng}(V) \supseteq W) \Rightarrow U = \emptyset$$

justifies the assertion $U = \emptyset$ immediately after the guard of this alternative, and this condition is not falsified thereafter. This leads us to

**Algorithm 0*. (action only)**

**action:**

\[ U, V := \emptyset, <> ; \{P0\} \]

\[ \text{do} \neg(U \cup \text{rng}(V) \supseteq W) \{ U = \emptyset \} \]

\[ \rightarrow \text{Let } x : x \in \text{OB} \setminus \text{rng}(V) ; \]

\[ \quad \text{if } \neg(\text{rng}(\text{args}(x)) \supseteq \text{rng}(V)) \rightarrow V := V \oplus < x > \]

\[ \quad \neg(\text{rng}(\text{args}(x)) \supseteq \text{rng}(V)) \rightarrow \text{skip} \]

\[ \fi \{P0\} \]

\[ \neg(V = < >) \land (U \cup \text{rng}(V) \supseteq W) \]

\[ \rightarrow \text{Let } x : x = \text{first}(V) ; \]

\[ \quad \text{val}(x) := \text{func}(x)(\text{VAL}(\text{args}(x))) ; \]

\[ \quad U, V := U \cup \{ x \}, \text{rest}(V) \{ P0 \} \]

\[ \od \{R0\} \]

Like in Section 2.1 we could be satisfied with the solution derived until now, which is undoubtedly (partially) correct and meets restriction $A$. However, the resulting program-text is a rather obscure representation of what has really happened: By the new guard, viz. $\neg(V = < >) \land (U \cup \text{rng}(V) \supseteq W)$, the condition $U \cup \text{rng}(V) \supseteq W$ holds as a precondition to the execution of the second alternative of the repetition. This, now, is a stable predicate to the algorithm and in addition it negation guards the first loop-alternative.

Thus the first alternative is never chosen after the second one has been chosen once, i.e. the alternatives are executed in strict succession.

To express this fact explicitly, the single loop of algorithm 0* can be turned into two separate loops. However, this kind of transformation is not covered by the correctness-preserving technique of Section 1.0.0; therefore we shall discuss it in a tentative way that, although probably capable of improvement, is believed to be convincing enough. Algorithm 0* is taken as a
starting-point and PO denotes PO.0 \land \cdots \land PO.7.

i) Changing the single loop into two separate loops we obtain:

\[ U, V := \emptyset, <> ; \ {P0 \land U = \emptyset} \]
\[ \text{do } \neg(U \cup \text{rng}(V) \supseteq W) \rightarrow S0 \ {P0 \land U = \emptyset} \ \text{od} ; \]
\[ \{ P0 \land U = \emptyset \land U \cup \text{rng}(V) \supseteq W \} \]
\[ \text{do } \neg(V = <> \land (U \cup \text{rng}(V) \supseteq W) \rightarrow S1 \ {P0 \land U \cup \text{rng}(V) \supseteq W} \ \text{od} \]
\[ \{ P0 \land V = <> \land U \cup \text{rng}(V) \supseteq W\} \text{, hence R0} \]

(here, and in the sequel, S0 and S1 stand for the statement-sequences inside the alternatives; they are not subject to changes.) Justification of assertions: As PO was invariant to both loop-alternatives in Algorithm 0*, and executing the alternatives in strict succession was already among the possibilities there, PO holds as indicated. The assertions above also reflect the fact that \( U = \emptyset \) holds as an invariant to the first loop and \( U \cup \text{rng}(V) \supseteq W \) does to the second one (as can be easily checked).

ii) Due to the invariance of \( U = \emptyset \) respectively \( U \cup \text{rng}(V) \supseteq W \) for the first and second loop, each one of the guards may now be simplified:

\[ U, V := \emptyset, <> ; \ {P0 \land U = \emptyset} \]
\[ \text{do } \neg(\text{rng}(V) \supseteq W) \rightarrow S0 \ {P0 \land U = \emptyset} \ \text{od} ; \]
\[ \{ P0 \land U = \emptyset \land U \cup \text{rng}(V) \supseteq W \} \]
\[ \text{do } \neg(V = <> \rightarrow S1 \ {P0 \land U \cup \text{rng}(V) \supseteq W} \ \text{od} \]
\[ \{ P0 \land V = <> \land U \cup \text{rng}(V) \supseteq W\} \text{, hence R0 } \]

iii) Having removed \( U \) from the first loop entirely, its initialization may be postponed (so as to get a more complete separation of concerns). This of course restricts the number of invariants that apply to the first loop (as variable \( U \) has no meaning yet). We present the following as a final solution:

Algorithm 1. [\cdot A ]

\text{var: } U : \text{set of objects} \quad V : \text{sequence of objects}
\text{val : } OB \rightarrow (\bigcup V : V \in \Omega : V)
\text{post: } R1.0 : Q1.0 \land \text{rng}(V) \supseteq W \quad \text{(first loop)}
R1.1 : U \supseteq W \land U \text{ consistent w.r.t. val} \quad \text{(second loop)}
\text{inv: } Q1 \equiv Q1.0 \land Q1.1 \land Q1.2 \land \text{where }
\begin{align*}
Q1.0 & : \text{rng}(V) \subseteq OB \\
Q1.1 & : (A i, j : 1 \leq i < j \leq \text{length}(V) : V.i \neq V.j) \\
Q1.2 & : (A V_0, V_1, x : V = V_0 \oplus <> \oplus V_1 : \text{rng(args}(x)) \subseteq \text{rng}(V_0))
\end{align*}
\text{P1 } \equiv \text{P1.0 } \land \cdots \land \text{P1.5 } \land \text{where }
\begin{align*}
P1.0 & : U \subseteq OB \land U \text{ consistent w.r.t. val }
\end{align*}
P1.1 : \( U = \text{dom}(\text{val}) \)

P1.2 : \( (A \land V_0, V_1, x : V = V_0 \oplus \langle x \rangle \oplus V_1 : \text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V_0)) \)

P1.3 : \( U \cap \text{rng}(V) = \emptyset \)

P1.4 : \( \neg(U \supseteq W) \lor V = \langle \rangle \)

P1.5 : \( U \cup \text{rng}(V) \supseteq W \)

**action:** \( V := \langle \rangle ; \{Q1\} \)

\[ \begin{align*}
\text{do } & \neg(\text{rng}(V) \supseteq W) \\
& \rightarrow \text{Let } x : x \in \text{OB}\setminus\text{rng}(V); \\
& \quad \text{if } \text{rng}(\text{args}(x)) \subseteq \text{rng}(V) \rightarrow V := V \oplus \langle x \rangle \\
& \quad \{Q1\} \neg(\text{rng}(\text{args}(x)) \subseteq \text{rng}(V)) \rightarrow \text{skip} \\
& \quad \{Q1\} \od ; \{R1.0\} \\
U := \emptyset; \{P1 \land Q1.0 \land Q1.1\} \\
\text{do } V \neq \langle \rangle \\
& \rightarrow \text{Let } x : x = \text{first}(V); \\
& \quad \text{val}(x) := \text{func}(x)(\text{VAL}(\text{args}(x)));
& \quad U, V := U \cup \{x\}, \text{rest}(V) \{P1 \land Q1.0 \land Q1.1\} \\
\od \{P1 \land Q1.0 \land Q1.1 \land V = \langle \rangle \text{, hence } R1.1\} \\
\end{align*} \]

**Notes**

i) One may wonder whether formulae Q1.0 and Q1.1 are required as invariants for the second loop. We suggest that they are indeed, because without them the invariance of P1.3 and P1.0 (first conjunct) cannot be proved. On the other hand the fact that they are needed is not surprising at all: As, at any moment, the contents of U fully depends on the (ex-) contents of V, one may expect to need some properties of V in order to be able to prove properties about U.

ii) In view of the strong (textual) resemblance between algorithm 0 and algorithm 1 there may be a way to derive the latter algorithm from the former which is shorter than the one followed here. However, if one is satisfied with algorithm 0* as a final solution then the amount of effort need to carry out the transformation seems reasonable.

Taking an algorithm like algorithm 1 as a starting-point for further transformations typically leads to evaluation algorithms who first sort the objects of Q (using relation \( \rightarrow \) as a criterion), after which they are evaluated. Often all objects are involved in the ordering process, although this is not necessary (provided that \( W \neq \text{OB} \)): more clever algorithms will only consider objects that are really necessary for the evaluation of objects in W. (i.e. objects \( y \) such that
Note that during the ordering process only a modest amount of information of the object evaluation structure needs to be available, viz. \( \text{OB}, \text{args} \) and (possibly) \( W \); All information concerning the value of objects may be temporarily disposed of. This may be of interest in practice from the viewpoint of (space-) efficiency.

### 2.2.1 Merging the selection and evaluation of objects

Impose the following restriction on algorithm 0:

**Restriction M**: Evaluation of an object is carried out together with its selection.

Here "together with" should be understood as "during the same iteration step as". The enforcement of restriction \( M \) induces a change of the loop-structure of algorithm 0, because in this algorithm with each iteration only one of the activities (selection or evaluation) can be performed. Thus, local replacements will not be sufficient.

We present the following algorithm as a first algorithm in which the evaluation of objects keeps pace with their selection. The validity of the intermediate assertions (inserted for later use) should be obvious. \( P0 \) denotes \( P0.0 \land \cdots \land P0.6 \), as in algorithm 0.

**Algorithm 0** (action only)

**action**

\[
U, V := \emptyset, <> ; \{ P0 \land V = <> \}
\]

**do**

\[
\neg (U \cup \text{rng}(V) \supseteq W) \quad \{ P0 \land V = <> \land \neg (U \cup \text{rng}(V) \supseteq W) \}
\]

\[
\rightarrow \text{Let } x : x \in \text{OB} \setminus (U \cup \text{rng}(V)) ;
\]

\[
\text{if } \text{rng(args}(x)) \subseteq U \cup \text{rng}(V) \rightarrow V := V \oplus <x> \quad \{ P0 \land V = <> \}
\]

\[
\neg (\text{rng(args}(x)) \subseteq U \cup \text{rng}(V)) \rightarrow \text{skip} \quad \{ P0 \land V = <> \}
\]

\[
\text{fi} ; \quad \{ P0 \land (V = <> \lor V = <x>) \}
\]

\[
\text{if } \neg (V = <>), \{ P0 \land V = <x> \}
\]

\[
\rightarrow \text{val}(x) := \text{func}(x) \cdot (\text{VAL(args}(x))) ;
\]

\[
U, V := U \cup \{ x \}, \text{rest}(V)
\]

\[
\rightarrow \text{skip}
\]

\[
\text{fi} \quad \{ P0 \land V = <> \}
\]

**od**

\[
\{ P0 \land V = <> \land U \cup \text{rng}(V) \supseteq W , \text{hence } R0 \}
\]

\[
\]
Note that the above assertions express the fact that \((P_0 \land V = <>\) holds an invariant to Algorithm \(0^{**}\). In two steps Algorithm \(0^{**}\) will be smoothed considerably:

i) Using the assertions observe that the effect of the second if-statement is nil except in combination with the first alternative of the first if-statement. The following algorithm is therefore equivalent to Algorithm \(0^{**}\):

**Algorithm \(0^{***}\) (action only)**

```plaintext
action

\[ U, V := \emptyset, <> \{ P_0 \land V = <> \} \]

\[ \text{do} \neg (U \cup \text{rng}(V) \supseteq W) \{ P_0 \land V = <> \land \neg (U \cup \text{rng}(V) \supseteq W) \} \]

\[ \rightarrow \text{Let } x : x \in \text{OB} \setminus (U \cup \text{rng}(V)) ; \]

\[ \text{if } \text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V) \]

\[ \rightarrow V := V \oplus < > ; \]

\[ \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))) ; \]

\[ U, V := U \cup \{ x \} , \text{rest}(V) \]

\[ \neg (\text{rng}(\text{args}(x)) \subseteq U \cup \text{rng}(V)) \]

\[ \rightarrow \text{skip} \]

\[ \text{fi } \{ P_0 \land V = <> \} \]

\[ \text{od } \{ P_0 \land V = <> \land U \cup \text{rng}(V) \supseteq W , \text{hence } R_0 \} \]

[]

ii) Prior to any use of \(V\) in a formula "\(U \cup \text{rng}(V)\)" the assertion \(V = <>\) holds. Therefore replacement

\[ U \cup \text{rng}(V) \rightarrow U \] (four times)

is in order. Now \(V\) has turned into a redundant variable (used only in assignments to itself), hence all assignments to \(V\) may be removed:

\[ U, V := \emptyset, <> \rightarrow U := \emptyset \]

\[ V := V \oplus < > ; \]

\[ \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))) ; \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))) ; \]

\[ U, V := U \cup \{ x \} , \text{rest}(V) \quad U := U \cup \{ x \} \]

After removal of \(V\) from the algorithm, assertions and invariants containing \(V\) do no longer hold. As it turns out, they may safely be discarded: The remaining invariants, \(P_0.0\) and \(P_0.1\), (together with the negation of the guard) are still strong enough to imply \(R_0\). Final algorithm:
Algorithm 2. [· M ]

\[ \text{var: } U : \text{set of objects } , \; \text{val: OB} \rightarrow (\bigcup V : V \in \Omega : V) \]

\[ \text{post: } R2 : U \supseteq W \land U \text{ consistent w.r.t. val} \]

\[ \text{inv: } P2.0 : U \subseteq \text{OB} \land U \text{ consistent w.r.t. val} \]

\[ P2.1 : U = \text{dom(val)} \]

\[ \text{action: } U := \emptyset ; \{ P2 \} \]

\[ \text{do } \neg (U \supseteq W) \]

\[ \rightarrow \text{Let } x : x \in \text{OB} \setminus U ; \]

\[ \text{if } \text{rng(args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL(args}(x))) ; \]

\[ U := U \cup \{x\} \]

\[ \neg (\text{rng(args}(x)) \supseteq W) \rightarrow \text{skip} \]

\[ \text{fi } \{ P2 \} \]

\[ \text{od } \{ R2 \} \]

Algorithm 2 will be used as a starting-point for the derivation of the algorithms in the next chapter. Note that the actions of selecting an object (defined as: adding an object to V) and evaluating an object (transferring it from V to U) have disappeared in their original form.

For the rest of this thesis we adopt the following convention:

**Terminology - convention 2.5** If object x is chosen in statement Let \( x : x \in \text{OB} \setminus U \) (or a corresponding statement in what follows) then we say that x is visited, or that a visit is paid to x. Such a visit is either successful (upon detecting \( \text{rng(args}(x)) \subseteq U \)) or unsuccessful (otherwise). Clearly, a visit to x may be interpreted as an attempt to evaluate x, which can be either successful or unsuccessful.

**Convention 2.6.** The relation \( \rightarrow \) between objects of an evaluation structure can be visualized by means of a directed graph. In doing so we shall draw a circle with the name of the object in question in it and connect these circles by arrows according to relation \( \rightarrow \).

For instance, if \( Q \) contains objects x, y and z, and \( \text{rng(args}(x)) = \{y\}, \text{rng(args}(y)) = \{y, z\}, \text{rng(args}(z)) = \emptyset \) then we obtain

![Diagram](Note that if \( Q \) is derived from a triplet \((G, t, W)\) using \( \alpha \) as in subsection 2.0.1 (it need not be!)), then there is an immediate correspondence between graph \( \text{dig}(t, G) \) and the drawing that belongs to \( Q \). We shall not make this relation precise.)

\[ \]
3. Derivation of some demand driven algorithms

As announced before, the algorithms in this chapter all originate from algorithm 2 (Subsection 2.2.1). After introducing the notion of demand driven evaluation, the first transformation on this algorithm serves to make it fit in with this idea. (Section 3.1). In subsequent sections the resulting algorithm will then be molded into more practical (and hence: terminating) ones by adding both problem- and algorithm-details to it.

3.0 Demand driven evaluation

The demand driven evaluation method embodies a particular way to limit the freedom contained in the choice of an object to be visited (i.e. the freedom in statement \( \text{Let } x : x \in \text{OB} \setminus U \); recall that it constituted one of the causes for non-terminating behavior of algorithm 2).

Underlying the method is the observation that eventually we are only interested in evaluating the objects in \( W \). By the consistency-requirement this may require ("demand") the evaluation of objects outside \( W \) also: If \( W \) itself is not self-contained then it cannot be consistent w.r.t. any \( \text{val} \). On the other hand, (unless \( U = \text{OB} \)) possibly not all objects in \( \text{OB} \) need to be evaluated. In this sense we discern useful and useless objects for \( W \): a useful object for \( W \) is an object without whose evaluation it is impossible to determine a pair \( (U, \text{val}) \) such that

\[
U \supseteq W \land U \text{ consistent w.r.t. } \text{val}
\]

holds; otherwise, of course, an object is called useless for \( W \). The objective of any demand driven algorithm now is to visit (and hence: evaluate) all and only the objects that are useful for \( W \).

The intuitive idea of useful objects for \( W \) given above will be shown to coincide with the notion to be introduced next (in the context of the standard evaluation structure, cf. convention 2.4):

**Definition 3.0** Function \( \text{dep} : \text{dep} \in (\text{P(OB)} \rightarrow \text{P(OB)}) \) is defined by

\[
\text{dep}(V) = \{y \mid y \in \text{OB} \land (\exists x : x \in V : y \rightarrow x)\}
\]

for all \( V : V \subseteq \text{OB} \).
Property 3.1

a) dep is a closure operation on OB, i.e. for all \( V, V' : V \subseteq OB \land V' \subseteq OB \):
   
   i. \( V \subseteq dep(V) \)
   
   ii. \( dep(dep(V)) = dep(V) \)
   
   iii. \( V \subseteq V' \Rightarrow dep(V) \subseteq dep(V') \)

b) \( (A V : V \subseteq OB : dep(V) = \emptyset \Rightarrow V = \emptyset) \)

Theorem 3.2 For all \( V : V \subseteq OB \):

a) \( dep(V) \) is self-contained

b) \( V \) is self-contained \( \equiv \; V = dep(V) \)

c) \( (A V' : V' \supseteq V \land V' \) self-contained : \( V' \supseteq dep(V) ) \)

Proof (bound \( x \in OB \) will be omitted throughout):

a) \( x \in dep(V) \land y \rightarrow x \)

\[ \Rightarrow \text{(definition 3.0)} \]

\[ (Ez : z \in V : x \rightarrow^* z) \land y \rightarrow x \]

\[ \Rightarrow \text{(transitivity of \( \rightarrow \))} \]

\[ (Ez : z \in V : y \rightarrow^* z) \]

\[ = \text{(definition 3.0)} \]

\[ y \in dep(V) \]

b) \( V = dep(V) \)

\[ = \text{(part (a))} \]

\( V = dep(V) \land dep(V) \) self-contained

\[ \Rightarrow \text{(substitution)} \]

\( V \) self-contained

\[ \Rightarrow \text{(property 2.3)} \]

\( (Ay : y \in OB \land (Ex : x \in V : y \rightarrow^* x)) : y \in V) \)

\[ = \text{(definition 3.0)} \]

\( (Ay : y \in dep(V)) : y \in V) \)

\[ = \text{(definition of \( \subseteq \))} \]

\( dep(V) \subseteq V \)

\[ = \text{(property 3.1 (a)(i))} \]

\( V = dep(V) \)

c) \( V' \supseteq V \land V' \) self-contained

\[ \Rightarrow \text{(property 3.1 (a)(iii))} \]

\( dep(V') \supseteq dep(V) \land V' \) self-contained

\[ = \text{(part (b))} \]

\( dep(V') \supseteq dep(V) \land V' = dep(V') \)
Let us show (as promised) that the set $\text{dep}(W)$ equals the set of useful objects for $W$; namely

\begin{align*}
\forall U \supseteq W \land U \text{ consistent w.r.t. } \text{val} \\
\Rightarrow \quad \{\text{consistency}\} \\
U \supseteq W \land U \text{ self-contained} \\
\Rightarrow \quad \{\text{theorem 3.2 (c)}\} \\
U \supseteq \text{dep}(W)
\end{align*}

(\textit{\#}) Let the pair $(U, \text{val})$ be such that $U \supseteq W \land U \text{ consistent w.r.t. } \text{val}$. (Hence: $\text{dep}(W) \subseteq U$.) Let function $\text{val}' \in (\Omega \rightarrow (\bigcup V : V \in \Omega : V))$ be such that

$$
\begin{align*}
\text{dom}(\text{val}') &= \text{dep}(W) \\
(\forall x : x \in \text{dom}(\text{val}'): \text{val}'(x) = \text{val}(x)),
\end{align*}
$$

then we shall show that the pair $(\text{dep}(W), \text{val}')$ is also a solution, i.e. $\text{dep}(W) \supseteq W \land \text{dep}(W)$ consistent w.r.t. $\text{val}'$. Thus, a solution can be found without evaluating objects outside $\text{dep}(W)$ or, stated differently, the set of useful objects for $W$ is contained in $\text{dep}(W)$.

\textbf{Proof:}

i. $\text{dep}(W) \supseteq W$ is property 3.1. (a)(i) and consistency of $\text{dep}(W)$ w.r.t. $\text{val}'$ can be shown by

ii. self-containedness of $\text{dep}(W)$ is guaranteed by theorem 3.2 (a)

iii. $\text{dep}(W) \subseteq \text{dom}(\text{val}')$ by the definition of $\text{val}'$

iv. type- and value-conformity follow from

$$
\begin{align*}
U \text{ consistent w.r.t. } \text{val} \\
\Rightarrow \quad \{\text{consistency}\} \\
(\forall x : x \in U : \text{val}(x) \in \text{type}(x) \land \text{val}(x) = \text{func}(x) (\text{VAL}(\text{args}(x)))) \\
\Rightarrow \quad \{\text{dep}(W) \subseteq U, \text{ see above }\} \\
(\forall x : x \in \text{dep}(W) : \text{val}(x) \in \text{type}(x) \land \text{val}(x) = \text{func}(x) (\text{VAL}(\text{args}(x)))) \\
\Rightarrow \quad \{\text{dep}(W) \text{ self-contained, hence } \text{rng}(\text{args}(x)) \subseteq \text{dep}(W) \text{ for all } x \in \text{dep}(W); \text{ definition of } \text{val}'\} \\
(\forall x : x \in \text{dep}(W) : \text{val}'(x) \in \text{type}(x) \land \text{val}'(x) = \text{func}(x) (\text{VAL}'(\text{args}(x))))
\end{align*}
$$

Imposing the demand driven property on algorithm 2 is thus accomplished by requiring that only objects in $\text{dep}(W)$ be visited. Note that, due to what has been proven under (\textit{\#}), this restriction is a viable one in the sense that it does not prohibit the existence of a consistent $(U, \text{val})$-pair altogether:

If algorithm 2 had the possibility to terminate, i.e. to come up with $(U, \text{val})$ such that $U \supseteq W \land U$
consistent w.r.t. val, then so has the modified algorithm. (Be it that the latter has \( U = \text{dep}(W) \).) This is a very important property in view of what was said in Subsection 1.0.0 under "Four remarks", point (iii).

Aside remarks

i. An entirely different strategy to limit the freedom in the visiting of objects is the so-called data-driven method. With this method the objective is to visit an evaluable object (i.e. to pay a successful visit) with each step of the repetition until the desired goal, viz. the evaluation of all objects of \( W \), is met. Thus, the visiting of objects is guided by knowledge about their evalutability, rather than their usefulness for \( W \). (Underlying this method is the observation that unsuccessful visits do not get us any further in reaching the desired goal.)

We note that at our current abstract level algorithm 2 can be turned into a data driven algorithm by keeping the set, \( M \) say, of objects amenable to evaluation:

\[
M = \{ x \mid x \in \text{OB} \land \text{rng}(\text{args}(x)) \subseteq U \} \cup U,
\]

and by visiting an object from \( M \) instead of \( \text{OB} \setminus U \) with each step of the repetition. Initialization reads

\[
U, M : = \emptyset, \{ x \mid x \in \text{OB} \land \text{args}(x) = <> \}
\]

and an extension of \( U \) may cause an extension of \( M \) also.

At a more concrete level, with a number of problem details at our disposal (revealing the interdependencies between the elements of \( \text{OB} \)), the objective may be reached by making use of details when choosing an object to be visited.

In this thesis, we shall not go into data driven algorithms any further.

ii. Ideally, one would like to use a mixture of the date driven method and the demand driven one, of course: Visit only objects that are known to be evaluable and are useful for \( W \).

However, in general these two objectives are conflicting: As we shall see, for the demand driven strategy to be interesting one must start visiting objects in \( W \) and, going back on the relation \( \rightarrow \), try to arrive at evaluable objects, while one must work exactly the opposite way in case of the data driven strategy.

\[\]

3.1 Toward a basic demand driven algorithm

Using the terminology of the preceding section, the demand driven property can be translated into the following restriction (to be imposed on algorithm 2):

**Restriction D.** All and only the objects that are useful for \( W \) may be visited.
Following the lines of the correctness preserving technique of Subsection 1.0.0, an algorithm meeting the above restriction will be developed in Subsection 3.1.0.

As, however, this algorithm turns out to be of little practical importance, a "better" (although less straightforward) solution is presented in Subsection 3.1.1. The latter algorithm will then serve as a starting-point for further transformations.

3.1.0 First solution

The enforcement of restriction \( D \) requires knowledge about the usefulness of objects for \( W \). Now introduce a variable set \( W' \) of objects with interpretation:

\[ W' \text{ contains the objects that are useful for } W \text{ and that have not been evaluated yet.} \]

Clearly, \( W' \) contains exactly the objects that still may be visited in any algorithm meeting restriction \( D \). (No re-evaluation!). Therefore the restriction can be imposed by always visiting an object from \( W' \), instead of \( OB \cup U \).

The interpretation of \( W' \) leads to the following invariant for the algorithm to be derived:

\[ P2.2: W' = dep(W) \setminus U. \]

Invariance of \( P2.2 \) is established respectively preserved by making the following additions in algorithm 2:

\[
\begin{align*}
U &:= \varnothing \quad \rightarrow U, \ W' := \varnothing, \ dep(W) \\
U &:= U \cup \{x\} \quad \rightarrow U, \ W := U \cup \{x\}, \ W' \setminus \{x\}
\end{align*}
\]

The resulting algorithm (omitted here) does not meet restriction \( D \) yet, of course. Therefore perform the following replacement:

Let \( x : x \in OB \cup U \quad \rightarrow \text{ Let } x : x \in W' \)

Thus we obtain:

Algorithm 2* (action only)

action:

\[
\begin{align*}
U, \ W' &= \varnothing, \ dep(W); \\
do \rightarrow (U \supseteq W) \{W' \neq \varnothing, \text{ see note 0} \} \\
&\rightarrow \text{ Let } x : x \in W' ; \\
&\quad \text{ if } rng(args(x)) \subseteq U \rightarrow \text{ val}(x) := func(x) (VAL(args(x))); \\
&\quad U, W' := U \cup \{x\}, \ W' \setminus \{x\} \\
&\quad \{\rightarrow (rng(args(x)) \subseteq U) \rightarrow \text{ skip}\} \\
&\od
\end{align*}
\]
Note 0:
\[ (U \ni W) \land P2.2 \]
\[ \Rightarrow \{ \text{def}(W) \ni W, \text{cf. property 3.1 (i)} \} \]
\[ (U \ni \text{def}(W)) \land P2.2 \]
\[ = \{ U \ni \emptyset, \text{hence def}(W) \neq \emptyset \} \]
\[ \text{def}(W) \cup U \neq \emptyset \land P2.2 \]
\[ \Rightarrow \{ \text{substitution} \} \]
\[ W' \neq \emptyset \]

This note guarantees the legality of the let-statement.

No variables have been made redundant, so we could take the foregoing program-text (supplemented with invariant P2.0 \land P2.1 \land P2.2 and postcondition R2) as the endpoint of the transformation. Two more steps will be taken, however, to smoothen the algorithm: In the first step invariants and postcondition will be strengthened so as to reflect explicitly the impact of restriction \( D \) on the contents of \( U \), and in the second step a small replacement will be performed in the program-text:

i. It is easy to see that condition \( U \subseteq \text{def}(W) \) holds as an additional invariant to algorithm 2*.
Since \( \text{def}(W) \subseteq OB \), we can now strengthen invariant P2.0 as

\[ P2.0' : U \subseteq \text{def}(W) \land U \text{ consistent w.r.t. val} \],
and as a new postcondition

\[ R2' : U = \text{def}(W) \land U \text{ consistent w.r.t val} \]
is valid, namely:

\[ P2.0' \land U \ni W \]
\[ = \{ \} \]
\[ U \ni W \land U \subseteq \text{def}(W) \land U \text{ consistent w.r.t val} \]
\[ \Rightarrow \{ \text{theorem 3.2 (c) ; } U \subseteq \text{def}(W) \} \]
\[ U = \text{def}(W) \land U \text{ consistent w.r.t val} \]

ii. Guard \( \neg(U \ni W) \) may be replaced by the equivalent expression \( W' \neq \emptyset : \)

\[ \text{do } \neg(U \ni W) \text{ } \mapsto \text{ do } W' \neq \emptyset. \]

On account of note 0 above, the following suffices to show the equivalence:

\[ U \ni W \land P2.0' \land P2.2 \]
\[ \Rightarrow \{ \text{definition of consistency} \} \]
\[ U \ni W \land U \text{ self-contained } \land P2.2 \]
\[ \Rightarrow \{ \text{theorem 3.2 (c)} \} \]
\[ U \supseteq \text{dep}(W) \land P2.2 \]
\[ = \{ \text{definition of } \backslash \} \]
\[ \text{dep}(W) \cup U = \emptyset \land P2.2 \]
\[ \Rightarrow \{ \text{substitution } \} \]
\[ W' = \emptyset \]

All in all we obtain as a final algorithm:

Algorithm 3 [\cdot MD]

\textbf{var:} \quad U, W' : set of objects

\textbf{val:} \quad \text{OB} \rightarrow \left( \bigcup V : V \in \Omega : V \right)

\textbf{post:} \quad R3 : U = \text{dep}(W) \land U \text{ consistent w.r.t. val}

\textbf{inv:} \quad P3.0 : U \subseteq \text{dep}(W) \land U \text{ consistent w.r.t. val}

\quad P3.1 : U = \text{dom(val)}

\quad P3.2 : W' = \text{dep}(W) \cup U

\textbf{action:} \quad U, W' := \emptyset, \text{dep}(W); \quad \{ P3.0 \land P3.1 \land P3.2 \}

\begin{algorithm}
\begin{algorithmic}
\State \textbf{do}
\State \textbf{Let} \( x : x \in W' \)
\State \textbf{if} \( \text{rng} (\text{args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))) \)
\State \hspace{1em} \( U, W' := U \cup \{ x \}, W' \backslash \{ x \} \)
\State \textbf{else} \( \neg (\text{rng}(\text{args}(x)) \subseteq U) \rightarrow \text{skip} \)
\State \hspace{1em} \{ P3.0 \land P3.1 \land P3.2 \}
\State \textbf{od} \quad \{ R3 \}
\end{algorithmic}
\end{algorithm}

The snag in this algorithm, which makes it uninteresting to use as a basis for further transformations, is of course the initialization \( W' := \text{dep}(W) \). Any reasonable implementation of this statement will result in a lengthy search process on the objects of \( Q \).

Moreover, if we do have to determine \( \text{dep}(W) \) beforehand, then we may as well do it in such a way so as to obtain an evaluation order for \( \text{dep}(W) \) right away, for instance by recording the elements in a sequence, ordered via relation \( \rightarrow \).

But in that case we are really deriving algorithms subject to restriction \( A \) (cf. Section 2.2.0).

The algorithm to be presented next is much more appealing in this respect: It starts off with an easy to determine amount of information in \( W' \) and the contents of this set is gradually extended during the computation, if required.
3.1.1 Second solution

Introduce a variable set $W'$ of objects with interpretation:

All objects in $W'$ are useful for $W$ and have not been evaluated yet.

Translating this interpretation into a (desired) invariant property, we obtain

$$P2.3 : W' \subseteq \text{dep}(W) \cup U.$$

(For use in what follows we note that $P2.3$ is equivalent with $W' \subseteq \text{dep}(W) \land W' \cap U = \emptyset$.)

$P2.3$ allows a great deal of freedom in the initialization for $W'$: any subset of $\text{dep}(W)$ will do (recall that initialization of $U$ reads $U := \emptyset$).

As it turns out, $W' := W$ is an attractive and simple choice (attractive in the sense that it gives rise to sufficiently strong additional invariants).

(note: the choice $W' := \emptyset$, which is even simpler, is not viable in view of the statement Let $x : x \in W'$ - i.e. the substitute for Let $x : x \in \text{OB}\cup\mathcal{U}$ - to be encountered.)

The following assignments to $W'$ will be added to algorithm 2, so as to establish and preserve the validity of $P2.3$:

$$U := \emptyset \rightarrow U, W' := \emptyset, W$$

$$U := U \cup \{x\} \rightarrow U, W' := U \cup \{x\}, W'\setminus\{x\}$$

Our next concern is to make the resulting algorithm (omitted here) obey restriction $D$. The property that only objects useful for $W$ may be visited is achieved by replacing:

Let $x : x \in \text{OB}\setminus\mathcal{U}$ \rightarrow Let $x : x \in W'$

But how about the property that they may all be visited? $W'$ is initialized at $W$ and during the computation it only shrinks, hence this requirement is only met if $W = \text{dep}(W)$ holds (i.e. $W$ is self-contained, according to theorem 3.2 (b)). In general, however, this is not the case and hence (in order to meet the requirement) it must be possible to extend $W'$ with (fresh) useful objects now and then, of course under invariance of $P2.3$. This last stipulation means that candidates for insertion with $W'$ are objects $y$ such that $y \in \text{dep}(W) \land y \notin U$. Such objects are at our disposal with the second alternative of the if-clause:

There $x \in \text{dep}(W)$ holds, hence also $\text{rng}(\text{args}(x)) \subseteq \text{dep}(W)$ and, in addition, by the guard $\neg(\text{rng}(\text{args}(x)) \subseteq U)$ holds. The objects contained in the (non-empty) set $\text{rng}(\text{args}(x))\setminus U$ are therefore amenable for insertion in $W'$, and we replace:

$$\text{skip} \rightarrow W' := W' \cup \text{subset}(\text{rng}(\text{args}(x))\setminus U),$$

where the subset-operator is defined to deliver a non-empty subset of the set is has as its argument; it may (and will) therefore only be applied if such a non-empty subset exists.
(Note 0: it may be tempting to add the entire set \( \text{rng}(\text{args}(x)) \setminus U \) to \( W' \) right away. In doing so, however, we would exclude the derivation of interesting algorithms still to come.

Note 1: check that the above replacement suffices to make all useful objects for \( W \) visitable: initially \( W' = W \) holds, and by going back on the argument-lists (starting from objects in \( W \)) the entire set \( \text{dep}(W) \) can be reached. Whether this goal is actually achieved of course still depends on the kindness of the mechanism controlling the various non-deterministic actions (Nb.: The fact that the entire \( \text{dep}(W) \) must be visitable immediately stems from what was shown on Section 3.0: if not, then surely no consistent \((U, \text{val})\)-pair can be found.))

All in all we arrive at the following algorithm (meeting both restriction \( D \) and \( \text{P2.3} \)):

Algorithm 2** (action only)

action:

\[ U, W' := \emptyset, W ; \]

\[ \text{do } \neg(U \supseteq W) \{ W' \neq \emptyset, \text{see note 1} \} \]

\[ \quad \text{- Let } x : x \in W' ; \]

\[ \quad \text{if } \text{rng}(\text{args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))) ; \]

\[ \quad U, W := U \cup \{ x \}, W \setminus \{ x \} \]

\[ \quad \neg(\text{rng}(\text{args}(x)) \subseteq U) \rightarrow W' := W' \cup \text{subset} (\text{rng}(\text{args}(x)) \setminus U) \]

\[ \text{fi} \]

\[ \text{od} \]

\[ \]
Like in the preceding subsection, no variables have been made redundant, so the remainder of the
transformation amounts to cleaning up the algorithm:
First, replace guard \( \neg(U \supseteq W) \) by its equivalent \( W' \neq \emptyset \).
Second, invariants and postcondition will be smoothened and strengthened as follows:
Consider the alternative formulation for P2.3 (beginning of this subsection). Taking its first con­
junct together with P2.4 we obtain \( U \cup W' \subseteq \text{dep}(W) \). The latter formula will be combined with
P2.5 from now on; the resulting conjunction is denoted in shorthand notation as
\( W \subseteq U \cup W' \subseteq \text{dep}(W) \), and will -in its turn- be used to replace condition \( U \subseteq \text{OB} \) in P2.0.
All in all we arrive at the following set of invariants for our new algorithm :

\[
\begin{align*}
P4.0 : & \ W \subseteq U \cup W' \subseteq \text{dep}(W) \land U \text{ consistent w.r.t val} \\
P4.1 : & \ U = \text{dom}(\text{val}) \\
P4.2 : & \ U \cap W' = \emptyset \\
\end{align*}
\]

which (together with the negation of the guard) justify

\[
R4 : \ U = \text{dep}(W) \land U \text{ consistent w.r.t. val}
\]
as a postcondition, namely :

\[
\begin{align*}
\Rightarrow & \ \ W' = \emptyset \\
\Rightarrow & \ \ \text{(substitution)} \\
\Rightarrow & \ \ W \subseteq U \cup U \subseteq \text{dep}(W) \land U \text{ consistent w.r.t. val} \\
\Rightarrow & \ \ \text{(theorem 3.2 (c) on first and third conjunct )} \\
\Rightarrow & \ \ U \supseteq \text{dep}(W) \land U \subseteq \text{dep}(W) \land U \text{ consistent w.r.t. val} \\
\Rightarrow & \ \ \{ \} \\
\Rightarrow & \ \ U = \text{dep}(W) \land U \text{ consistent w.r.t. val}
\end{align*}
\]

Final algorithm :

**Algorithm 4 \([ \cdot MD \])**

\[
\begin{align*}
\text{var:} & \ U, W' : \text{set of objects} \\
\text{val} : & \ \text{OB} \rightarrow (\bigcup V : V \in \Omega : V) \\
\text{post:} & \ R4 : \ U = \text{dep}(W) \land U \text{ consistent w.r.t. val} \\
\text{inv:} & \ P4 \equiv P4.0 \land P4.1 \land P4.2 \ (\text{see above})
\end{align*}
\]
action:
\[ U, W' := \emptyset, W; \{P4\} \]
do \( W' \neq \emptyset \)
\[ \rightarrow \text{Let } x : x \in W' ; \]
if \( \text{rng}(\text{args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))) ; \]
\[ U, W' := U \cup \{ x \} , W' \setminus \{ x \} \]
\[ \emptyset \rightarrow (\text{rng}(\text{args}(x)) \subseteq U) \rightarrow W' := W' \cup \text{subset}(\text{rng}(\text{args}(x)) \cup U) \]
fi \{P4\}
od \{R4\}

\[
\]

Terminology-convention 3.3. If during a visit to \( x \) the second alternative of the if-clause happens to be chosen then statement \( W' := W' \cup \text{subset}(\text{rng}(\text{args}(x)) \cup U) \) is executed, and we say that the subset-operator is invoked for \( x \). Also, if an invocation of the subset-operator for \( x \) delivers (among others) object \( y \) (hence: \( y \in \text{rng}(\text{args}(x)) \cup U \)) then \( y \) is traced from \( x \) (or: \( y \) is traced starting from \( x \)). The addition "from \( x \)" intends to stress the fact that the choice of \( y \) is a consequence of its occurrence in \( \text{args}(x) \). (Note that \( y \) may occur in the argument-list of objects other than \( x \) also.)

\[
\]

3.2 Restricting the visiting of objects
The aim of this section is the imposition of

Restriction V. Objects are visited in a last-in-first-out manner (with respect to \( W' \)).

\[
\]

The reason for preferring last-in-first-out behavior to, say, first-in-first-out is obvious: If a visit to object \( x \) is paid (i.e. \( x \) is selected from \( W' \)), and it is unsuccessful, then a number of arguments of \( x \) are added to \( W' \), and the intention is of course to visit (i.e. to try and evaluate) these arguments before visiting \( x \) itself again.

A first step to the enforcement of restriction \( V \) is the introduction of a stack of objects, called \( S \), to model the contents of \( W' \) when subject to the restriction.
However, it is impossible to have a one-to-one correspondence between the objects in \( W' \) and in \( S \), because (with the current amount of nondeterminism) it may be unavoidable that \( S \) contains multiple occurrences of some object \( x \) (whereas set (!) \( W' \) doesn't). Consider for instance
Example 3.4. Let OB consist of three objects $x$, $y$ and $z$ as follows:

![Diagram]

and let $W = \{z\}$.

Following algorithm 4, where $W'$ (or $S$) behaves like a stack the contents of $S$ and $U$ may look like (after initialization and after subsequent iteration steps):

a) $S : <z> \quad U : \emptyset$

b) $<z, x, y> \quad \emptyset$

c) $<z, x, y, x> \quad \emptyset$

d) $<z, x, y> \quad \{x\}$

e) $<z, x> \quad \{x, y\}$

f) $<z> \quad \{x, y\}$

g) $<> \quad \{x, y, z\}$

Once we have situation b), the double occurrence of $x$ in situation c) is unavoidable, because otherwise object $y$ could never be evaluated (the $x$-occurrence below $y$ is unreachable and hence useless).

Due to this multiple occurrences of an object $x$ the operation "$S := \text{POP}(S)"$ would not be a correct implementation of "$W' := W \setminus \{x\}"$ and (the counterpart of) invariant 4.2 may be falsified. The way out is to represent $W'$ by $S$ according to the rule (presented as an additional invariant):

$P4.3 : W' = \text{rng}(S)\setminus U$.

Also, for use in what follows we shall maintain

$P4.4 : S = <> \lor \text{TOP}(S) \in U$.

To enforce the validity of $P4.3$ and $P4.4$ we now make the following additions and replacements in algorithm 4 (replacements are necessary to limit the freedom present in "Let $x : x \in W'$"):

i) $U, W' := \emptyset, W \rightarrow U, W', S := \emptyset, W, <>;$

$\text{for } \omega : \omega \in W \rightarrow \text{PUSH}(S, \omega) \text{ ref}$

ii) Consider Let $x : x \in W'$. In view of restriction $V$, $P4.3$ and the nature of $S$ (stack), this statement must be replaced by selection of the top-most object $x$ in $S$ such that $x \notin U$. By $P4.4$ the object $\text{TOP}(S)$ satisfies this condition (note that $S \neq <>$) and we replace

$\text{Let } x : x \in W' \rightarrow \text{Let } x : x = \text{TOP}(S)$.
iii) Statement $U, W' := U \cup \{x\}, X' \setminus \{x\}$ does not affect the validity of P4.3. However, addition of $x$ (i.e. TOP($S$)) to $U$ falsifies P4.4. The latter can be restored by

$$U, W' := U \cup \{x\}, W' \setminus \{x\} \rightarrow U, W := U \cup \{x\}, W' \setminus \{x\};$$

$$\text{do } S \neq <> \text{ and } \text{TOP}(S) \in U$$

$$\rightarrow S := \text{POP}(S)$$

iv) $W' := W' \cup \text{subset}(\text{rng}(/\text{args}(/x)/)U) \rightarrow$

Let $Y : Y = \text{subset}(\text{rng}(/\text{args}(/x)/)U);$  
$W' := W' \cup Y; \text{ for } y : y \in Y \rightarrow \text{PUSH}(S, y) \text{ rof}$

As intermediate algorithm we obtain

Algorithm 4* . (action only)

action:

$U, W', S := \emptyset, W, <>; \text{ for } \omega \in W \rightarrow \text{PUSH}(S, \omega) \text{ rof};$

$\text{do } W' \neq \emptyset \{\text{TOP}(S) \in U\}$

$\rightarrow \text{Let } x : x = \text{TOP}(S);$

$$\text{if } \text{rng}(/\text{args}(/x)/) \subseteq U \rightarrow \text{val}(x) := \text{func}(x)(\text{VAL}(/\text{args}(/x)/));$$

$U, W' := U \cup \{x\}, W' \setminus \{x\};$

$\text{do } S \neq <> \text{ and } \text{TOP}(S) \in U \rightarrow S := \text{POP}(S) \text{ od}$

$$\text{if } (\text{rng}(/\text{args}(/x)/) \subseteq U) \rightarrow \text{Let } Y : Y = \text{subset}(\text{rng}(/\text{args}(/x)/)U);$  
$W' := W' \cup Y; \text{ for } y : y \in Y \rightarrow \text{PUSH}(S, y) \text{ rof}$

$\text{fi}$

$\text{od}$


To complete the transformation step, variable $W'$ must be removed (both from program-text and invariants). The only place where $W'$ is actually used is in the guard $W' \neq \emptyset$. By P4.3 this is equivalent to $\text{rng}(S) \cap U \neq \emptyset$, which in its turn is equivalent to $\text{rng}(S) \neq \emptyset$; namely

$$\text{rng}(S) \cup U \neq \emptyset$$

$\Rightarrow$  

{definition of $\cap$}

$$\text{rng}(S) \neq \emptyset$$

$\Rightarrow$  

{P4.4}

$$\text{TOP}(S) \in U$$

$\Rightarrow$  

{definition of $\cap$}

$$\text{rng}(S) \cup U \neq \emptyset$$
Thus, guard $W \neq \emptyset$ can be replaced by the equivalent form $S \neq <>$, thereby making $W'$ redundant in the program-text. Also, using P4.3 the invariants can be rewritten, and we obtain as a final algorithm:

**Algorithm 5 [ ; MDV ]**

**var:** $U$ : set of objects, $S$ : stack of objects ,

val : OB $\rightarrow$ ( $\bigcup V : V \in \Omega : V$ )

**post:** R5 : $U = dep(W) \land U$ consistent w.r.t. val

**inv:** P5 = P5.0 $\land$ P5.1 $\land$ P5.2, where

$P5.0 : W \subseteq U \cup \text{rng}(S) \subseteq dep(W) \land U$ consistent w.r.t. val

$P5.1 : U = \text{dom}(\text{val})$

$P5.2 : S = <> \text{ cor } \text{TOP}(S) \in U$

**action:**

$U , S := \emptyset , <> ; \text{ for } \omega : \omega \in W \rightarrow \text{PUSH}(S, \omega) \text{ rof ; } \{P5\}$

do $S \neq <>$

$\rightarrow \text{Let } x : x = \text{TOP}(S);$  

if $\text{rng}(\text{args}(x)) \subseteq U \rightarrow \text{val}(x) : = \text{func}(x) (\text{VAL}(\text{args}(x))) ;$

$U := U \cup \{x\} ;$

do $S \neq <> \text{ cand TOP}(S) \in U$

$\rightarrow S := \text{POP}(S)$

od

$[\neg(\text{rng}(\text{args}(x)) \subseteq U) \rightarrow \text{Let } Y : Y = \text{subset } (\text{rng}(\text{args}(x)) \cup U) ;$

for $y : y \in Y \rightarrow \text{PUSH}(S, y) \text{ rof}$

fi $\{P5\}$

od $\{R5\}$

**Notes**

i) Observe that P5.0 contains expression $U \cup \text{rng}(S)$ instead of the equivalent form $U \cup (\text{rng}(S) \cup U)$. Also, $U \cap (\text{rng}(S) \cup U) = \emptyset$, i.e. the counterpart of P4.2, has been omitted as an invariant.

ii) The introduction of P4.4 as an additional invariant enables the replacement of $W' \neq \emptyset$ by $S \neq <>$. Without P4.4 it is impossible to prove the equivalence $\text{rng}(S) \cup U \neq \emptyset \equiv \text{rng}(S) \neq \emptyset$ and we can only replace $W' \neq \emptyset$ by $\text{rng}(S) \cup U \neq \emptyset$. The latter expression, however, is not allowed in our algorithms as -in general- its truth cannot be decided without inspecting the contents of POP(S), while we adopt the convention that (apart from a nil-test) a stack is only provided with operations concerning its top-element.
3.3 Termination and detection of circularities

In adding (to algorithm 5) a tool for detecting and responding to circularly dependent objects we shall deviate from Jonkers' method, as the latter seems to lead to a needlessly complex treatise. In particular, the problem lies in the requirement to express relations between old and new variables in a number of invariants. These relations cannot be fixed without introducing additional variables (revealing the internal structure of \( S \)), which is dissatisfying, as it leads to a substantial increase in proof obligations. Things get even more dissatisfying when it turns out that eventually the relations are not needed to carry out the transformation step (i.e., to impose the detection method). (The latter is caused by the fact that the transformation step only amounts to adding new variables (plus assignments to them), whereas relations between old and new variables become of importance once replacements must be made (cf. Subsection 1.0.0.).)

Therefore, we shall go about by introducing new variables and carrying out the transformation step without having made relations between old and new variables explicit.

The organization of this section is as follows: In Subsection 3.3.0 a circularity detection method is applied to algorithm 5, yielding algorithm 6 which behaves exactly like algorithm 5, except that it terminates its activities in case a circular dependency is detected. It will be shown that algorithm 6 is a properly terminating algorithm and, provided \( \text{dep}(W) \) is free from circularly dependent objects, so is algorithm 5.

In Subsection 3.3.1 we present as algorithm 7 an algorithm that is equivalent to algorithm 6 in its operation, but that has smoother program text. In Subsection 3.3.2 a particular implementation of algorithm 7 is presented by changing the representation of some of its variables. The resulting algorithm is also linked to an algorithm known from literature. Finally, in Subsection 3.3.3 we shall do away with the existence of circularities by simply demanding that they do not occur. From that moment on, all algorithms derived will be totally correct.

3.3.0 A method for detecting circularities

Introduce a set of objects \( V \) and (by adding assignments to \( V \)) transform algorithm 5 into

Algorithm 5* (action only)

\[
\begin{align*}
U \cdot S &:= \emptyset \cdot < > ; \text{for } \omega : \omega \in W \rightarrow \text{PUSH}(S, \omega) \text{ end} ; \\
\text{if } S \neq < > &\rightarrow V := \{\text{TOP}(S)\} \left[ \begin{array}{c}
\text{if } S = < > \rightarrow V := \emptyset \text{ fi} ; \\
\text{do } S \neq < > \\
\rightarrow \text{Let } x : x = \text{TOP}(S) ;
\end{array} \right]
\end{align*}
\]
if \( \text{rng}(\text{args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))) ; \)
\[ U, V := U \cup \{x\}, V\setminus \{x\} ; \]
do \( S \neq < > \text{ cand } \text{TOP}(S) \in U \)
\[ \rightarrow := \text{POP}(S) \]
od ;
if \( S \neq < > \rightarrow V := V \cup \{\text{TOP}(S)\} \)
\[ \text{skip} \]
fi
\[ \neg (\text{rng}(\text{args}(x)) \subseteq U) \rightarrow \text{Let } Y : Y = \text{subset}(\text{rng}(\text{args}(x)) \setminus U) ; \]
for \( y : y \in Y \rightarrow \text{PUSH}(S, y) \text{ ref} ; \)
\[ V := V \cup \{\text{TOP}(S)\} \]
fi

(As announced before, we choose not to make the relation between \( V \) and old variables explicit at this point. However, for the sake of completeness and to reveal the heuristics that led us to introduce the particular set \( V \), we shall present the relations we have in mind at the end of this subsection.)

It should be apparent that for algorithm \( 5^* \) the following invariant properties hold:

P5.3 : \( U \cap V = \emptyset \)

P5.4 : \( S = < > \text{ cor } \text{TOP}(S) \in V \)

(Note that P5.3 and P5.4 together imply P5.2.)

In addition, we have (not only as a loop invariant, but everywhere):

P5.5 : \( (A y : y \in \text{OB} : y \notin V \lor \)
\[ (E i : 1 \leq i \leq \text{length}(S) : \)
\[ S \cdot i = y \land (A j : i < j \leq \text{length}(S) : S \cdot j \rightarrow y) \)
\]

P5.5 states that for each object \( y \) in \( V \), \( y \) appears at a certain position in \( S \) and such, that it depends on all objects that occur at a higher position in \( S \).

The validity of P5.5 will be shown as follows:

i) The addition of \( \text{TOP}(S) \) to \( V \) (three times) does not affect P5.5 as
\[ S \cdot \text{length}(S) = \text{TOP}(S) \land (A j : \text{length}(S) < j \leq \text{length}(S) : S \cdot j \rightarrow \text{TOP}(S)) \]
holds.

ii) Concurrent assignment \( U, V := U \cup \{x\}, V\setminus \{x\} \) establishes \( x \notin V \).
iii) Popping an object, say \( r \), during the inner do-loop is harmless; For \( r \) itself \( r \notin V \) holds as a precondition (use P5.3), and for an arbitrary other object \( y \)

- either \( y \notin V \) holds, which is not falsified
- or \( S \cdot k = y \land (A_j : k < j \leq \text{length}(S) : S \cdot j \rightarrow y) \) holds for some \( k : 1 \leq k < \text{length}(S) \), which is not falsified either: the \( k \)-th element is not popped as \( k < \text{length}(S) \) and the domain of the universal quantification only shrinks.

iv) Consider the for-statement. Prior to its execution \( x = \text{TOP}(S) \) holds, say \( x = S \cdot I \), where \( I = \text{length}(S) \). All objects \( y \) added to \( S \) during the for-statement are arguments of \( x \), hence as far as \( x \) is concerned

\[
S \cdot I = x \land (A_j : I < j \leq I' : S \cdot j \rightarrow x)
\]

holds afterwards, where \( I' \) is the new value of \( \text{length}(S) \).

For an arbitrary other object \( y \):

- either \( y \notin V \), which is not falsified
- or \( S \cdot k = y \land (A_j : k < j \leq I : S \cdot j \rightarrow y) \) holds beforehand for a certain \( k : 1 \leq k < I \), so that especially \( x \rightarrow y \) (as \( S \cdot I = x \)) holds beforehand.

Transitivity of \( \rightarrow \) then yields that afterwards

\[
S \cdot k = y \land (A_j : k < j \leq I' : S \cdot j \rightarrow y)
\]

holds.

P5.5 enables us to detect the presence of a circularity in \( \text{rng}(S) \) and hence in \( \text{dep}(W) \): If during the for-statement an object \( y \) is pushed on \( S \) for which \( y \in V \) holds, then P5.5 justifies the conclusion \( y \rightarrow y \). Such a detection will be signalled via a new boolean variable \( \text{circ} \); This variable will then be used to terminate the activities of the algorithm in a "standard" manner:

Algorithm 6. (action only)

Action:

\[
\begin{align*}
&U, S, \text{circ} := \emptyset, <>, \text{false} ; \text{for } \omega : \omega \in W \rightarrow \text{PUSH}(S, \omega) \text{ ref} ; \\
&\text{if } S \neq <> \rightarrow V := \{\text{TOP}(S)\} \quad [S = <> \rightarrow V := \emptyset \text{ fi} ; \\
&\text{do } S \neq <> \land \neg \text{circ} \rightarrow \text{Let } x : x = \text{TOP}(S) ; \\
&\quad \text{if } \text{rng}(\text{args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x)(\text{VAL}(\text{args}(x))) ; \\
&\quad U, V := U \cup \{x\}, V \setminus \{x\} ; \\
&\quad \text{do } S \neq <> \land \text{cand TOP}(S) \in U \rightarrow S := \text{POP}(S) \od ; \\
&\quad \text{if } S \neq <> \rightarrow V := V \cup \{\text{TOP}(S)\} \\
&\quad \quad \quad [S = <> \rightarrow \text{skip} \\
&\quad \text{fi}
\end{align*}
\]
\[ \neg (\text{rng}(\text{args}(x)) \subseteq U) \rightarrow \text{Let } Y : Y = \text{subset } (\text{rng}(\text{args}(x)) \cup U); \]

\[ \text{for } y : y \in Y \]

\[ \rightarrow \text{PUSH}(S, y); \]

\[ \text{if } y \notin V \rightarrow \text{skip} \]

\[ \text{else } y \in V \rightarrow \text{circ} := \text{true} \]

\[ \text{end}; \]

\[ V := V \cup \{\text{TOP}(S)\} \]

Claim: Algorithm 6 terminates and the number of iteration steps is bounded by \(2 \ast \#(\text{dep}(W)) + 1\).

Proof: We shall show that each iteration step (possibly except for the last one) causes an increase of the number \(#(U \cup V) + #(U)\). As both \(U\) and \(V\) are bound by \(\text{dep}(W)\) (and hence: \(#(U \cup V)\) as well as \(#(U)\) is at most \(#(\text{dep}(W))\)), this proves the claim. Here we go:

With an execution of the first alternative of the main if-clause an object \(x, x \in V \land x \notin U,\) (on account of P5.4 \& P5.2) is transferred from \(V\) to \(U\), which brings about an increase of \(#(U)\) by one while \(#(U \cup V)\) remains unchanged. Then, the (current) object \(\text{TOP}(S)\) may be added to \(V\), where \(\text{TOP}(S) \notin U\) due to the guard of the preceding do-loop. This either causes \(#(U \cup V)\) to increase by one (viz. if \(\text{TOP}(S) \notin V\) holds prior to execution) or causes no change whatsoever (otherwise).

As for an execution of the second alternative of the if-clause distinguish between two cases:

i) None of the objects pushed on \(S\) during the for-loop is an element of \(V\) (i.e. of the small if-clause within the for-loop always the first alternative is chosen). In that case an object \(y, y \notin (U \cup V),\) is added to \(V\) so that \(#(U \cup V)\) increases by one while \(#(U)\) remains unchanged. (The object \(y\) in question is the last one pushed on \(S\) during the for-loop; Note that it is well defined as the subset-operator does not deliver the empty set.)

ii) At least one of the objects pushed on \(S\) during the for-loop is an element of \(V\). Then \(\text{circ}\) is given the value true and the iteration step concerned at the same time is the last step.

[]

Thanks to the claim the following conclusions can be drawn:

i) In case of a circular dependency within \(\text{dep}(W)\) no regular termination (i.e. due to \(S = <>\)) is possible and hence termination is caused by the true-setting of \(\text{circ}\). From this we infer that the detection method introduced here is sufficiently powerful to discover any circularity. As, conversely, variable \(\text{circ}\) does not receive the value true on wrong grounds (no "false alarm") we conclude that the cases in which \(\text{circ}\) causes termination and in which a circularity occurs within \(\text{dep}(W)\) coincide.
ii) Assume given the fact that no circularity occurs within \( \text{dep}(W) \). On account of (i) variable \( \text{circ} \) then plays no role for the termination of algorithm 6. In that case algorithm 6 and algorithm 5 are equivalent in their operation and the conclusion is justified that -in absence of circularities- algorithm 5 is a totally correct algorithm. (Note that its partial correctness is guaranteed by construction.)

iii) Algorithm 6 meets a somewhat different specification than preceding algorithms, namely its postcondition reads

\[
\text{circ} = (\exists x : x \in \text{dep}(W) : x \rightarrow x) \\
\land \ (\text{circ} \lor (U = \text{dep}(W) \land U \text{ consistent w.r.t. val})).
\]

The formulation of algorithm 6 concludes the transformation step. Let us finally make the relation between \( V \) and \( S \) explicit, thereby revealing the heuristics that led to \( V \)'s introduction. The idea is based on the observation that -in algorithm 5 -all objects pushed on \( S \) during (an execution of) the second alternative of the if-clause are arguments of the object \( x \) immediately below them on \( S \) (and, in addition, \( x \) has not been evaluated yet). Thus it appears that \( S \) can be regarded to consist of a number of sub-sequences ("sub-stacks") \( S_0, S_1, \ldots, S_{n-1} \) \((n \geq 0)\) such that

(i) \( (Ai : 1 \leq i < n : (Ay : y \in \text{rng}(S_i) : y \rightarrow \text{TOP}(S_{i-1}))) \)

(ii) \( (Ai : 0 \leq i < n : \text{TOP}(S_i) \notin U) \)

(where, rather than following from the above reasoning, \( \text{TOP}(S_{n-1}) \notin U \) is implied by P5.2 if \( S \neq <> \), i.e. \( n > 0 \)).

Using the transitivity of \( \rightarrow \),

\[
(Ai,j : 0 \leq i < j < n : (Ay : y \in \text{rng}(S_j) : y \rightarrow \text{TOP}(S_j)))
\]

can immediately be derived from (i), from which it follows that the set \( \{\text{TOP}(S_i) | 0 \leq i < n-1\} \) is of interest for the detection of circularities (in the sense explained immediately before algorithm 6).

Now what we have done is keeping \( V \) such that \( V = \{\text{TOP}(S_i) | 0 \leq i < n\} \) holds. (The element \( \text{TOP}(S_{n-1}) \) is included in \( V \) for "technical" reasons; see statement (ii) at the end of Subsection 3.3.1.) Somewhat more detailed, the following invariant property holds for algorithm 5* :

\[
P : (\forall n, S_0, S_1, \ldots, S_{n-1} : n \geq 0 \land S = S_0 \oplus S_1 \oplus \cdots \oplus S_{n-1} : \\
n = 0 \lor (n > 0 \land \text{rng}(S_0) \subseteq W) \\
\land (Ai : 0 \leq i < n : S_i \neq <> ) \\
\land (Ai : 0 \leq i < n : \text{TOP}(S_i) \notin U) \\
\land (Ai : 1 \leq i < n : (Ay : y \in \text{rng}(S_i) : y \rightarrow \text{TOP}(S_{i-1}))) \\
\land V = \{\text{TOP}(S_i) | 0 \leq i < n\}
\)

which can be proved by actually adding assignments to \( n \) and the \( S_i \).
Note: In a Jonkers-like transformation, variables $V$, $n$ and the $S_i$ should be introduced explicitly, and assignments to them should be inserted in algorithm 5, after which an invariant similar to $P$ (namely: without quantification over $n$ and the $S_i$) can be proved to hold. One could then for instance proceed by introducing circ and deriving an algorithm that is equal to our algorithm 6, except for assignments to $n$ and the $S_i$. Finally, observing that $n$ and the $S_i$ act as redundant variables (i.e. are used only in assignments to themselves), they can be removed from the program-text again, yielding our algorithm 6 (with invariant $P$, however).

To us, this seems too laborious.

3.3.1 An alternative for algorithm 6

By a transformation based on "operational grounds", algorithm 6 will be converted into a more elegant, though equivalent, algorithm as follows:

At initialization and concluding each alternative of the main if-clause there is a statement adding the object currently on top of $S$ to $V$. As the object which is on top of $S$ at the end of a step (eq. the initialization) is still on top of $S$ at the beginning of the next step (eq. first step) we suggest that the statements referred to above can be combined and moved to the beginning of the loop body, yielding:

**Algorithm 7** (action only)

```plaintext
action:
    $U, S, V, \text{circ} := \emptyset, <>, \emptyset, \text{false}$ ; for $\omega : \omega \in W \rightarrow \text{PUSH}(\omega, \text{rof})$
    do $S \neq <> \land \neg \text{circ}$
        $\rightarrow \text{Let } x : x = \text{TOP}(S) ; V := V \cup \{x\}$
        if $\text{rng}(\text{args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x)(\text{VAL}(\text{args}(x)))$
            $U, V := U \cup \{x\}, V \setminus \{x\}$
            do $S \neq <> \land \text{TOP}(S) \in U$
                $\rightarrow S := \text{POP}(S)$
            od
            $\neg(\text{rng}(\text{args}(x)) \subseteq U) \rightarrow \text{Let } Y : Y = \text{subset}(\text{rng}(\text{args}(x)) \setminus U)$
            for $y : y \in Y$
                $\rightarrow \text{PUSH}(S, y)$
                if $y \notin V \rightarrow \text{skip}$
                $\neg y \in V \rightarrow \text{circ} := \text{true}$
            fi
        fi
    fi
od
```

[]
Without discussing them in detail, we make the following statements:

i) In algorithm 7, $V$ has the (operational) interpretation:

$V$ contains the objects that have been visited and that are not yet evaluated.

ii) It is fairly easy to see that P5.3 and P5.5 still hold for algorithm 7. P5.4, however, does not longer hold: It cannot be asserted whether $\text{TOP}(S)$ is in $V$ or not. As a consequence, $V$ cannot be linked to $S$ via a relation similar to $P$ (p. 55); We can only get as far as

$$\{\text{TOP}(S_i) \mid 0 \leq i < n-1\} \subseteq V \subseteq \{\text{TOP}(S_i) \mid 0 \leq i < n\}.$$ 

iii) For algorithm 7, the termination argument (and together with that the correctness proof for the circularity detection method) is less elegant than the one for algorithm 6 (cf. the claim on p. 54). Actually, this was the reason for deriving algorithm 6 first (instead of deriving algorithm 7 right away).

3.3.2 An implementation of algorithm 7

In algorithm 7 each object belongs to exactly one of the following sets:

- the set of evaluated objects $U$
- the set of visited but not-yet-evaluated objects $V$
- the set of objects that are not visited (and hence not evaluated) $\text{OB} \setminus (U \cup V)$.

Therefore $U$ and $V$ can be implemented by a variable, three-valued function $\text{mark} : \text{OB} \to \{e, v, f\}$ such that

$$\text{mark}(x) = \begin{cases} 
  e & \text{if } x \in U \\
  v & \text{if } x \in V \\
  f & \text{if } x \in \text{OB} \setminus (U \cup V)
\end{cases}$$

for all $x : x \in \text{OB}$. Assignments to $\text{mark}$ can be inserted in algorithm 7, after which $U$ and $V$ may be removed. This transformation step is regarded trivial, so let us immediately present the resulting algorithm:
Algorithm 8 (action only)

action:
\[ S, \text{circ} := <> \text{, false} ; \text{for } \omega : \omega \in W \rightarrow \text{PUSH}(S, \omega) \text{ rof} ; \]
\[ \text{for } x : x \in \text{OB} \rightarrow \text{mark}(x) := f \text{ rof} ; \]
\[ \text{do } S \neq <> \text{ and } \neg \text{circ} \]
\[ \rightarrow \text{Let } x : x = \text{TOP}(S) ; \]
\[ \text{mark}(x) := v ; \]
\[ \text{if } (Ai : 1 \leq i \leq \text{length}(\text{args}(x)) : \text{mark}(\text{args}(x) \cdot i) = e) \]
\[ \rightarrow \text{val}(x) := \text{func}(x)(\text{VAL}(\text{args}(x))) ; \]
\[ \text{mark}(x) := e ; \]
\[ \text{do } S \neq <> \text{ and } \text{cand} \text{ mark}(\text{TOP}(S)) = e \rightarrow S := \text{POP}(S) \text{ od} \]
\[ \text{[] (Ai : 1 \leq i \leq \text{length}(\text{args}(x)) : \text{mark}(\text{args}(x) \cdot i) = e) } \]
\[ \rightarrow \text{Let } Y : Y = \text{subset}(\text{rng}(\text{args}(x)) \setminus \{ y \mid \text{mark}(y) = e \}) ; \]
\[ \text{for } y : y \in Y \rightarrow \text{PUSH}(S, y) ; \]
\[ \text{if } \text{mark}(y) \neq v \rightarrow \text{skip} \text{ [] mark}(y) = v \rightarrow \text{circ} := \text{true fi } \]
\[ \text{rof} \]
\[ \text{fi} \]
\[ \text{od} \]
\[ \text{fi} \]

If we choose in algorithm 8 for the subset-operator the identity function, i.e. impose the restriction

"During an unsuccessful visit to \( x \) all of \( x \)'s arguments that have not been evaluated are traced from \( x \).", then we essentially obtain the algorithm presented in [Jal83].

3.3.3 Excluding circularities

For all remaining algorithms we suppose the following restriction to hold true:

Restriction C. None of the objects contained in \( \text{dep}(W) \) depends on itself.

(Nb: It should be noted here that restriction C is imposed only for the sake of brevity. If desired, the circularity detection method introduced in this section could be carried along in all algorithms still to follow.)

Also, in deriving further algorithms, we start again from algorithm 5 (Section 3.2).
3.4 Restricting the size of \( W \)

Imposing the following problem detail on algorithm 5:

**Restriction E.** The set \( W \) of wanted objects contains exactly one object.

For convenience, the only object in \( W \) will be referred to by always calling it \( z \):

**Notational convention 3.5.** In all algorithms meeting restriction E, the letter \( z \) is used exclusively to denote the object in \( W \).

The restriction imposed here is not as drastic as it may seem: Using a small (and obviously correctness preserving) transformation we shall show how all of our future algorithms meeting restriction E can be made to work equally well if this restriction is dropped; see Section 3.8. In fact, the resulting algorithm then reflects the way in which most demand driven algorithms (known from literature) deal with a set \( W \) containing more than one element.

(In this context it should be noted that the advantage of using demand driven algorithms, viz. that no objects except useful ones (for \( W \)) are evaluated, gradually disappears if \( W \) grows: one may expect the number of useful objects to grow also. Thus, demand driven algorithms are of particular interest if the size of \( W \) is limited to a few objects.)

Enforcing restriction E only amounts to a change of notation, and we obtain (from algorithm 5):

**Algorithm 9 \([CE\cdot MDV]\)**

\[
\begin{aligned}
\text{var:} & \quad U : \text{set of objects, } S : \text{stack of objects,} \\
& \quad \text{val : } \text{OB} \xrightarrow{p} (\bigcup V : V \in \Omega : V) \\
\text{post:} & \quad R9 : U = \text{dep}(\{z\}) \land U \text{ consistent w.r.t. val} \\
\text{inv:} & \quad P9 = P9.0 \land P9.1 \land P9.2, \text{ where} \\
& \quad P9.0 : \{z\} \subseteq U \cup \text{rng}(S) \subseteq \text{dep}(\{z\}) \land U \text{ consistent w.r.t. val} \\
& \quad P9.1 : U = \text{dom}(\text{val}) \\
& \quad P9.2 : S = <> \text{ cor } \text{TOP}(S) \notin U \\
\text{action:} \\
& \quad U, S := \emptyset, <> ; \text{PUSH}(S, z) ; [P9] \\
& \quad \text{do } S \neq <> \\
& \quad \quad \rightarrow \text{Let } x : x = \text{TOP}(S); \\
\end{aligned}
\]
3.5 Restricting the tracing of objects

Impose the following restriction:

Restriction T. With an unsuccessful visit to an object exactly one of its arguments it traced.

This restriction does not require the introduction of new variables; it is accomplished simply by replacing:

\[
\text{Let } Y : Y = \text{subset } (\text{rng(args(x)} \cap U) ;
\text{for } y : y \in Y \rightarrow \text{PUSH}(S, y) \text{ rof}
\]

Due to this replacement, three additional invariants can shown to hold. Firstly:

\[
P9.3 : (A_i : 1 \leq i < \text{length}(S) : S \cdot (i + 1) \rightarrow S \cdot i),
\]

the invariance of which is obvious. (Note however that, as for the initialization to establish P9.3, we make use of detail E (Section 3.4.).)

Secondly:

\[
P9.4 : (A_i, j : 1 \leq i < j \leq \text{length}(S) : S \cdot i \neq S \cdot j),
\]

which follows immediately from P9.3 and the fact that \(\text{dep}([z])\) is acyclic (i.e. \((A_x : x \in \text{dep}([z])) : \neg (x \rightarrow x))\), namely:
(Ai : 1 ≤ i < length(S) : S·(i+1) → S·i)
⇒ [transitivity of → ]
(Ai,j : 1 ≤ i < j ≤ length(S) : S·j →+ S·i)
⇒ [ rng(S) ⊆ dep( {z} ) ∧ dep( {z} ) acyclic ]
(Ai,j : 1 ≤ i < j ≤ length(S) : S·i ≠ S·j).

And thirdly:

P9.5 : rng(S) ∩ U = ∅,
namely: If S = <> then the validity of P9.5 is obvious; otherwise we derive from P9.3 and P9.2 :
(Ai : 1 ≤ i < length(S) : S·(i+1) → S·i) ∧ TOP(S) ∈ U
⇒ [transitivity of → ]
(Ai,j : 1 ≤ i < j ≤ length(S) : S·j →+ S·i) ∧ TOP(S) ∈ U
⇒ [ fix j at length(S) ]
(Ai : 1 ≤ i < length(S) : TOP(S) →+ S·i) ∧ TOP(S) ∈ U
⇒ { U self-contained }
(Ai : 1 ≤ i < length(S) : S·i ∈ U)
=
{ }

rng(S) ∩ U = ∅.

On account of P9.5 invariant P9.2 becomes trivially satisfied. For the final algorithm invariants P9.3 and P9.4 will be omitted, and, finally, P9.4 and P9.5 together enable us to make an additional replacement. Therefore observe that P9.4 implies TOP(S) ∈ rng(POP(S)) and consider statement "U := U ∪ {x}" (where x = TOP(S)) with annotation as follows:

{ rng(S) ∩ U = ∅ ∧ TOP(S) ∈ rng(POP(S)) }
{ rng(POP(S)) ∩ (U ∪ {TOP(S)}) = ∅ }
U := U ∪ {x}
{ rng(POP(S)) ∩ U = ∅ }

The last assertion acts as a precondition to the inner do-loop "do S <> cand TOP(S) ∈ U → S := POP(S) od", hence replacement

do S <> cand TOP(S) ∈ U → S := POP(S) od → S := POP(S)

(i.e.: execution of the loop amount to one step only) is in order.

Thus, we obtain:
Algorithm 10. [CE·MDVT]

\[ \text{var: } U : \text{set of objects}, S : \text{stack of objects} \]
\[ \text{val: } OB \xrightarrow{P} (\bigcup V : V \in \Omega : V) \]

\[ \text{post: } R10 : U = \text{dep}(\{z\}) \land U \text{ consistent w.r.t. val} \]

\[ \text{inv: } \text{P10} = \text{P10.0} \land \text{P10.1} \land \text{P10.2}, \text{where} \]
\[ \text{P10.0} : \{z\} \subseteq U \cup \text{rng}(S) \subseteq \text{dep}(\{z\}) \land U \text{ consistent w.r.t. val} \]
\[ \text{P10.1} : U = \text{dom}(\text{val}) \]
\[ \text{P10.2} : \text{rng}(S) \cap U = \emptyset \]

\[ \text{action:} \]
\[ U, S := \emptyset, <>; \text{PUSH}(S, z); \{\text{P10}\} \]
\[ \text{do } S \neq <> \]
\[ \rightarrow \text{Let } x : x = \text{TOP}(S); \]
\[ \text{if } \text{rng}(\text{args}(x)) \subseteq U \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL}(\text{args}(x))); \]
\[ U, S := U \cup \{x\}, \text{POP}(S) \]
\[ \rightarrow \neg(\text{rng}(\text{args}(x)) \subseteq U) \rightarrow \text{Let } y : y \in \text{rng}(\text{args}(x)) \setminus U; \]
\[ \text{PUSH}(S, y) \]
\[ \text{fi } \{\text{P10}\} \]
\[ \text{od } \{R10\} \]

The reader is invited to compare this algorithm with algorithm 4 (Subsection 3.1.1.).

3.6 Removing the final nondeterminism

In this section a number of restrictions on algorithm 10 are imposed simultaneously. The reason for this is twofold:

i) the restrictions concerned are closely related.

ii) together the restrictions give rise to a compact algorithm that can be used as a basis for further transformations.

The restrictions have in common the fact that they all deal with actions that are only implicitly visible in algorithm 10; in fact they intend to make these actions more explicit.

Consider the guards of the if-clause:

\[ \text{rng}(\text{args}(x)) \subseteq U \text{ resp. } \neg(\text{rng}(\text{args}(x)) \subseteq U). \]

To determine which one of these conditions holds true (i.e. whether the visit to \( x \) will be successful respectively unsuccessful) arguments of \( x \) may be checked on membership of \( U \) one by one.
until either all of them are checked (successful visit) or an object is encountered that is not contained in $U$ (unsuccessful visit). The following restriction makes this process explicit:

Restriction $H_0$: Determining whether a visit to $x$ is successful respectively unsuccessful is accomplished by checking the arguments of $x$ one by one until either all of them are checked or one is encountered that is not yet contained in $U$.

Suppose the visit is unsuccessful. Then an object must be searched for to be traced in statement "Let $y : y \in \text{rng} \left( \text{args} (x) \right) \setminus U". The following restriction gives a particular implementation for this statement:

Restriction $H_1$: In the case of an unsuccessful visit to $x$, the object to be traced is the argument of $x$ that was found not to be in $U$ during the inspection process specified in restriction $H_0$.

Restriction $H_1$ links the inspection process and the tracing process. However, due to the non-determinism still contained in the order of inspecting objects, the combined inspector/tracer is still nondeterministic. This is resolved by imposing

Restriction $H_2$. Arguments of $x$ are traced in the order of their appearance in the sequence $\text{args} (x)$.

We shall now present as algorithm $10^*$ the algorithm derived from algorithm 10 by imposing restrictions $H_0$, $H_1$ and $H_2$ simultaneously. This algorithm requires only the introduction of a local integer variable $i$.

Convention 3.6. For the remainder of this thesis we draw the convention that a constant function $m : m \in (\text{OB} \rightarrow \mathbb{N})$ exists such, that for all $x : x \in \text{OB}$ $m (x) = \text{length} (\text{args} (x))$. This to save ourselves a lot of writing.

It is not difficult to see that, apart from P10.0 through P10.2, the following holds invariantly true for algorithm $10^*$:

For all $n, y_1, y_2, \ldots, y_n$ such that $n \geq 0$ and $S = <y_1, y_2, \ldots, y_n>$:

$(Ai : 1 \leq i < n) :$

$(Bj : 1 \leq j \leq m (y_i) : y_{i+1} = \text{args} (y_i) \cdot j \land (Ak : 1 \leq k < j : \text{args} (y_i) \cdot k \in U))$

which is in fact nothing more than a formalization of restriction $H_2$. 
Algorithm 10*. (action only)

action:

\[ U, S := \emptyset, <>; \text{PUSH}(S, z); \]

\[ \text{do } S \neq <> \]

\[ \rightarrow \text{Let } x : x = \text{TOP}(S); \]

\[ i := 1; \]

\[ \{ \text{invariant } Q : (Aj : 1 \leq j < i : \text{args}(x) \cdot j \in U) \land 1 \leq i \leq m(x) + 1 \} \]

\[ \text{do } i \neq m(x) + 1 \text{ cand } \text{args}(x) \cdot i \in U \rightarrow i := i + 1 \text{ od}; \]

\[ \{ (i = m(x) + 1 \text{ cor } \text{args}(x) \cdot i \notin U) \land (Aj : 1 \leq j < i : \text{args}(x) \cdot j \in U) \} \]

\[ \text{if } i = m(x) + 1 \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL(args}(x))); \]

\[ U, S := U \cup \{ x \}, \text{POP}(S) \]

\[ \{ i \neq m(x) + 1 \rightarrow \text{PUSH}(S, \text{args}(x) \cdot i) \} \]

\[ \text{fi} \]

\[ \text{od} \]

\[ \[ \]

There is a time-inefficiency in the above algorithm. Suppose that during a visit to \( x \) the object \( \text{args}(x) \cdot i0 \) is traced for some \( i0 : 1 \leq i0 \leq m(x) \). Then apparently

\[ (Aj : 1 \leq j < i0 : \text{args}(x) \cdot j \in U) \]

holds. Since "being in \( U \)" is a stable predicate for objects this condition will still hold during following visits to \( x \) (if any), i.e. the objects \{ \text{args}(x) \cdot j | 1 \leq j < i0 \} \) are no candidates for tracing during any following visit to \( x \) and don't have to be considered anymore. This is the purpose of

Restriction H3. Arguments of \( x \) that are found to be in \( U \) during some visit to \( x \) are not considered during any following visit to \( x \).

\[ \[ \]

The enforcement of restriction H3 requires an overhead of one integer variable per object. Therefore introduce function \( k : k \in (\text{OB} \to \mathbb{N}) \) with interpretation

For all objects \( x, \text{args}(x) \cdot k(x) \) it the argument of \( x \) to be considered first during the next visit to \( x \), if \( 1 \leq k(x) \leq m(x) \). If \( k(x) = m(x) + 1 \) then no argument of \( x \) will ever be considered (either because \( x \in U \) or because \( x \) has no arguments at all).

As additional invariants, according to this interpretation, we maintain :

\[ \text{P10.3 : } (Ax : x \in \text{OB} : 1 \leq k(x) \leq m(x) + 1) \]

\[ \text{P10.4 : } (Ax : x \in \text{OB} : (Ai : 1 \leq i < k(x) : \text{args}(x) \cdot i \in U)) ; \]

and it is also easy to see that the following holds :
P10.5: For all \( n, y_1, y_2, \ldots, y_n \) such that \( n \geq 0 \) and \( S = \langle y_1, y_2, \ldots, y_n \rangle \):

a) \( (Ai: 1 \leq i < n : k(y_i) \neq m(y_i) + 1) \)
b) \( (Ai: 1 \leq i < n : \text{args}(y_i) \cdot k(y_i) = y_{i+1}) \)

If we let letter \( H \) denote the conjunction of \( H_0, H_1, H_2 \) and \( H_3 \) then the resulting (and final) algorithm reads:

**Algorithm 11. [EC \cdot MDVTH]**

**var:**
- \( U \) : set of objects
- \( S \) : stack of objects
- \( k : \text{OB} \rightarrow \mathbb{N} \), \( \text{val} : \text{OB} \rightarrow (\bigcup V : V \in \Omega : V) \)

**post:**
- \( R11 : U = \text{dep} (\{z\}) \land U \text{ consistent w.r.t. val} \)

**inv:**
- \( P11.0, 11.1, 11.2 : \text{as P10.0, 10.1, 10.2} \)
- \( P11.3, 11.4, 11.5 : \text{as P10.3, 10.4, 10.5} \)

**action:**

```plaintext
U, S := \emptyset, <> ; PUSH(S, z); 
for x : x \epsilon \text{OB} \rightarrow k(x) := 1 \text{ ref} ; \{P11\} 
do S \neq <>
do Let x : x = \text{TOP}(S) ; 
do k(x) \neq m(x) + 1 \text{ cand args}(x) \cdot k(x) \epsilon U 
   \rightarrow k(x) := k(x) + 1 
od ; 
if k(x) = m(x) + 1 \rightarrow \text{val}(x) := \text{func}(x) (\text{VAL(args}(x))) ; 
   U, S := U \cup \{x\} , \text{POP}(S) 
   U, S := U \cup \{x\} , \text{POP}(S) 
   U, S := U \cup \{x\} , \text{POP}(S) 
   \{P11\} 
fi \{P11\} 
```

3.7 Changing the representation of \( S \) : the DSW-idea

The Deutsch-Schorr-Waite (DSW)-idea [S&W67] embodies a way to represent the stack \( S \) (almost) without any space-overhead. It does not bring about a reduction of nondeterminism (which would be impossible anyhow in the case of algorithm 11), and it can be used only if \( \text{dep}(\{z\}) \) contains no circularly dependent objects. The idea is based on the invariant property:
P11.5: For all \( n, y_1, \ldots, y_n \) such that \( S = <y_1, \ldots, y_n> \) and \( n \geq 0 \):

a) \( (Ai : 1 \leq i < n : k(y_i) \neq m(y_i) + 1) \)

b) \( (Ai : 1 \leq i < n : \text{args}(y_i) \cdot k(y_i) = y_{i+1}) \)

For stack \( S \) as in P11.5, property (b) will be depicted as

\[
\begin{array}{c}
\vdots \\
\downarrow \\
y_2 \\
\downarrow \\
y_1 \\
\vdots \\
\end{array}
\]

Figure 0

That is, the dotted arrow from \( y_1 \) to \( y_2 \) symbolizes the fact that \( \text{args}(y_1) \cdot k(y_1) = y_2 \), and so on.

Now the DSW-idea is, that by changing ("reversing") the contents of \( \text{args}(y_i) \cdot k(y_i) \) for \( i : 1 \leq i < n \) and introducing two variables \( p \) and \( q \) the configuration of figure 0 can be transformed without loss of information into that of

\[
\begin{array}{c}
\vdots \\
\downarrow \\
y_2 \\
\downarrow \\
y_1 \\
\vdots \\
\end{array}
\]

Figure 1

That is, \( \text{args}(y_2) \cdot k(y_2) = y_1 \), and so on. Note that the contents of \( y_n \)'s argument-list has not been changed.

Variable \( q \) refers to \( y_{n-1} \) and variable \( p \) refers to \( y_n \).

To be able to deal with the cases \( n = 0 \) and \( n = 1 \) we introduce two additional variables \( y_0 \) and \( y_{-1} \) that are both equal to nil, nil being a dummy object outside OB. For later use we also remark that \( \text{args}(y_1) \cdot k(y_1) \) equals nil whenever \( n \geq 2 \) (i.e. \( S \) contains more than one object), although this doesn't show in figure 1.

The configuration of figure 1 has the advantage over that of figure 0 that stack \( S \) has become superfluous: The operations on \( S \) can be expressed in terms of the variables \( p \) and \( q \) and the contents of \( \text{args}(y_i) \cdot k(y_i) \), \( i : 1 \leq i \leq n \).

Before the DSW-idea can be expressed formally, one problem still has to be solved. Comparing figure 1 with figure 0 one can see that the argument-values of the objects \( y_1, \ldots, y_{n-1} \) have been changed, whereas up till now we have assumed that (for an object \( x \)) \( \text{args}(x) \) is a constant sequence of objects.

Simply turning \( \text{args}(x) \) into a variable sequence is not allowed, as it may influence the evaluation process and lead to wrong results. Therefore we shall go about as follows:

Introduce an additional variable functions \( \text{args}' : \text{args}' \in (\text{OB} \rightarrow \text{OB}^*) \) and make sure that initially
(Ax: x ∈ OB: args'(x) = args(x))

holds. To express the DSW-idea (as shown in figure 1) assignments to \( p, q \) and the sequences \( args'(x) \) are then added to algorithm 11. (Which does not affect the latter's correctness.) In the resulting algorithm (algorithm 11*) it will be shown that

i) each time when (an element of) a sequence \( args(x) \) is addressed the relation \( args'(x) = args(x) \) hold true, and

ii) the final value of each sequence \( args'(x) \) equals \( args(x) \);

from which it follows that we may just as well remove \( args \) from the algorithm and use \( args' \) instead. Stated differently:

We could have saved ourselves the trouble of introducing the notion \( args' \) altogether by using \( args \) (made variable!) right away.

(Which justifies the statement made earlier, viz. that the DSW-idea embodies a way to represent \( S \) without any space-overhead.)

Then, finally algorithm 12 is obtained by actually removing \( args \) and \( S \) from algorithm 11*, which concludes the transformation-step.

**Intermediate remarks**

1) The requirement that the final value of \( args' \) should be equal to that of \( args \) -see (ii) above- stems from the fact that \( args \) appears implicitly in the postcondition, namely in the definition of consistency. A permanent change of the sequence \( args(x) \) -made variable, as was the ultimate goal- for some object \( x \) would give the statement

\[ U \text{ consistent w.r.t. } \text{val} \]

in the postcondition a different purport.

2) The attentive reader may have been wondering whether the configuration of figure 0 could also be changed into

\[
\begin{array}{cccccccc}
\circ y_1 \rightarrow \circ y_2 \rightarrow \ldots \rightarrow y_{n-1} \rightarrow \circ y_n \\
\downarrow & & \uparrow \\
p & & & &
\end{array}
\]

instead of figure 1, thereby saving variable \( q \). The reason for not doing so is manyfold:

a) Observe from Pl1.5 (a) that \( k(y_n) \) may be equal to \( m(y_n) + 1 \), in which case it is impossible to postulate that \( args(y_n) \cdot k(y_n) = y_{n-1} \), as the dotted arrow in figure 2 indicates.
b) Besides, if \( k(y_n) \) were different from \( m(y_n) + 1 \), the question remains how we could ever restore the original value of \( args(y_n) \cdot k(y_n) \), as it is apparently lost.

c) Since \( y_n = \text{TOP}(S) \) holds, the sequence \( args(y_n) \) is frequently addressed in the algorithms, and in view of requirement (i) above we better make sure that \( args(y_n) \) has its original value.

3) The application of the DSW-idea depends critically on the fact that no object in \( \text{dep}(\{z\}) \) depends on itself (hence, all objects in \( S \) are different). See what may happen otherwise; for instance consider

\[
\begin{array}{c}
\text{z} \\
\text{y} \\
\text{x}
\end{array}
\]

A possible stack-configuration (compare figure 0) is

\[
\begin{array}{c}
\text{z} \rightarrow \text{y} \rightarrow \text{x} \rightarrow \text{y} \rightarrow \text{x}
\end{array}
\]

\[
\begin{align*}
\text{args}(z) \cdot k(z) &= y \\
\text{args}(y) \cdot k(y) &= x \\
\text{args}(x) \cdot k(x) &= y
\end{align*}
\]

with corresponding configuration according to the DSW-idea

\[
\begin{array}{c}
\text{z} \leftarrow \text{y} \leftarrow \text{x} \leftarrow \text{y} \leftarrow \text{x}
\end{array}
\]

\[
\begin{array}{c}
q \\
p
\end{array}
\]

Apparently \( \text{args}(y) \cdot k(y) = x \) and \( \text{args}(y) \cdot k(y) = z \); hence \( x = z \), a contradiction.

\[
\]

Inspired by figure 1, we formulate the relations between the old variables and the new ones \((\text{args}', p, q)\) as follows:

**P11.6:** Let \( V, n, y_1, y_2, \ldots, y_n \) be such that

\[
n \geq 0, S = \langle y_1, y_2, \ldots, y_n \rangle, V = \{y_i \mid 1 \leq i \leq n\};
\]

Let \( y_0 \) and \( y_{-1} \) denote nil.

Then:

a) \( p = y_n \)

b) \( q = y_{n-1} \)

c) \( (\forall i : 1 \leq i < n : \text{args}'(y_i) \cdot k(y_i) = y_{i-1}) \)
d) \((\forall x, j \in V \land 1 \leq j \leq m(x)):
\quad j \neq k(x) \Rightarrow \text{args}'(x) \cdot j = \text{args}(x) \cdot j\)

e) \((\exists x \in \text{Ob} \setminus V : \text{args}'(x) = \text{args}(x))\)

(Note: Due to the absence of circularities, the implication in PI 1.6 (d) may be replaced by an equivalence. To see this, use PI 1.5 (b), PI 1.6 (c) and the fact that all elements of \(S\) are different.)

Assignments to \(\text{args}'\), \(p\) and \(q\) are now added in algorithm 11 in such a way that PI 1.6 is satisfied:

i) Initialization:

\[
U, S := \emptyset, < > ; \text{PUSH}(S, z) \Rightarrow
U, S, p, q := \emptyset, < >, z, \text{nil} ; \text{PUSH}(S, z);
\]

\[
\text{for } x : x \in \text{Ob} \rightarrow \text{args}'(x) := \text{args}(x)\]

ii) The inner do-loop does not affect PI 1.6 as its precondition implies \(x = p\), hence \(x \notin V\).

iii) Consider \(U, S := U \cup \{x\}, \text{POP}(S)\) and observe that PI 1.5 and PI 1.6 hold as a precondition to it. Also \(S \neq < >\), hence \(p \neq \text{nil}\). Now distinguish between the case that \(S\) contains only one object, i.e. \(q = \text{nil}\), and the case that \(S\) contains more than one object \(q \neq \text{nil}\).

In the former case \(V\) is empty and will remain so, hence parts (c), (d) and (e) are not affected, and neither is (b) as \(y_0 = y_{-1} = \text{nil}\). Part (a) is restored by assigning \(\text{nil}\) to \(p\). In the latter case, \(q\) is removed from \(V\). Hence \(V\) shrinks and parts (c), (d) remain valid. Part (e) can be restored by making \(\text{args}'(q)\) equal to \(\text{args}(q)\). Using (d) this is accomplished already by assigning \(\text{args}(q) \cdot k(q)\) to \(\text{args}'(q) \cdot k(q)\), where \(\text{args}(q) \cdot k(q) = p\) according to PI 1.5 (b) and PI 1.6 (a), (b). Finally parts (a) and (b) are restored by assigning \(q\) to \(p\) respectively assigning to \(q\) the object "below" \(q\) (which may be \(\text{nil}\)), i.e. \(\text{args}'(q) \cdot k(q)\) according to PI 1.6 (c). All in all we arrive at the addition

\[
U, S := U \cup \{x\}, \text{POP}(S) \Rightarrow
U, S := U \cup \{x\}, \text{POP}(S);
\]

\[
\text{if } q = \text{nil} \rightarrow p := \text{nil}
\]

\[
[ ] q \neq \text{nil} \rightarrow p, q, \text{args}'(q) \cdot k(q) := q, \text{args}'(q) \cdot k(q), p
\]

iv) As for the operation \(\text{PUSH}(S, \text{args}(x) \cdot k(x))\) we remark that it amounts to an extension of \(V\) with \(x\) (or: \(p\)); Therefore PI 1.6 (e) remains valid. Part (d) is not falsified either and part (c) can be restored by assigning to \(\text{args}'(x) \cdot k(x)\) the value \(y_{a-1}\), which is equal to \(q\) by part (b). Also, \(p = y_a\) must be assigned to \(q\) to restore (b). Finally, "\(p := \text{args}(x) \cdot k(x)\)" restores (a), but since \(x \notin V\) we may also write "\(p := \text{args}'(x) \cdot k(x)\)" on account of (e).

In summary:
PUSH(S, args(x)·k(x)) →
PUSH(S, args(x)·k(x));
p, q, args'(x)·k(x) := args(x)·k(x), p, q

The resulting algorithm is:

Algorithm 11* (action only)

action:

- U, S, p, q := ∅, <>, z, nil; PUSH(S, z);
- for x : x ∈ OB → args'(x) := args(x); k(x) := 1 rof;
- do S ≠ <>
  
  -> Let x : x = TOP(S);
  
  do k(x) ≠ m(x) + 1 cand args'(x)·k(x) ∈ U
  
  -> k(x) := k(x) + 1
  
  od;

  if k(x) = m(x) + 1 → val(x) := func(x) (VAL(args(x)));
  
  U, S := U ∪ {x}, POP(S);
  
  if q = nil → p := nil
  
  [] q ≠ nil → p, q, args'(q)·k(q) :=

  q, args'(q)·k(q), p

  fi

  [] k(x) ≠ m(x) + 1 → PUSH(S, args(x)·k(x));

  p, q, args'(x)·k(x) := args(x)·k(x), p, q
  
  fi

od

[]

First of all we show (as promised earlier) that the final value of args'(x) equals args(x) for all x : x ∈ OB, namely: At termination of algorithm 11* the assertion S = <> holds. Therefore the set V of P11.6 is empty and by P11.6 (c):

(Ax : x ∈ OB : args'(x) = args(x))

holds true.

Next, by making use of P11.6, stack S and args(x) will be removed from algorithm 11*:

Stack S is used only in guard S ≠ <> and statement "Let x : x = TOP(S)". On account of P11.6, especially part (a), S = <> is equivalent with p = nil, thus the replacement

do S ≠ <> |→ do p ≠ nil

is a legal one.

As for statement "Let x : x = TOP(S)" observe (P11.6, S ≠ <>) that p = TOP(S) holds at that stage, so we could replace Let x : x = TOP(S) |→ Let x : x = p. However, it is easy to see that x = p holds whenever x is used, so we can go even further by replacing x by p whenever it occurs.
This makes the let-statement superfluous:

Let \( x : x = \text{TOP}(S) \imp skip \)

Now consider expression "\( \text{args}(x) \cdot k(x) \in U \)" in the guard of the inner do-loop. It was shown already that this loop does not violate P11.6 whatsoever (see addition (ii) above). That is, each time the expression is evaluated P11.6 holds. In addition \( x \not\in V \) holds and hence \( \text{args}(x) \cdot k(x) = \text{args}'(x) \cdot k(x) \) by P11.6 (e).

This justifies

\[
\begin{align*}
\text{do } & k(x) \neq m(x) + 1 \text{ and } \text{args}(x) \cdot k(x) \in U \imp \text{do } & k(x) \neq m(x) + 1 \text{ and } \text{args}'(x) \cdot k(x) \in U
\end{align*}
\]

On similar grounds as above we can replace

\[
\begin{align*}
\text{val}(x) & := \text{func}(x) (\text{VAL}(\text{args}(x))) \imp \\
\text{val}(x) & := \text{func}(x) (\text{VAL}(\text{args}'(x)))
\end{align*}
\]

What remains is the for-statement during initialization, but that may be omitted: Now we have shown that it is allowed to use \( \text{args} \) (when made variable) in our algorithm we assume -just like before- that it is correctly initialized beforehand as such a datum to our algorithm.

Before giving the final algorithm of this transformation some words should be devoted to the way stack \( S \) is removed from the invariants. Up till now it has been possible to rewrite the invariants (containing the redundant variables) in terms of new variables using the relation between the old and the new variables. In the case of P11.6, however, this is impossible as the new variables \( (p, q, \text{args}') \) are expressed in terms of the "internal structure" of \( S \) (viz. the \( y_i \)) instead of in terms of \( S \) itself. To keep track of this internal structure of \( S \) is "removed" by putting a large existential quantifier in front of it: \((ES: S < Y_1, \ldots, Y_n> : p = y_n \land \cdots)\).

---

**Algorithm 12 \([EC \cdot \text{MDTVHW}]\)**

\[
\begin{align*}
\text{var: } & U : \text{set of objects}, p, q : \text{object} \\
& \text{val} : \text{OB} \to \left( \bigcup V : V \in \Omega : V \right) \\
& \text{args}' : \text{OB} \to \text{OB}^*, \ k : \text{OB} \to \mathbb{N} \\
\text{post: } & R12.0 : U = \text{dep}(\{z\}) \land U \text{ consistent w.r.t. val} \\
& R12.1 : \text{args}' = \text{args} \\
\text{inv: } & P12.0 : (Ax : x \in \text{OB} : 1 \leq k(x) \leq m(x) + 1) \\
& P12.1 : (Ax : x \in \text{OB} : (Ai : 1 \leq i < k(x) : \text{args}(x) \cdot i \in U)) \\
& P12.2 : \text{There is a stack of objects } S = < y_1, \ldots, y_n > (n \geq 0) \text{ such that} \\
& P12.2.0 : \{z\} \subseteq U \cup \text{rng}(S) \subseteq \text{dep}(\{z\}) \land U \text{ consistent w.r.t. val} \\
& P12.2.1 : \text{rng}(S) \cap U = \emptyset \\
& P12.2.2 : (Ai : 1 \leq i < n : k(y_i) \neq m(y_i) + 1) \\
& P12.2.3 : (Ai : 1 \leq i < n : \text{args}(y_i) \cdot k(y_i) = y_{i+1}) \\
& P12.2.4 : \text{Let } y_0 = y_{-1} = \text{nil}, \text{ let } V = \{y_i | 1 \leq i < n\} ; \text{ then}
\end{align*}
\]
3.8 Relaxing the size of W

In this section we shall show how algorithms meeting restriction E ("set \( W \) contains exactly one object") can be made to work for \( W \) with larger sizes also. Essentially the idea is to apply the algorithm meeting restriction E once for each object in \( W \).

To avoid re-evaluation of objects and to prevent already computed object-values from being thrown away, the set \( U \) must of course be initialized only once beforehand.

Now if \( S \) is the statement sequence of one of the algorithms meeting restriction E, and \( S' \) is the statement sequence obtained from \( S \) by leaving out the initialization of \( U \) then the following is the statement sequence of an algorithm that operates on \( W \) with size larger than one, thereby using \( S' \):

\[
\begin{align*}
U &:= \emptyset; \\
\text{for } \omega : \omega \in W & \\
\rightarrow \text{if } \omega \in U \rightarrow \text{skip} \\
\text{if } \omega \notin U & \rightarrow z := \omega; S' \\
\text{rof}
\end{align*}
\]

(Nb: As for algorithm 11 (Section 3.6) and algorithm 12 (Section 3.7) the initialization of function \( k \) may be left as it is or may be done once beforehand; it makes no difference whatsoever. (Check this.))
4. Miscellaneous

4.0 On the method of algorithm transformation

In connection with the transformational method (as a tool for the derivation of algorithms) we remark:

1. The literature seems to lack a thorough treatment of the "rules of the game" allowed with the various strategies of program transformation (such as: exchanging iteration and recursion, imperative and applicative techniques). (Recently a fairly extensive overview has become available in [Fea87].) At least for one strategy, viz. changing the representation of data, such a treatment is available in Jonkers' Ph.D thesis [Jon82]. As the applicability of this technique is necessarily limited to algorithms of a fairly constant shape (it uses only local transformations, e.g. fusion of loops is not covered by it), one may expect to need other strategies as well. (And indeed, this expectation was confirmed pretty soon in this thesis.)

2. A drawback of the method is that, at a given stage of the derivation, the number of ways to continue is overwhelming, and it is often far from obvious which way is "the best" in the sense that—in the end—it allows you to arrive at a desired algorithm and, in addition, that it does so in a fairly easy way. (In this context it should be noted that the order in which two restrictions \( p \) and \( q \) are imposed can be of decisive significance for the complexity of the resulting transformation steps: imposing \( p \) after imposing \( q \) may lead to far more complex transformation steps than imposing them the other way round.) Also, you are in constant danger of imposing a "wrong" restriction in the sense that it either blocks the path to a desired solution, or it concerns a problem that turns out to be self-resolving during the rest of the transformation process. An example of both can be found in this paper; namely consider the act of tracing an object, first introduced (together with the subset-operator) in Subsection 3.1.1 (see in particular terminology convention 3.3). As long as we have algorithms operating on set \( W' \) the tracing is done via statement

\[
W' := W' \cup \text{subset}(\text{rng}(\text{args}(x)) \setminus U).
\]

Two viable restrictions now appear:

i) The statement above has no effect if objects are traced that are already contained in \( W' \), so we may just as well write
\[ W' := W' \cup \text{subset} (\text{rng(args(x))} \setminus (U \cup W')) \]  

(1)

as an equivalent form.

However, this decision has a dramatic impact when \( W' \) is transformed in stack \( S \) sometime later on (Section 3.2) : Statement (1) above must then be rewritten as

\[
\text{Let } Y : Y = \text{subset} (\text{rng(args(x))} \setminus (U \cup \text{rng(S)}))
\]

in algorithm 4*.

Apart from the problem this poses because only TOP(S) can be inspected, the drawback of this strategy is that it excludes termination altogether in certain cases; consider for instance example 3.4 : going from situation (b) to situation (c) is then impossible and hence the algorithm never terminates (although object \( x \), needed for evaluation of \( y \), is in \( S \), it is unreachable).

ii) It is reasonable to demand that each argument of an object \( x \) is traced at most once from \( x \) (see terminology convention 3.3 for this notion).

This objective can be accomplished by introducing per object \( x \) a set \( T(x) \) that contains the arguments of \( x \) that still may be traced and by always choosing arguments from \( T(x) \setminus U \) instead of \( \text{rng(args(x))} \setminus U \), i.e. replace

\[
\text{Let } Y : Y = \text{subset} (\text{rng(args(x))} \setminus U) \rightarrow \\
\text{Let } Y : Y = \text{subset} (T(x) \setminus U) ; T(x) := T(x) \setminus Y
\]

However, the condition that each argument of \( x \) is traced only once from \( x \) is automatically established later on; without proof we mention that it holds from (and including) algorithm 6 onwards.

From point 2 above we draw the conclusion that, in order to be able to apply the transformational method in a way that is not too time-consuming one must know (or at least: about know) what kind of algorithms one is aiming at. When trying to classify a number of known algorithms this condition is fulfilled, but for the actual derivation of a new algorithm this may be different; and -in view of our experiences- in any case the conclusion is justified that application of the transformational method requires a considerable amount of backtracking.

4.1 On related work concerning attribute evaluation

Below we briefly mention three papers known from literature concerning classification of attribute evaluation algorithms and attribute grammars.

i) In [Fil83] Filè gives an hierarchy of eight tree-walking evaluators. A tree walking evaluator is an evaluation algorithm that takes as input a derivation tree of the attribute grammar and tries to evaluate (all of) the attributes of this tree by visiting the nodes of the tree one by one (where repeated visits to a node are allowed). During the visit to a node the evaluator then evaluates some of the attributes of that node.
Filè now starts off with an algorithm that is free to determine
- the order in which neighboring nodes are visited (no jumps in the tree are allowed).
- the attributes of a node to be evaluated during a particular visit to that node.
By limiting these two sorts of freedom Filè then arrives at eight types of tree-walking evaluators.
Next these evaluators are used to define classes of attribute grammars: An attribute grammar is of type X if the evaluator of type X can perform attribute evaluation for each derivation tree of this grammar.
As an important result it is then shown how the classes of attribute grammars thus obtained are related to each other: For some classes this relation is the strict containment of one class in another class, while other classes turn out to be incomparable with each other.
The relevance of Filè’s work in connection with the work started in this thesis is, that maybe the eight types of evaluators turn out to fit naturally in the hierarchy whose development was started here.

ii) In [Eng84] Engelfriet gives a considerable amount of attribute evaluation strategies known from literature. These strategies are of all different kinds, such as tree-walking evaluators and applicative evaluation methods. Little or no effort is made to relate strategies to each other; merely their existence is mentioned.

iii) In [DJL85/86], Vol. 1 an approach similar to that of Engelfriet can be found, be it that the relation of evaluators to the classes of attribute grammars they can handle is pointed out more precisely; and even, evaluators are ordered via the growing complexity of these classes.

The relevance of (ii) and (iii) for this thesis is, that a number of strategies already known are listed. These strategies should be made to fit in our resulting hierarchy. In view of the last paragraph of Section 4.0, knowing what one is aiming at is important if one is to use the transformational method.
References


Vol. 1 : *Main results on attribute grammars*, (1986)
Vol. 3 : *Classified bibliography*, (1985)
(Rapports de recherche, no. 485, 510, 417) INRIA - Rocquencourt, France


