stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 134/80 APRIL

H.B.M. JONKERS

DERIVING ALGORITHMS BY ADDING AND REMOVING VARIABLES

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

1980 Mathematics subject classification: 68B10

ACM-Computing Reviews-category: 5.24

Deriving algorithms by adding and removing variables*)

by

H.B.M. Jonkers

ABSTRACT

A simple method of deriving algorithms and showing the correctness of the derivation is described. It is based on decomposing global transformations amounting to changes of representation into a number of local transformations, the correctness of which is self-evident. The effectiveness of the method is demonstrated in a derivation and proof of correctness of the Deutsch-Schorr-Waite marking algorithm.

KEY WORDS & PHRASES: transformational programming, redundant variable, intermediate assertion, nondeterminism

---

*) This report will be submitted for publication elsewhere.

1. INTRODUCTION

Of late the transformational approach to algorithm construction is enjoying an increasing popularity. The basic idea behind the method of algorithm transformation (or "algorithmics" [10]) is to start with a simple "abstract" algorithm, which can easily be proved correct but which may be intolerably inefficient. Then a number of correctness-preserving transformations are applied to the algorithm, turning it into a more complex "concrete" algorithm, which is still correct and (hopefully) more efficient. The virtues of this approach are widely known and will not be discussed here. For a short introduction and survey the reader is referred to [3].

The correctness of the abstract algorithm which serves as a starting point for the transformation process can be proved by conventional means, e.g. by using the axiomatic method [6]. If the abstract algorithm and the problem specification coincide, this step is not even necessary. Problems arise, however, if an attempt is made to prove that the transformations applied to the abstract algorithm do not affect the correctness of the algorithm. The conventional verification methods fall short here. They must be extended with the ability to prove the correctness of algorithm transformations (see e.g. [1]), which increases the complexity of the verification process considerably.

One of the ways to overcome the above problems is not to let the algorithm constructor prove the correctness of each individual transformation applied by him, but provide him with a catalogue of transformation rules [4]. Such a transformation rule is basically a parameterized transformation, which by verifying a number of "premises" and providing the right parameters may be applied to an algorithm. Each transformation obtained this way from a transformation rule is automatically correctness-preserving. This can be proved formally, using some extended verification method [1], but that is of no concern to the algorithm constructor. The only thing he has to do is to verify the premises which must be satisfied in order to apply a transformation rule. These premises can be verified in the traditional way.

The catalogue method of transformation constitutes an interesting

approach to algorithmics. Yet there are a number of drawbacks attached to it. First of all, for any but a toy algorithmic language a rather large set of transformation rules is required in order to be able to perform all useful transformations. Second, transformation rules often deal with global transformations, which affect the entire structure of an algorithm. The correctness of such a global transformation is usually far from trivial to comprehend. Also global transformations tend to obscure an algorithm. In a mechanical algorithm transformer, such as an optimizing compiler, this is not really an objection. For a human algorithm transformer, on his way to derive a new algorithm, it is, however. He may easily lose insight into the algorithm and overlook the proper transformation.

This paper addresses the above two problems. It is argued that a rather small number of local transformation rules is sufficient to accomplish most of the necessary transformations, even global ones. The method will be described in detail in section 2. In a nutshell the idea is as follows. Let an algorithm S be given which is a correct solution to a certain problem. The introduction in S of a new variable X and the addition to S of a number of well-defined assignments to X will not affect the correctness of S. After having added X to S a number of intermediate assertions, which relate X to the other variables in S, can be proved to hold inside S. These intermediate assertions can be used to replace certain expressions in S by equivalent or more restrictive ones, which clearly does not affect the correctness of S. It may turn out then that a variable Y used in S is not used anywhere else but in assignments to Y. Consequently Y has turned into a "redundant" variable, the assignments to which may be removed from S, as well as Y itself, without affecting the correctness of S. Thus global transformations of S can be performed step by step by the following simple transformations: adding a variable X to S and adding assignments to X, making local replacements in S, removing assignments to a redundant variable Y in S and removing Y.

The above scheme constitutes a very flexible way to change the representation of variables. Since the derivation of many algorithms amounts to continually changing the representation of variables, it is also very general. In a derivation of an algorithm according to this scheme only small steps are taken, which can easily be seen to be correctness-

preserving by proving intermediate assertions (if necessary). No enhancement of existing verification techniques is therefore required, at least not to convince oneself intuitively of the correctness preservation of each step. From a strict formal point of view such an enhancement is still necessary of course. The formalization of the scheme would among many other things require a precise definition of concepts such as "correctness preservation", "redundant variable", "local replacement", etc. It is believed that this formalization will not pose any serious problems. The level of formality required for it is not sought for in this paper. Things will be kept intuitive, yet sufficiently precise to be confident about the formal soundness.

The effectiveness of the method will be demonstrated in the derivation of a well-known test case for verification techniques: the Deutsch-Schorr-Waite marking algorithm [12], henceforth called the DSW-algorithm. In contrast with most other proofs of correctness of the DSW-algorithm [5, 8, 11, 13] the most general form of the algorithm will be chosen here. In section 3 the problem will be defined precisely. From the specifications given there a simple algorithm can be derived almost immediately. This algorithm is given and proved correct in section 4 using the axiomatic method. Then, in five subsequent "phases" (sections 5 - 9), each of which follows exactly the scheme described in section 2, the DSW-algorithm is derived from this algorithm by correctness-preserving transformations. The intermediate assertions which are required in this derivation process are again proved by using the axiomatic method. The algorithmic language used is somewhat informal. As far as the semantics of the constructs of this language is not self-evident, it will be explained.

2. METHOD

In this section a detailed outline of the method will be presented as it will be applied in the next sections. We assume the problem is to construct an (efficient) algorithm S operating on a set of variables W in such a way that if the precondition P^W_{in} holds the postcondition P^W_{out} will

hold. The first stage is:

(0) Construct a simple algorithm S operating on a set of variables X, where $W \subset X$.
Assuming that the precondition P^W_{in} holds prove and insert intermediate assertions P^X_i (i = 1, 2, ...) in S.
Prove that the postcondition P^W_{out} holds.

Through the above a partially correct abstract algorithm together with a number of valid intermediate assertions is obtained. If sufficiently abstract this algorithm will probably be highly nondeterministic. Though it need not necessarily terminate, it must be such that a terminating (and consequently totally correct) algorithm can be derived from it by curtailing the nondeterminism. Termination will therefore be considered at the relevant point in the derivation.

An iterative process of correctness-preserving algorithm transformations is now started. Each iteration or "phase" can be decomposed in a number of steps which will be described below. What we have is an algorithm S operating on a set of variables X. What we want is to make S more efficient (which among other things implies making S terminate). The first step to achieve this is to introduce a number of fresh variables in the algorithm. The purpose of these variables is to gather additional information which can be used to increase the efficiency of the algorithm. Two major examples of the use of this additional information are: replacing nondeterministic operations by less nondeterministic ones and making variables redundant by replacements. The latter amounts to changing the representation of a set of variables into a more efficient one. The information to be gathered in the newly added variables should be formulated in terms of additional intermediate assertions which we wish to be valid for these variables. The first step of the iterative transformation process therefore reads as follows:

(1) Introduce a set of variables Y in S, where $X \cap Y = \emptyset$ (and possibly $Y = \emptyset$).
Formulate and insert additional intermediate assertions $Q_i^{X,Y}$ ($i = 1, 2, \ldots$) to be valid for the X- and Y-variables.

The next step is to add assignments to the Y-variables to S, in order to make the additional intermediate assertions $Q_i^{X,Y}$ hold. As it turns out, however, it is not always possible to make the intermediate assertions $Q_i^{X,Y}$ hold simply by adding assignments to the Y-variables. It may be necessary to apply a number of replacements also, which are based on the assumption that the intermediate assertions $Q_i^{X,Y}$ already hold. This situation (examples of which will be encountered) typically occurs with intermediate assertions $Q_i^{X,Y}$ inside loops, which are introduced in order to replace nondeterministic operations on the X-variables inside the loop by more deterministic operations. Intermediate assertions of this type allow the assertions on the X-variables to be strengthened. Hence it is impossible to make the $Q_i^{X,Y}$ hold solely by adding assignments to the Y-variables. Replacements involving the X-variables must also be performed. Because of the cyclic nature of loops, however, the correctness of these replacements may depend on the intermediate assertions $Q_i^{X,Y}$, the truth of which the replacements are supposed to establish. The only way out here is to assume that for the purpose of these replacements the $Q_i^{X,Y}$ already hold. After having made the assertions $Q_i^{X,Y}$ hold, their validity can then be proved.

At first sight the transformation step described above may seem to be incorrect. The point is that the intermediate assertions $Q_i^{X,Y}$ are used for replacement purposes before their truth has been established. In fact these replacements are used to help establish the truth of the $Q_i^{X,Y}$! Contradictory as it may seem this can do no harm, however. We will show that now by applying the same transformation step in a more circumstantial way.

Consider a statement S_i^X in S, prior to which the intermediate assertion P_i^X holds. This will be denoted as follows:

$$S = \ldots \{P_i^X\}\ S_i^X \ldots$$

First of all strengthen P_i^X to R_i^X, where R_i^X is the strongest assertion which holds prior to S_i^X (consequently $R_i^X \Rightarrow P_i^X$):

$$S = \ldots \{R_i^X\}\ S_i^X \ldots$$

Insert a nondeterministic assignment "X, Y := $[R_i^X \wedge Q_i^{X,Y}]$" prior to S_i^X, which assigns values to the X- and Y-variables in such a way that $R_i^X \wedge Q_i^{X,Y}$ holds afterwards. This does not affect the correctness of the algorithm, because R_i^X still holds prior to S_i^X:

$$S = \ldots \{R_i^X\}\ X, Y := [R_i^X \wedge Q_i^{X,Y}]\ \{R_i^X \wedge Q_i^{X,Y}\}\ S_i^X \ldots$$

Make replacements in S_i^X based on the validity of $Q_i^{X,Y}$ prior to S_i^X. Suppose these replacements turn S_i^X into $S_i^{X,Y}$:

$$S = \ldots \{R_i^X\}\ X, Y := [R_i^X \wedge Q_i^{X,Y}]\ \{R_i^X \wedge Q_i^{X,Y}\}\ S_i^{X,Y} \ldots$$

Make additional replacements in S and add assignments to Y-variables to S in such a way that $Q_i^{X,Y}$ will hold prior to "X, Y := $[R_i^X \wedge Q_i^{X,Y}]$":

$$S = \ldots \{R_i^X \wedge Q_i^{X,Y}\}\ X, Y := [R_i^X \wedge Q_i^{X,Y}]\ \{R_i^X \wedge Q_i^{X,Y}\}\ S_i^{X,Y} \ldots$$

Remove the nondeterministic assignment "X, Y := $[R_i^X \wedge Q_i^{X,Y}]$":

$$S = \ldots \{R_i^X \wedge Q_i^{X,Y}\}\ S_i^{X,Y} \ldots$$

Finally weaken R_i^X to P_i^X:

$$S = \ldots \{P_i^X \wedge Q_i^{X,Y}\}\ S_i^{X,Y} \ldots$$

The above sequence of transformation steps is evidently correct and can be applied simultaneously to all statements of S. In its effect it is the same as the original transformation step the correctness of which was questioned. Consequently the latter is also correct. This step is summarized below:

(2) Assuming the additional intermediate assertions $Q_i^{X,Y}$ hold make replacements in S and add assignments to the Y-variables to S in order to make the $Q_i^{X,Y}$ hold.
Prove that the intermediate assertions $Q_i^{X,Y}$ hold.

The third step is to fully exploit the new intermediate assertions to make replacements in S. Strictly speaking this could already be done in the second step, but from a conceptual point of view it is better to separate the replacements necessary to make the intermediate assertions $Q_i^{X,Y}$ hold from the other "optimizing" replacements. The class of replacements allowed will not be defined here. The only requirement is that the replacements must be very simple and evidently correctness-preserving. The replacements can be used either to replace expressions by more efficient ones, or to turn certain variables into redundant variables. What is exactly meant by a "redundant variable" will not be defined here. Broadly speaking a variable is redundant in an algorithm if it is a local variable of the algorithm and it is used in assignments to itself only. It is obvious that the assignments to such a variable may be removed from the algorithm without affecting the correctness. This is step 3:

(3) Choose a set of variables $Z \subset X \cup Y$, where $W \cap Z = \emptyset$, which are to be made redundant (possibly $Z = \emptyset$).
Using the intermediate assertions make a number of replacements in S which turn the Z-variables into redundant variables and remove all assignments to Z-variables.

The third step can be viewed in a sense as the reverse of the second step. Analogously the fourth step can be viewed as the reverse of the first step. Instead of introducing variables we are going to remove them and instead of strengthening the intermediate assertions we are going to weaken them. In step 3 all assignments to redundant variables have been removed. Consequently these variables have turned into "ghost variables", which may be removed from the algorithm. However, these variables may (and probably will) still occur in intermediate assertions. From a strict point of view these assertions no longer hold now. Simply throwing them away would

probably make the remaining intermediate assertions too weak for further use. Therefore new and sufficiently strong assertions, in which the redundant variables no longer occur, must be derived from the old assertions to take their place. This could be done in a systematic way by putting an existential quantifier before each intermediate assertion, quantifying over each redundant variable. It is easy to see that these derived intermediate assertions will hold. So we have:

(4) Replace the old intermediate assertions $P_i^X \wedge Q_i^{X,Y}$ by new assertions P_i^V implied by the old and containing only V-variables, where $V = (X \cup Y) \setminus Z$.

Through steps 1 to 4 a global correctness-preserving transformation can be performed in a stepwise way. These steps can be repeated until a sufficiently efficient algorithm is obtained. If necessary, prior to step 1 or between steps 2 and 3 new intermediate assertions can be proved and inserted. Though the final algorithm obtained this way is partially correct "by construction", it must still be proved to terminate. This need not necessarily be done afterwards, but can be done at some intermediate stage in the derivation.

If desirable, the intermediate assertions of the final algorithm can be used to give an independent proof of correctness of that algorithm. This saves one the trouble of inventing the intermediate assertions required for an independent proof of correctness. It may turn out, however, that the intermediate assertions of the final algorithm are too weak for that purpose. If an independent proof of correctness of the final algorithm should be possible, care must therefore be taken to keep the intermediate assertions strong enough. The latter is entirely the responsibility of the algorithm constructor.

The effectiveness of the method will now be demonstrated in a derivation of the DSW-algorithm exactly along the lines described above. Since the algorithm consists of a single loop, it is more convenient in the derivation to keep track of the loop invariants instead of the intermediate assertions mentioned above. Invariants instead of intermediate assertions will therefore be used in the sequel. Each invariant corresponds to four

intermediate assertions: one immediately before the loop, one at the beginning and one at the end of the loop body, and one immediately after the loop. If intermediate assertions at other places in the algorithm are required in order to apply a transformation, they can usually be derived from the invariants rather easily. We start with a definition of the problem in the next section.

3. PROBLEM

Given is a finite set G of objects. Each object is composed of a finite number of components. The set of all components of an object X is denoted as comp(X). Different objects have different components (so objects do not "overlap"). Associated to each object X is a unique reference, denoted as ref(X), which is said to refer to X. The unique object which has reference p associated to it, will be denoted as obj(p). Each component C of an object contains a value, denoted as val(C). A reference is a value. Among other values (which we are not interested in here) references may therefore be contained in components of objects. A component of an object which contains a reference will be called a branch of the object. The set of all branches of an object X will be denoted as base(X) and the number of branches as degree(X). The branches of X are numbered from 1 to degree(X). The i-th branch of X (where $1 \leq i \leq$ degree(X)) is denoted as branch(X, i). Objects will be pictured as in Fig. 1. There is a dummy object, denoted as null, which is not an element of G. The reference of null is denoted as nil: nil = ref(null).

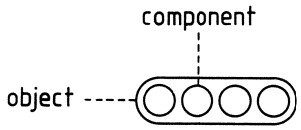


Fig. 1

The set G of objects is closed. This implies that for each reference p contained in a branch of an object in G, the object referred to by p is also in G. There is one special object R in G, called the root. G can now be viewed as a directed graph, where the objects are the nodes and the references contained in branches are the edges of the graph. An example of how G may look like is given in Fig. 2.

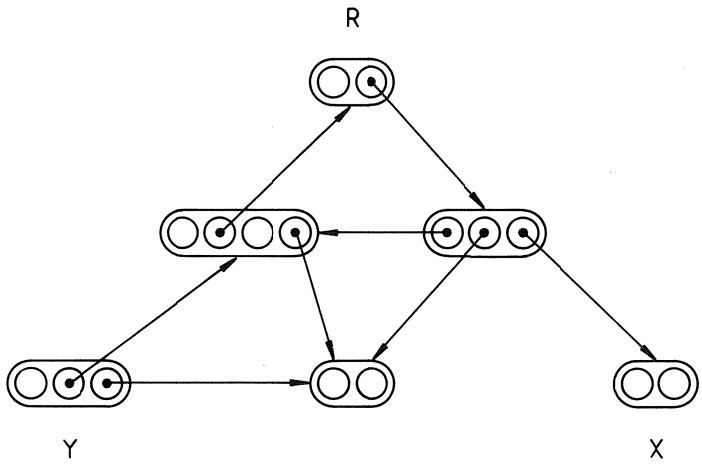


Fig.2

The concept of reachability for objects in G is defined by the following rules:

(1) The root R is reachable.
(2) If X is a reachable object,
    $B \in$ base(X),
    Y = obj(val(B)),
    then Y is reachable.
(3) An object is reachable on account of the above rules only.

For instance in Fig. 2 X is a reachable object and Y is an unreachable object.

The problem is to construct an algorithm which determines the set of all reachable objects. Such an algorithm is traditionally called a "marking algorithm". For the description of marking algorithms a variable set M of

objects will be introduced. It is the job of a marking algorithm to establish the truth of the following assertion:

$M = \{X \in G \mid X \text{ is reachable}\}$

It follows directly from the definition of reachability that this assertion is equivalent to the conjunction of the following three assertions:

A1. $R \in M$.
A2. $\forall X \in M \; \forall B \in base(X)$ [obj(val(B)) $\in M$].
A3. $\forall X \in M$ [X is reachable].

The DSW-algorithm, which is a particular solution to the above problem, will now be derived in six "phases". In the initial phase (phase 0) a simple algorithm is constructed, which serves as the starting point.

4. PHASE 0: GETTING STARTED

Looking at the definition of reachability one sees that it is almost an algorithm itself. That is, if we start with $M = \{R\}$ and repeat the following actions "long enough", M will finally become equal to the set of reachable objects:

```
Let X ∈ M.
If base(X) ≠ ∅
| Let B ∈ base(X).
| Let Y = obj(val(B))'.
| M := M ∪ {Y}.
```

Here the operations "Let $X \in M$" and "Let $B \in base(X)$" select an element from a set in a nondeterministic way. This nondeterminism can be thought of as being governed by a "demon". The first part of the derivation of the DSW-algorithm mainly consists of "exorcising" this demon, i.e. convert it

to determinism.

The question is what "long enough" means. A marking algorithm should establish the truth of the assertions A1, A2 and A3. The assertions A1 and A3 are initially true and are not affected by the above actions. Now one could say that "long enough" means: until assertion A2 holds. The process need not stop exactly at the point where this assertion holds for the first time, however (most known marking algorithms don't). Any point beyond this point will do as a termination point. In order to model this the following nondeterministic construct will be introduced:

```
Beyond A
| S.
```

where S is a series of actions and A is an assertion. It prescribes that S must be repeated until some (but not necessarily the first) point where A holds. Note that prior to an execution of S, the assertion $\neg A$ need not necessarily hold. The termination point is supposed to be chosen nondeterministically by the demon.

The above construct turns out to be very useful in the derivation of algorithms. From an algorithm containing this construct a new algorithm can be derived by replacing the assertion A by an other assertion B which is a sufficient condition for A, i.e. $B \Rightarrow A$. If the old algorithm was partially correct, the new one will also be. Neither of the algorithms needs to terminate, however. The termination of any algorithm containing the above construct will depend upon the nature of the demon. The demon could for instance be "unfair" and refuse to choose a termination point even if the termination condition holds after each iteration. This can be prevented by replacing the above construct by the deterministic construct:

```
Until A
| S.
```

which prescribes zero or more repetitions of S until A holds for the first time. Note that prior to an execution of S the assertion $\neg A$ will now hold.

As indicated above, the nondeterministic algorithms considered here

need not terminate. Therefore some people may not call them algorithms at all, but here we will. Nondeterministic algorithms are viewed here as "abstractions" of (more) deterministic algorithms. The demon represents the part of these abstract algorithms which has been "abstracted away". Certain terminating and non-terminating algorithms have the same abstraction. So in the inverse process of abstraction, i.e. the derivation of algorithms, it is often possible to derive both terminating and non-terminating algorithms from nondeterministic algorithms. This also applies to the following nondeterministic algorithm which will be chosen as a starting point for the derivation of the DSW-algorithm:

Algorithm 1

```
M := {R}.
Beyond ∀ X ∈ M ∀ B ∈ base(X) [obj(val(B)) ∈ M]
| Let X ∈ M.
| If base(X) ≠ ∅
| | Let B ∈ base(X).
| | Let Y = obj(val(B)).
| | M := M ∪ {Y}.
```

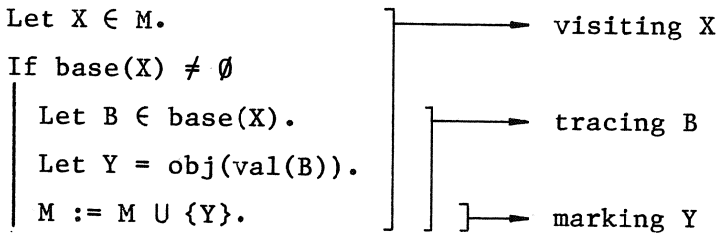
The (partial) correctness of this algorithm should be obvious. It can formally be established by proving that A1 and A3 hold immediately before the loop and are kept invariant by the loop body. Assertions which satisfy the latter properties will (as usual) be referred to as "invariants". So for Algorithm 1 we have:

Invariants

1.1. R ∈ M.

1.2. ∀ X ∈ M [X is reachable].

In the next sections the actions occurring in the body of the loop will be referred to as indicated below:



All following sections will be divided in four subsections, each of which corresponds to one of the four transformation steps described in section 2. The next two derivation phases will consist of limiting the freedom of the demon in such a way, that even though the algorithm remains nondeterministic, termination is guaranteed.

5. PHASE 1: RESTRICTING THE TRACING OF BRANCHES

Even if in Algorithm 1 the beyond-construct was replaced by an until-construct, the algorithm need not terminate. The reason is that there is too much freedom in the choice of objects to be visited and branches to be traced. The demon could for instance choose the same object and the same branch in each iteration of the loop. Consequently the termination condition would never hold (except in trivial cases). Our primary concern will therefore be to impose restrictions on the visiting of objects and tracing of branches in such a way, that the termination condition of Algorithm 1 will hold in a finite number of iterations.

What are reasonable restrictions? A general reasonable restriction which may be imposed on an algorithm is, that it should not do the same thing twice if once is enough. Let us apply this principle to the tracing of branches first. It is easy to see that it makes no sense to trace a branch more than once in Algorithm 1. The second time a branch B would be traced, the object Y referred to by the value of B would already have been marked. So the following restriction is reasonable:

Restriction 1

A branch may be traced only once.

We will now transform Algorithm 1 in such a way that this restriction is met.

Step 1

The enforcement of Restriction 1 introduces a certain overhead. The demon must be prevented to select a branch which has already been traced. For that purpose a variable set C(X) of branches of X will be associated to each object X with the following interpretation:

Interpretation 1

For each object $X \in M$, C(X) is equal to the set of branches of X which have not yet been traced.

This interpretation of C, which is of course strictly informal, can immediately be translated in a number of invariants for the algorithm to be derived (by adding C). First of all the obvious invariant:

Invariant 1.3

$\forall\ X \in M\ [C(X) \subset base(X)]$.

Second, each branch of an object X which is not an element of C(X) has already been traced. For each branch B which has been traced the object referred to by the value of B has been marked. Consequently we have:

Invariant 1.4

$\forall\ X \in M\ \forall\ B \in base(X) \setminus C(X)\ [obj(val(B)) \in M]$.

Step 2

Let us now insert assignments to C in Algorithm 1 according to Interpretation 1, thus making sure Invariants 1.3 and 1.4 hold. First of all C(X) must be properly initialized for each object X. For the root this leads to:

Addition 1.1

M := {R} ⟶

M, C(R) := {R}, base(R)

For all other objects Y, C(Y) must be initialized to base(Y) as soon as Y is marked for the first time. Whether an object is marked for the first time can be determined by testing whether $Y \notin M$ prior to marking Y, resulting in:

Addition 1.2

M := M ∪ {Y} ⟶

If $Y \notin M$

| C(Y) := base(Y).

M := M ∪ {Y}

After having traced a branch B of an object X, B must be removed from C(X). This can be accomplished by:

Addition 1.3

Let $B \in base(X)$ ⟶

Let $B \in base(X)$.

$C(X) := C(X) \setminus \{B\}$

Note that C(X) is well-defined here because $X \in M$. The above additions transform Algorithm 1 into Algorithm 1a for which besides Invariants 1.1 and 1.2 the additional Invariants 1.3 and 1.4 hold, as can easily be proved:

Algorithm 1a

```
M, C(R) := {R}, base(R).
Beyond ∀ X ∈ M ∀ B ∈ base(X) [obj(val(B)) ∈ M]
| Let X ∈ M.
| If base(X) ≠ ∅
| | Let B ∈ base(X).
| | C(X) := C(X) \ {B}.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | C(Y) := base(Y).
| | M := M ∪ {Y}.
```

Step 3

In this step the invariants will be used to make replacements in Algorithm 1a. Among other things these replacements will be used to enforce Restriction 1. No variables will be made redundant. First, suppose an object X for which $C(X) = \emptyset$ is visited. All branches of X have then already been traced, and using Invariant 1.4 it can easily be seen that tracing a branch B of X has no effect whatsoever on M or C. Consequently tracing a branch B of an object X may be omitted if $C(X) = \emptyset$, which justifies the following replacement:

Replacement 1.1

$base(X) \neq \emptyset \longrightarrow$
$\quad C(X) \neq \emptyset$

Since we are now sure that $C(X) \neq \emptyset$, when selecting a branch B of X to be traced, B can just as well be selected from C(X) (which is a subset of base(X) according to Invariant 1.3) instead of base(X):

Replacement 1.2

Let $B \in base(X)$ ⟶
    Let $B \in C(X)$

The above two replacements enforce Restriction 1. Two more replacements will be applied in order to "improve" Algorithm 1a.

Let us look at the termination condition of Algorithm 1a (i.e. assertion A2). It follows directly from Invariant 1.4 that this condition is implied by the simpler condition:

$$\forall X \in M\ [C(X) = \emptyset]$$

Hence the following replacement is in order:

Replacement 1.3

$\forall X \in M\ \forall B \in base(X)\ [obj(val(B)) \in M]$ ⟶
    $\forall X \in M\ [C(X) = \emptyset]$

Finally it is easy to see that marking an object Y makes sense only if $Y \notin M$. This leads to the following optimization:

Replacement 1.4

If $Y \notin M$
| $C(Y) := base(Y)$.
$M := M \cup \{Y\}$ ⟶

    If $Y \notin M$
    | $M, C(Y) := M \cup \{Y\}, base(Y)$

This concludes the third step.

Step 4

In this step possible redundant variables are supposed to be removed. Since there are none, it suffices to give the final algorithm of this first

transformation phase together with its invariants:

Algorithm 2

```
M, C(R) := {R}, base(R).
Beyond ∀ X ∈ M [C(X) = ∅]
| Let X ∈ M.
| If C(X) ≠ ∅
| | Let B ∈ C(X).
| | C(X) := C(X) \ {B}.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, C(Y) := M ∪ {Y}, base(Y).
```

Invariants

2.1. $R \in M$.

2.2. $\forall X \in M$ [X is reachable].

2.3. $\forall X \in M$ [$C(X) \subset base(X)$].

2.4. $\forall X \in M \; \forall B \in base(X) \setminus C(X)$ [$obj(val(B)) \in M$].

Note that only Invariant 2.4 is temporarily disturbed inside the loop.

6. PHASE 2: RESTRICTING THE VISITING OF OBJECTS

In this phase restrictions will be imposed on the visiting of objects. Visiting an object X is useless if all branches of X have already been traced. A proper restriction would therefore be: only objects X with $C(X) \neq \emptyset$ may be visited. Since in Algorithm 2 at the beginning of a visit to an object X it is already checked whether $C(X) \neq \emptyset$, it is convenient to weaken this restriction a little and allow for one visit when $C(X) = \emptyset$. This extra visit can then be used to establish that $C(X) = \emptyset$ and take measures to prevent that X is visited again. Hence we will impose the following restriction:

Restriction 2

As soon as $C(X) = \emptyset$, X may be selected for a visit at most once.

Step 1

Again the enforcement of this restriction introduces a certain overhead. The demon must be prevented to select an object X for a visit for which $C(X) = \emptyset$ and which has already been visited (once) since $C(X) = \emptyset$. This will be accomplished through the introduction of a variable set U of marked objects. U has the following interpretation:

Interpretation 2

U is equal to the set of all marked objects X for which either:

- $C(X) \neq \emptyset$, or
- $C(X) = \emptyset$ and X has not been selected for a visit since $C(X) = \emptyset$.

It follows immediately from this interpretation of U that the following invariant should hold:

Invariant 2.5

$U \subset M$.

Since for each marked object X, $X \notin U$ implies that $\neg(C(X) \neq \emptyset)$, we also have:

Invariant 2.6

$\forall X \in M \setminus U \; [C(X) = \emptyset]$.

Step 2

Assignments to U will now be added to Algorithm 2 according to Interpretation 2, so as to make Invariants 2.5 and 2.6 hold. First the initialization of U, which is obvious:

Addition 2.1

M, C(R) := {R}, base(R) ⟶

M, C(R), U := {R}, base(R), {R}

The first (and as will turn out the only) time an object is a candidate for addition to U is when the object is marked. At the moment an object X is marked (for the first and only time) in Algorithm 2 it clearly satisfies one of the two conditions specified in Interpretation 2. It should therefore be added to U:

Addition 2.2

M, C(Y) := M ∪ {Y}, base(Y) ⟶

M, C(Y), U := M ∪ {Y}, base(Y), U ∪ {Y}

It follows from Interpretation 2 that an object X must be removed from U the first time it is selected for a visit when C(X) = ∅. This can be accomplished by adding an else-part to the conditional clause "If C(X) ≠ ∅ ..." in Algorithm 2:

Addition 2.3

```
If C(X) ≠ ∅  ⟶
| ...
```

```
If C(X) ≠ ∅
| ...
else
| U := U \ {X}
```

As soon as an object X is removed from U, C(X) = ∅ and will remain so. Hence X need never be added to U again. All provisions to keep track of U according to Interpretation 2 have thus been made. The additional Invariants 2.5 and 2.6 can easily be proved to hold for the algorithm obtained by applying the above additions to Algorithm 2:

Algorithm 2a

```
M, C(R), U := {R}, base(R), {R}.
Beyond ∀ X ∈ M [C(X) = ∅]
| Let X ∈ M.
| If C(X) ≠ ∅
| | Let B ∈ C(X).
| | C(X) := C(X) \ {B}.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, C(Y), U := M ∪ {Y}, base(Y), U ∪ {Y}.
| else
| | U := U \ {X}.
```

Step 3

Replacements will now be made to enforce Restriction 2, using the additional information gathered in the variable U. At first sight Restriction 2 can easily be enforced by selecting an object X for a visit from U instead of M. This poses a little problem, however, because U may be empty. Therefore first provisions will be made to ensure that $U \neq \emptyset$ prior to an iteration of the loop.

Consider the termination condition of Algorithm 2. It follows from Invariant 2.6 that this condition is implied by the condition:

$$U = \emptyset$$

So the following replacement is allowed:

Replacement 2.1

$$\forall X \in M\ [C(X) = \emptyset] \longrightarrow U = \emptyset$$

This replacement in itself is not enough to ensure that $U \neq \emptyset$ prior to an iteration of the loop. It is, however, if the beyond construct is replaced

by an until construct:

Replacement 2.2

Beyond ⟶

    Until

Restriction 2 is now enforced by:

Replacement 2.3

Let X ∈ M ⟶

    Let X ∈ U

Step 4

Again no redundant variables occur in the algorithm derived so far. The variables C and U have only been used to restrict nondeterminism and not to change the representation of other variables. The final algorithm of this transformation step (and consequently the entire transformation phase) is therefore equal to the final algorithm of the previous step:

Algorithm 3

```
M, C(R), U := {R}, base(R), {R}.
Until U = ∅
| Let X ∈ U.
| If C(X) ≠ ∅
| | Let B ∈ C(X).
| | C(X) := C(X) \ {B}.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, C(Y), U := M ∪ {Y}, base(Y), U ∪ {Y}.
| else
| | U := U \ {X}.
```

Invariants

3.1. $R \in M$.

3.2. $\forall X \in M$ [X is reachable].

3.3. $\forall X \in M$ [$C(X) \subset base(X)$].

3.4. $\forall X \in M \; \forall B \in base(X) \setminus C(X)$ [$obj(val(B)) \in M$].

3.5. $U \subset M$.

3.6. $\forall X \in M \setminus U$ [$C(X) = \emptyset$].

Interlude: termination

Having restrained the visiting of objects and tracing of branches drastically and having replaced the nondeterministic beyond construct by the deterministic until construct, Algorithm 3 may be expected to terminate irrespective of the nature of the (not yet fully excorcised) demon. This can be established more formally as follows. During each iteration of the loop in Algorithm 3 a marked object X is visited. If $C(X) \neq \emptyset$, a branch B of X is traced, which has not yet been traced before according to Restriction 1. If $C(X) = \emptyset$, X is removed from U and will not be visited a next time according to Restriction 2. Hence the sum of the number of branches of marked objects, which have already been traced, and the number of marked objects which will not be visited again, will increase by one with each iteration of the loop. Translated into more formal terms this implies that the value of the following expression will increase by one with each iteration of the loop:

$$\# M \setminus U + \sum_{X \in M} \# (base(X) \setminus C(X))$$

The fact that this is indeed so, can easily be verified. Because of the finiteness of the number of objects and branches, the value of this expression has a finite upper bound. Termination of Algorithm 3 is thereby guaranteed.

The fact that the value of the above expression increases by 1 with each iteration of the loop allows an even stronger statement on the termination of Algorithm 3. The initial value of the above expression is 1.

At termination of Algorithm 3 $U = \emptyset$ and $C(X) = \emptyset$ for each $X \in M$. The final value of the expression is therefore:

$$\# Q + \sum_{X \in Q} \# \text{base}(X)$$

where Q is the set of reachable objects. Consequently Algorithm 3 will terminate after the following number of iterations:

$$-1 + \sum_{X \in Q} (1 + \text{degree}(X))$$

This implies that Algorithm 3 operates in a time which is linear in the number of reachable objects and the number of branches of reachable objects, which is the best we can get.

7. PHASE 3: CHANGING THE REPRESENTATION OF C

In this step and the following the exorcising of the demon will be completed. The remaining places where the demon resides are the operations "Let $X \in U$" and "Let $B \in C(X)$". Here we shall consider the operation "Let $B \in C(X)$". The only operations which are performed on C(X) are initialization, testing for equality to $\emptyset$, and selecting and immediately thereafter removing an element. The following restriction, which eliminates the demon from "Let $B \in C(X)$", is therefore enforceable:

Restriction 3

Branches are selected and removed from C(X) in the order of their numbering.

Step 1

Restriction 3 can be complied with by associating a variable counter k(X) to each marked object X with the following interpretation:

Interpretation 3

For each object $X \in M$, $k(X)$ is the number of the last branch which has been removed from $C(X)$. If no branches have been removed from $C(X)$ yet, $k(X) = 0$.

This interpretation implies that k must first of all satisfy the following invariant:

Invariant 3.7

$\forall\ X \in M\ [0 \leq k(X) \leq \text{degree}(X)]$.

Moreover, Restriction 3 together with Interpretations 1 and 3 imply that the following invariant should hold:

Invariant 3.8

$\forall\ X \in M\ [C(X) = \{\text{branch}(X, i) \mid k(X) < i \leq \text{degree}(X)\}]$.

Step 2

In this step assignments to k should be added in agreement with Interpretation 3 in order to make Invariants 3.7 and 3.8 hold. However, Invariant 3.8 cannot be made to hold without also making some replacements, which are based on the assumption that Invariants 3.7 and 3.8 already hold. The reason for that is that in contrast with the invariants derived before, Invariant 3.8 depends critically on the restriction of nondeterminism (Restriction 3) to be enforced and not solely on the interpretation of the new variable (k). Invariant 3.8 can therefore only be made to hold by enforcing that restriction through a replacement first. This is an example, in which it is essential that the new intermediate assertions (the invariants) are used for replacements before their truth has been established. Another example will be met in the next phase.

Let us perform the additions and replacements required to make Invariants 3.7 and 3.8 hold now. The initialization of k, which should be done together with the initialization of C, is obvious and leads to the

following additions:

Addition 3.1

M, C(R), U := {R}, base(R), {R} ⟶
    M, C(R), U, k(R) := {R}, base(R), {R}, 0

Addition 3.2

M, C(Y), U := M ∪ {Y}, base(Y), U ∪ {Y} ⟶
    M, C(Y), U, k(Y) := M ∪ {Y}, base(Y), U ∪ {Y}, 0

The only statement which disturbs Invariant 3.8 is "C(X) := C(X) \ {B}". Hence an assignment to k(X) should be added to this statement. First we must make sure, however, that B is chosen according to Restriction 3, because otherwise it is impossible to restore Invariant 3.8. That is, instead of selecting an arbitrary branch B from C(X), the (k(X) + 1)-st branch of X must be chosen. It must be assumed for that purpose, that Invariants 3.7 and 3.8 hold prior to "Let B ∈ C(X)". From these invariants and the fact that C(X) ≠ ∅ can be derived that indeed $1 \leq k(X) + 1 \leq degree(X)$ and branch(X, k(X) + 1) ∈ C(X):

Replacement 3.1

Let B ∈ C(X) ⟶
    Let B = branch(X, k(X) + 1)

Invariant 3.8 is now restored by:

Addition 3.3

C(X) := C(X) \ {B} ⟶
    C(X), k(X) := C(X) \ {B}, k(X) + 1

The only thing that remains to be done is to prove that Invariants 3.7 and 3.8 hold indeed, which is left to the reader. This completes step 2, in which Restriction 3 was enforced. This is the algorithm we have so far:

Algorithm 3a

```
M, C(R), U, k(R) := {R}, base(R), {R}, 0.
Until U = ∅
| Let X ∈ U.
| If C(X) ≠ ∅
| | Let B = branch(X, k(X) + 1).
| | C(X), k(X) := C(X) \ {B}, k(X) + 1.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, C(Y), U, k(Y) := M ∪ {Y}, base(Y), U ∪ {Y}, 0.
| else
| | U := U \ {X}.
```

Step 3

In this step C will be turned into a redundant variable. The only place where the value of C is used in Algorithm 3a is in the test "C(X) ≠ ∅". Invariants 3.7 and 3.8 imply that this test is equivalent to "k(X) ≠ degree(X)", which results in the following replacement:

Replacement 3.2

C(X) ≠ ∅ ⟶
    k(X) ≠ degree(X)

C has now turned into a redundant variable the assignments to which may be removed:

Removal 3.1

C(X), k(X) := C(X) \ {B}, k(X) + 1 ⟶
    k(X) := k(X) + 1

Removal 3.2

M, C(Y), U, k(Y) := M ∪ {Y}, base(Y), U ∪ {Y}, 0 ⟶
    M, U, k(Y) := M ∪ {Y}, U ∪ {Y}, 0

Removal 3.3

M, C(R), U, k(R) := {R}, base(R), {R}, 0 ⟶
    M, U, k(R) := {R}, {R}, 0

Finally the following optimizing replacement is applied, the omission of which would be an eye-sore to any right-minded programmer:

Replacement 3.3

Let B = branch(X, k(X) + 1).
k(X) := k(X) + 1 ⟶
    k(X) := k(X) + 1.
    Let B = branch(X, k(X))

Step 4

The variable C no longer occurs in the algorithm and may be disposed of. Yet C still occurs in the invariants. New (and preferably equivalent) invariants must be derived from these invariants. This is a straightforward matter. The final algorithm and the result of rewriting the invariants is:

Algorithm 4

```
M, U, k(R) := {R}, {R}, 0.
Until U = ∅
| Let X ∈ U.
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X, k(X)).
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, U, k(Y) := M ∪ {Y}, U ∪ {Y}, 0.
| else
| | U := U \ {X}.
```

Invariants

4.1. $R \in M$.

4.2. $\forall X \in M$ [X is reachable].

4.3. $\forall X \in M$ [$0 \leq k(X) \leq$ degree(X)].

4.4. $\forall X \in M \; \forall i = 1, \ldots, k(X)$ [obj(val(branch(X, i))) $\in M$].

4.5. $U \subset M$.

4.6. $\forall X \in M \setminus U$ [k(X) = degree(X)].

8. PHASE 4: CHANGING THE REPRESENTATION OF U

Let us consider the operation "Let $X \in U$" now. Apart from this operation the only operations which are performed on U are adding an object Y (which is not yet in U) to U and removing the (arbitrarily chosen) object X from U. This makes the following a feasible restriction:

Restriction 4

Objects are added to and removed from U in a last-in first-out manner.

The purpose of this restriction is, of course, to be able to "implement" U efficiently as a stack.

Step 1

Introduce a variable stack S of objects. This stack has the following obvious interpretation:

Interpretation 4

S contains the objects in U in the order of their addition to U (the most recently added object at the top of S).

This interpretation of S implies the following invariant:

Invariant 4.7

If $S = \langle X_1, \ldots, X_n \rangle$ then $U = \{X_1, \ldots, X_n\}$.

Here $\langle X_1, \ldots, X_n \rangle$ is the stack containing the objects $X_1, \ldots, X_n$, where X_n is the top of the stack.

Step 2

Assignments to S should be added according to Restriction 4 and Interpretation 4, thereby establishing the truth of Invariant 4.7. As in the second step of the previous phase, this is not possible without making some replacements based on Invariant 4.7 also. All operations modifying U must be accompanied by operations modifying S. First of all S should be initialized together with U:

Addition 4.1

M, U, k(R) := {R}, {R}, 0 ⟶

M, U, k(R), S := {R}, {R}, 0, <R>

The addition of an element to U should be accompanied by a "push" operation:

Addition 4.2

M, U, k(Y) := M ∪ {Y}, U ∪ {Y}, 0 ⟶
    M, U, k(Y), S := M ∪ {Y}, U ∪ {Y}, 0, push(S, Y)

The removal of an element from U (in "U := U \ {X}") poses a problem, because we can only remove an element from S if that element is at the top of S (through a "pop" operation). So we must make sure X is at the top of S. Invariant 4.7 implies that top(S) ∈ U, which justifies the following replacement:

Replacement 4.1

Let X ∈ U ⟶
    Let X = top(S)

X can now be popped from S:

Addition 4.3

U := U \ {X} ⟶
    U, S := U \ {X}, pop(S)

The conclusion of this step is to prove that Invariant 4.7 holds in the newly derived algorithm. Notice that for this proof the proof of an additional invariant is required:

Invariant 4.8

All elements of S are different.

The combined proof of Invariants 4.7 and 4.8 is simple (use Invariant 4.5). Here is the final algorithm of this step:

Algorithm 4a

```
M, U, k(R), S := {R}, {R}, 0, <R>.
Until U = ∅
| Let X = top(S).
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X, k(X)).
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, U, k(Y), S := M ∪ {Y}, U ∪ {Y}, 0, push(S, Y).
| else
| | U, S := U \ {X}, pop(S).
```

Step 3

In this step the change of representation from U to S must be completed by turning U into a redundant variable and by subsequently removing all assignments to U. The value of U is used in Algorithm 4a only in the test "U = ∅". Invariant 4.7 implies that this test is equivalent to "S = <>", where "<>" is the empty stack:

Replacement 4.2

U = ∅ ⟶
S = <>

U has become a redundant variable this way. All assignments to U may be removed:

Removal 4.1

U, S := U \ {X}, pop(S) ⟶
S := pop(S)

Removal 4.2

M, U, k(Y), S := M U {Y}, U U {Y}, 0, push(S, Y) ⟶

M, k(Y), S := M U {Y}, 0, push(S, Y)

Removal 4.3

M, U, k(R), S := {R}, {R}, 0, <R> ⟶

M, k(R), S := {R}, 0, <R>

Step 4

In this step the removal of U must formally be completed by eliminating U also from the invariants. As in the previous phase this is straightforward. The final algorithm of this phase together with the rewritten invariants is given below. For notational convenience the stack S is occasionally considered as the set of its elements in the invariants.

Algorithm 5

```
M, k(R), S := {R}, 0, <R>.
Until S = <>
| Let X = top(S)
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X, k(X)).
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, k(Y), S := M U {Y}, 0, push(S, Y).
| else
| | S := pop(S).
```

Invariants

5.1. $R \in M$.

5.2. $\forall X \in M$ [X is reachable].

5.3. $\forall X \in M$ $[0 \leq k(X) \leq \text{degree}(X)]$.

5.4. $\forall X \in M \; \forall i = 1, \ldots, k(X)$ $[\text{obj}(\text{val}(\text{branch}(X, i))) \in M]$.

5.5. $S \subset M$.

5.6. $\forall X \in M \setminus S$ $[k(X) = \text{degree}(X)]$.

5.7. All elements of S are different.

9. PHASE 5: CHANGING THE REPRESENTATION OF S, OR: THE DSW-IDEA

In this phase the actual DSW-idea will be applied, which in fact is nothing but a change of representation. In contrast with the previous changes of representation (from C to k and U to S) this change of representation is not accompanied by a reduction of nondeterminism. This would be impossible in the first place, because through the successive restrictions enforced in the previous phases Algorithm 5 has turned into a completely deterministic algorithm. No "restrictions" will or can therefore be imposed in this phase.

In order to demonstrate the DSW-idea let us take a closer look at Algorithm 5. It is very easy to infer from Algorithm 5 that whenever there is an object X at the top of the stack S and an object Y is pushed on top of it, the k(X)-th branch of X contains a reference to Y. This makes S look as shown in Fig. 3.a (in this picture objects are assumed to be composed of exactly four branches). It amounts to the following invariant which can easily be proved:

Invariant 5.8

If $S = \langle X_1, \ldots, X_n \rangle$ then

1. $\forall i = 1, \ldots, n - 1$ $[k(X_i) > 0]$.

2. $\forall i = 1, \ldots, n - 1$ $[\text{val}(\text{branch}(X_i, k(X_i))) = \text{ref}(X_{i+1})]$.

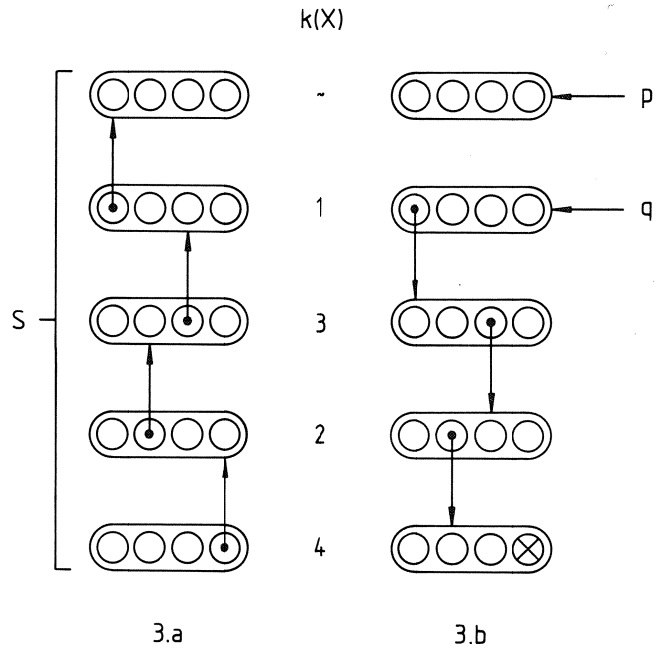


Fig. 3

The basic DSW-idea is that using two variable references p and q the situation of Fig. 3.a can be transformed without loss of information into the situation of Fig. 3.b. Here the cross in the fourth branch of the object at the bottom of the stack is the dummy reference nil (see section 3). The situation of Fig. 3.b has the advantage over the situation of Fig. 3.a that it makes the stack S redundant: all stack operations can be expressed in terms of operations on the variables p and q and the contents of branches. Put otherwise: Fig. 3.b sketches an implementation of S without any space overhead (apart from the two variable references p and q).

The application of the DSW-idea to Algorithm 5 raises a little problem. It is apparently assumed that the value of a component of an object is variable. Otherwise the transformation from Fig. 3.a to Fig. 3.b would never be possible. Up till now the value of a component of an object was assumed to be constant. Simply making the function val variable and adding modifications of val (according to Fig. 3.b) to Algorithm 5 does not work, however, because these changes may affect the correctness of the

algorithm. The solution, of course, is to introduce alongside the constant function val an extra variable function VAL, which is initially equal to val. Modifications to VAL may freely be added to Algorithm 5 because they in no way affect the correctness of the algorithm. After having added the variables p, q and VAL according to the DSW-idea to Algorithm 5, the job is then to eliminate the stack S and the function val from the algorithm (using invariants). Finally, in order to show that VAL can just as well be replaced by val (made variable) it must be shown that the final value of VAL is equal to val.

Step 1

Let us now introduce the variables p, q and VAL according to the DSW-idea. Using Fig. 3 as a guide this idea can be translated in the following invariant which the new algorithm should satisfy:

Invariant 5.9

Let $S = \langle X_1, \ldots, X_n \rangle$ and let $X_0 = X_{-1} = \text{null}$.

Let $V = \{\text{branch}(X_i, k(X_i)) \mid i = 1, \ldots, n - 1\}$.

Then

1. $p = \text{ref}(X_n)$.
2. $q = \text{ref}(X_{n-1})$.
3. $\forall\, i = 1, \ldots, n - 1\ [\text{VAL}(\text{branch}(X_i, k(X_i))) = \text{ref}(X_{i-1})]$.
4. $\forall\, X \in G\ \forall\, C \in \text{comp}(X)\ [C \notin V \Rightarrow \text{VAL}(C) = \text{val}(C)]$.

Note that as implied by this invariant the situation where S = <> corresponds to p = nil, q = nil and VAL = val.

Step 2

Assignments to the variables p, q and VAL must be added to Algorithm 5 in such a way that Invariant 5.9 is satisfied. First the variables should be initialized properly. VAL is implicitly assumed to be equal to val at

the beginning of the algorithm. The initialization therefore amounts to:

Addition 5.1

M, k(R), S := {R}, 0, <R> ⟶
    M, k(R), S, p, q := {R}, 0, <R>, ref(R), nil

Invariant 5.9 now holds initially. The only operations which disturb Invariant 5.9 are the operations which modify S: "S := push(S, Y)" and "S := pop(S)". Consequently these operations should be accompanied by modifications of p, q and VAL in order to restore Invariant 5.9.

Consider the operation "S := push(S, Y)" first. This operation makes Y the top element of S and X the subtop element. Hence the set of branches V in Invariant 5.9 is extended by this operation with branch(X, k(X)), which is denoted by B in Algorithm 5. This affects parts 1, 2 and 3 but not part 4 of Invariant 5.9. Part 1 can be restored by assigning to p the value ref(Y), which is equal to val(B). Part 4 of Invariant 5.9 implies, since $B \notin V$, that val(B) = VAL(B). Part 1 can therefore be restored by assigning to p the value VAL(B). Part 2 can be restored by assigning to q the value ref(X), which is equal to p. Finally part 3 can be restored by assigning to VAL(B) the reference of the object "below" X in S, i.e. the value q. (Notice that this assignment to VAL does not affect part 4 of Invariant 5.9). This leads to:

Addition 5.2

M, k(Y), S := M ∪ {Y}, 0, push(S, Y) ⟶
    M, k(Y), S, p, q, VAL(B) := M ∪ {Y}, 0, push(S, Y), VAL(B), p, q

The operation "S := pop(S)" removes the object X at the top of S from S. In order to investigate the way this operation affects Invariant 5.9 two cases must be distinguished: the case where S contains a single object and the case where S contains two or more objects. Consider the former first. If S contains only one object the set V in Invariant 5.9 is empty and will be so after the operation "S := pop(S)". This implies that parts 3 and 4 of Invariant 5.9 are not affected. Part 2 is neither affected because $ref(X_0) = ref(X_{-1}) = nil$. Only part 1 must be restored which can be done by

assigning the value ref(X_0) = nil to p. This covers the first case.

In the second case S contains two or more objects and consequently $V \neq \emptyset$. Let Y be the subtop element of S, i.e. the object referred to by q and let B = branch(Y, k(Y)), then $B \in V$. The effect of the operation "S := pop(S)" on V is that B is removed from V. This does not affect part 3 of Invariant 5.9 (n decreases by one). It does affect parts 1, 2 and 4 though. Part 1 can be restored by assigning to p the value ref(Y), which is equal to q. Part 2 can be restored by assigning to q the value ref(Z), where Z is the (possibly imaginary) object below Y in S. Part 3 of Invariant 5.9 implies that ref(Z) = VAL(branch(Y, k(Y))) = VAL(B). So part 2 can be restored by assigning the value VAL(B) to q. Remains part 4. This part of the invariant is disturbed because B is removed from V and the assertion VAL(B) = val(B) is not guaranteed to hold. As a consequence part 4 can be restored by assigning the value val(B) to VAL(B). (Notice that this does not affect part 3 of Invariant 5.9). According to part 2 of Invariant 5.8, val(B) = val(branch(Y, k(Y))) = ref(X) = p. So part 3 of Invariant 5.9 can be restored by assigning the value p to VAL(B).

Immediately before the operation "S := pop(S)" in Algorithm 5 the assertion $S \neq <>$ holds. This implies that the distinction between the two cases considered above can be made by testing whether q = nil or not (see Invariant 5.9). All in all this amounts to:

Addition 5.3

```
S := pop(S) ⟶
     If q = nil
     |  S, p := pop(S), nil.
     else
     |  Let Y = obj(q).
     |  Let B = branch(Y, k(Y)).
     |  S, p, q, VAL(B) := pop(S), q, VAL(B), p
```

The algorithm obtained through the above additions to Algorithm 5 is given below. Though we made sure Invariant 5.9 is satisfied (not only as a loop invariant, but "everywhere"), a formal proof is still required. This proof will be obvious now and is omitted.

Algorithm 5a

```
M, k(R), S, p, q := {R}, 0, <R>, ref(R), nil.
Until S = <>
| Let X = top(S).
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X, k(X)).
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M, k(Y), S, p, q, VAL(B) := M U {Y}, 0, push(S, Y), VAL(B), p, q.
| else
| | If q = nil
| | | S, p := pop(S), nil.
| | else
| | | Let Y = obj(q).
| | | Let B = branch(Y, k(Y)).
| | | S, p, q, VAL(B) := pop(S), q, VAL(B), p.
```

Before removing S it should be proved that the effect of the algorithm on VAL is nil. In other words, it must be proved that the postcondition VAL = val holds. Proof: at termination of the algorithm S = <>, which implies that V = ∅ in Invariant 5.9, which implies that VAL = val according to part 4 of Invariant 5.9.

Step 3

In this step the invariants will be applied so as to eliminate S and val from Algorithm 5a through replacements. Invariant 5.9 part 1 implies that the assertion S = <> is equivalent to p = nil, which results in:

Replacement 5.1

S = <> ⟶

p = nil

Invariant 5.9 part 1 also implies that, if S ≠ <>, top(S) = obj(p). This gives us:

Replacement 5.2

Let X = top(S) ⟶

    Let X = obj(p)

Immediately after the statement "Let B = branch(X, k(X))" the assertion B ∉ V holds. From part 4 of Invariant 5.9 (which also holds there) can be inferred that this implies that val(B) = VAL(B), which justifies:

Replacement 5.3

Let Y = obj(val(B)) ⟶

    Let Y = obj(VAL(B))

The application of the above replacements transform Algorithm 5a into an algorithm in which val no longer occurs and in which S has become a redundant variable. The assignments to S can now be removed:

Removal 5.1

S, p, q, VAL(B) := pop(S), q, VAL(B), p ⟶

    p, q, VAL(B) := q, VAL(B), p

Removal 5.2

S, p := pop(S), nil ⟶

    p := nil

Removal 5.3

M, k(Y), S, p, q, VAL(B) := M U {Y}, 0, push(S, Y), VAL(B), p, q ⟶

    M, k(Y), p, q, VAL(B) := M U {Y}, 0, VAL(B), p, q

Removal 5.4

M, k(R), S, p, q := {R}, 0, <R>, ref(R), nil ⟶

    M, k(R), p, q := {R}, 0, ref(R), nil

Step 4

In this step S will be removed from the invariants. Though in the previous steps the constant function val was removed from the algorithm together with S, this function need (and should) not be removed from the invariants (val is part of the problem specification). In contrast with the previous two phases the rewriting of the invariants containing S so as to eliminate S is far from obvious. Therefore the invariants will not be rewritten and an existential quantifier will be used to "eliminate" S. The final algorithm of this phase and of the entire derivation, the DSW-algorithm, is given below together with its invariants, pre- and postconditions. Strictly speaking the invariants are superfluous now, but they could be used for an independent proof of correctness, if desired.

Algorithm 6 (Deutsch-Schorr-Waite)

```
M, k(R), p, q := {R}, 0, ref(R), nil.
Until p = nil
| Let X = obj(p).
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X, k(X)).
| | Let Y = obj(VAL(B)).
| | If Y ∉ M
| | | M, k(Y), p, q, VAL(B) := M U {Y}, 0, VAL(B), p, q.
| else
| | If q = nil
| | | p := nil.
| | else
| | | Let Y = obj(q).
| | | Let B = branch(Y, k(Y)).
| | | p, q, VAL(B) := q, VAL(B), p.
```

Preconditions

6.1. VAL = val.

Invariants

6.1. $R \in M$.

6.2. $\forall X \in M$ [X is reachable].

6.3. $\forall X \in M$ [$0 \leq k(X) \leq$ degree(X)].

6.4. $\forall X \in M \; \forall i = 1, \ldots, k(X)$ [obj(val(branch(X, i))) $\in M$].

6.5. There is a stack of objects $S = \langle X_1, \ldots, X_n \rangle$ such that

6.5.1. $S \subset M$.

6.5.2. $\forall X \in M \setminus S$ [$k(X) =$ degree(X)].

6.5.3. All elements of S are different.

6.5.4. $\forall i = 1, \ldots, n - 1$ [$k(X_i) > 0$].

6.5.5. $\forall i = 1, \ldots, n - 1$ [$val(branch(X_i, k(X_i))) = ref(X_{i+1})$].

6.5.6. Let $X_0 = X_{-1} =$ null.

Let $V = \{branch(X_i, k(X_i)) \mid i = 1, \ldots, n - 1\}$.

Then

6.5.6.1. $p = ref(X_n)$.

6.5.6.2. $q = ref(X_{n-1})$.

6.5.6.3. $\forall i = 1, \ldots, n - 1$ [$VAL(branch(X_i, k(X_i))) = ref(X_{i-1})$].

6.5.6.4. $\forall X \in G \; \forall C \in comp(X)$ [$C \notin V \Rightarrow VAL(C) = val(C)$].

Postconditions

6.1. $M = \{X \in G \mid X \text{ is reachable}\}$.

6.2. VAL = val.

10. CONCLUSION

There are three different ways to look at the method of deriving algorithms described and demonstrated in this paper. The first is from the viewpoint of algorithm construction. Can the method be of any help in the process of constructing (deriving) a new algorithm? It would not be entirely fair to judge this from the derivation of the DSW-algorithm given above. We knew beforehand what target we were aiming at and carefully directed the derivation process in order to hit that target. In constructing a new algorithm the target is unknown. Yet the derivation

method described here is believed to be of help in deriving new algorithms too. The first reason is that performing global transformations in a stepwise way aids in retaining or even gaining insight in the algorithm under development, which may lead to the discovery of new useful transformations. The second reason is that the algorithm constructor is invited to try and perform a complex transformation, even if he has only some intuitive idea of it. He can cast his idea in a number of new variables and assertions on these variables, and start adding assignments to the variables and making replacements based on the assertions. If he does not achieve what he had in mind, too bad. If he does, he need only proof the assertions he postulated and remove whatever variables he made redundant.

The second way to look at the method is from the point of view of algorithm verification. The method constitutes a simple way to proof the correctness of global transformations which amount to changes of data representation. The correctness of such a transformation is proved by decomposing the transformation into a sequence of simple and evidently correct transformations. No comprehensive catalogue of transformation rules is required, nor the use of an "abstraction function" as in [7]. The method is also very flexible in that it allows very complex changes of representation (such as the DSW-transformation) to be proved correct without the need for enhanced verification techniques.

In relation to the above it is interesting to compare the correctness proof of the DSW-algorithm given here with other proofs of correctness of the DSW-algorithm [5, 8, 11, 13]. The first thing to be noted is that all of the latter were proofs of more or less simplified versions of the DSW-algorithm instead of the general DSW-algorithm considered here. The second thing to be noted is that in [5, 8, 11, 13] the DSW-algorithm is considered as a given algorithm which is proved correct "independently". Here the DSW-algorithm is proved correct by proving a simple abstract algorithm correct and deriving the DSW-algorithm through a number of correctness-preserving transformations from this algorithm. In fact we proved the correctness of a number of algorithms (Algorithms 1 - 6). Consequently the proof given here is much longer than the proofs in [5, 8, 11, 13]. We could have chosen Algorithm 5 (the stack algorithm) as a starting point, however.

The length of the proof would then have been comparable to the length of the proofs in [5, 8, 11, 13]. The advantage of the approach pursued here is, that the correctness proof is "factorized", which makes it more suitable for human consumption. The only similar approach to a correctness proof of the DSW-algorithm is [9], in which the outline of a correctness proof using the catalogue approach is given (only the intermediate algorithms are given). Apart from not being complete, the proof (the derivation) sketched there seems to be more complicated than the one given here.

The third way to consider the method described here is from the viewpoint of algorithm presentation. Presenting an algorithm by showing how it can be derived by a number of transformations from a simple algorithm adds considerably to understandability. This is an inherent advantage of the transformational method. It adds even more to understandability if not only the initial algorithm, but also all transformations applied to it are simple. The latter holds for the method described here. The transformational method in general is also very suitable for presenting classes of algorithms. Instead of walking to the DSW-algorithm straight ahead, we could have turned into several sideways in the derivation. If this is done in a systematic way, the entire class of marking algorithms can be discussed with a minimum of effort and a maximum of coherence. On a small scale and in a somewhat different context this was done in [2] for sorting algorithms. On a larger scale, using a more coarsely grained version of the method described here, this will be done in a survey of garbage collection algorithms (both marking and compaction algorithms) which I am currently working on .

REFERENCES

[1] BACK, R., On the correctness of refinement steps in program development, Ph.D. thesis, University of Helsinki, Helsinki (1978).

[2] DARLINGTON, J., A synthesis of several sorting algorithms, Acta Informatica 11 (1978), 1-30.

[3] DARLINGTON, J., Program transformation: an introduction and survey, Computer Bulletin 2, 22 (1979), 22-24.

[4] GERHART, S.L., Correctness-preserving program transformations, Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto (1975), 54-66.

[5] GRIES, D., The Schorr-Waite graph marking algorithm, Acta Informatica 11 (1979), 223-232.

[6] HOARE, C.A.R., An axiomatic basis for computer programming, Communications of the ACM 12 (1969), 576-580.

[7] HOARE, C.A.R., Proof of correctness of data representations, Acta Informatica 1 (1972), 271-281.

[8] KOWALTOWSKI, T., Data structures and correctness of programs, Journal of the ACM 26 (1979), 283-301.

[9] LEE, S., W.P. DE ROEVER & S.L. GERHART, The evolution of list-copying algorithms and the need for structured program verification, Conference Record of the Sixth ACM Symposium on Principles of Programming Languages, San Antonio (1979), 53-67.

[10] MEERTENS, L.G.L.T., Abstracto 84: the next generation, Proceedings of the 1979 Annual Conference of the ACM, Detroit (1979), 33-39.

[11] ROEVER, W.P. DE, On backtracking and greatest fixpoints, in: Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.), North-Holland Publishing Company (1978), 621-639.

[12] SCHORR, H. & W.M. WAITE, An efficient machine-independent procedure for garbage collection in various list structures, Communications of the ACM 10 (1967), 501-506.

[13] TOPOR, R.W., The correctness of the Schorr-Waite list marking algorithm, Acta Informatica 11 (1979), 211-221.