The Deutsch-Schorr-Waite graph marking algorithm
(mainly for my own understanding)

0. Reachability

We consider a finite set, the elements of which we call nodes. We assume the existence of a (binary) relation on the set of nodes; we denote this relation by the infix operator $\rightarrow$. In what follows, A is a given node, dummies x, y, z range over the nodes, and predicates R, Y, Z are boolean functions on the set of nodes.

We wish to derive an algorithm for the computation of the set of all nodes that are "reachable from A". We postulate that these are precisely those nodes that satisfy predicate R defined by:

(0) $R.A$

(1) $(\forall x,y : x \rightarrow y : R.x \Rightarrow R.y)$

(2) R is the strongest of all predicates satisfying (0) $\wedge$ (1)

Our algorithm will have as postcondition $Z = R$, where Z is a variable of type "predicate". Using (0), (1), and (2), we rewrite this postcondition as follows:

(3) $Z.A$

(4) $(\forall x,y : x \rightarrow y : Z.x \Rightarrow Z.y)$

(5) $[\ Z \Rightarrow R\ ]$

The conjunction of these three formulae does not admit a simple and obvious initialisation of Z (, although (3) ∧ (4) admits Z = true and (4) ∧ (5) admits Z = false). Therefore, we weaken the post-condition by replacing one of Z's occurrences by a new variable Y ; for this, the left-most occurrence of Z in (4) is a good candidate. Thus, we obtain the following invariant for a repetition:

P0 : Z.A
P1 : (∀x,y : x→y : Y.x ⇒ Z.y)
P2 : [Z ⇒ R]
P3 : [Y ⇒ Z]

The inclusion of P3 is, in this stage, not obvious; the development of the following program, however, shows its desirability. To shorten the presentation we have included P3 right away.

From P0 ∧ P1 ∧ P2 and Z=Y we now conclude Z=R. From Z≠Y and P3 we conclude the existence of a node x satisfying

¬ Y.x ∧ Z.x

(note: P3 eliminates the awkward case Y.x ∧ ¬Z.x .)
By Y.x := true the number of nodes satisfying ¬Y decreases; by P1, this assignment has precondition:

(∀y : x→y : Z.y)

If, for some node y, we have x→y ∧ ¬Z.y, then Z.y := true decreases the number of nodes

satisfying $\neg Z$; the precondition of this assignment is, by P_2, $R.y$, which follows from $x \to y \wedge R.x$ (by (1)), which in turn follows from $x \to y \wedge Z.x$ (by P_2). Moreover, the assignment $Z.y := true$ does not falsify $Z \neq Y$; so, we can use a nested repetition to establish $(\forall y : x \to y : Z.y)$. Thus, we obtain our first program — in which $\emptyset$ denotes the constant predicate with value false and in which $\{A\}$ denotes the point predicate that is true in A only—.

program 0

$$
\begin{array}{l}
\quad Y, Z := \emptyset, \{A\} \\
\{\ \underline{\text{invariant}}: \ P_0 \wedge P_1 \wedge P_2 \wedge P_3 \ ; \ \underline{\text{vf}}: (\#x :: \neg Y.x)\ \} \\
; \ \underline{\text{do}}\ Y \neq Z \to [\![\ x : \neg Y.x \wedge Z.x \\
\qquad \{\ \underline{\text{invariant}}: P_0 \wedge P_1 \wedge P_2 \wedge P_3 \wedge \neg Y.x \wedge Z.x \\
\qquad\ \wedge Y \neq Z \ ; \ \ \underline{\text{vf}}: (\#y :: \neg Z.y) \qquad\qquad \} \\
\qquad ; \ \underline{\text{do}}\ (\exists y : x \to y : \neg Z.y) \\
\qquad\quad \to [\![\ y : x \to y \wedge \neg Z.y \\
\qquad\qquad \{\ \neg Y.y \wedge \neg Z.y\ \} \\
\qquad\qquad ; \ Z.y := true \\
\qquad\qquad \{\ \neg Y.y \wedge Z.y \quad (\underline{\text{note}}, \text{see below})\ \} \\
\qquad\qquad ; \ \ x := y \\
\qquad\qquad]\!] \\
\qquad \underline{\text{od}} \\
\qquad \{\ (\forall y : x \to y : Z.y)\ \} \\
\qquad ; \ Y.x := true \\
\qquad]\!] \\
\quad \underline{\text{od}}
\end{array}
$$

□

note : By P3 we have $\neg Z.y \Rightarrow \neg Y.y$; hence, $Z.y := true$ has precondition $\neg Y.y \wedge \neg Z.y$ and postcondition $\neg Y.y \wedge Z.y$; this postcondition justifies the correctness of the assignment $x := y$. The statement $x := y$ is logically superfluous but its presence changes the operational characteristics of the program drastically . This is crucial for the transformations performed in the following sections . With the statement $x := y$ program 0 is known under the name "depth-first search" .

□

1. Implementation of $x : \neg Y.x \wedge Z.x$

The set of nodes satisfying $\neg Y \wedge Z$ can be represented by a list s , say , which happens to be used as a stack . List s is coupled to Y and Z by the following representation invariants :

P4: all elements of s are different

P5: $(\forall z :: z \in s \equiv \neg Y.z \wedge Z.z)$.

P4 and P5 are invariants of both repetitions . Variable Y now becomes superfluous and can be eliminated . (By P3 and P5 we have $Y = Z \equiv s = []$.) Thus, we obtain a new version of our program . Notice that, due to the assignment $x := y$, the inner repetition maintains $x = s.0$; as a result the removal of x from s boils down to $s := s\downarrow 1$.

program 1

```
s , Z := [A] , {A}
; do s ≠ [] → ||[ x := s.0
              ; do (∃y : x→y : ¬Z.y)
                   → ||[y : x→y ∧ ¬Z.y
                       ; Z.y := true ; s := y:s
                       ; x := y
                       ]||
                od
              ; s := s↓1
              ]||
   od
□
```

2. Implementation of $(\exists y : x \to y : \neg Z.y)$

We simplify the complicated guard $(\exists y : x \to y : \neg Z.y)$ by weakening it ; this is safe as long as we can still guarantee termination of the repetition. We define

$$N_x = (\#y :: x \to y) \quad , \text{ for all nodes } x ,$$

and per node z in s we introduce an integer variable n_z satisfying :

P6 : $(\forall z : z \in s : 0 \leq n_z \leq N_z)$

The idea is that $(\#y : x \to y : Z.y) \geq n_x$ will be invariant as well ; hence we will have :

$$n_x = N_x \;\Rightarrow\; (\forall y : x \to y : Z.y)$$

For this purpose we number the elements of the set $\{ y : x \to y : y \}$ — the "direct successors" of x — consecutively from 1. We denote x's successor with number i, $0 < i \leq N_x$, by $x!i$; so, we have (by definition):

$$\{ y : x \to y : y \} = \{ i : 0 < i \leq N_x : x!i \}, \text{ for all } x.$$

The required $(\#y : x \to y : Z.y) \geq n_x$ now follows from the invariance of:

$$P_7 : \quad (\forall z, i : z \in s \wedge 0 < i \leq n_z : Z.(z!i))$$

For the inner repetition we need a new variant function, namely:

$$(\Sigma z : z \in s : N_z - n_z) + (\Sigma z : z \notin s : N_z)$$

(If we set $n_z = 0$ for all z, $z \notin s$, this can be simplified to $(\Sigma z :: N_z - n_z)$.)

The guard $(\exists y : x \to y : \neg Z.y)$ can now be weakened to $n_x \neq N_x$ and the assignment $y : x \to y \wedge \neg Z.y$ can be "refined" to $y := x!n_x$; because this does not admit $\neg Z.y$ as postcondition, some case analysis is unavoidable.

program 2

```
s, Z, n_A := [A], {A}, 0
; do s ≠ [] → ||[ x := s.0
              ; do n_x ≠ N_x
                   →  n_x := n_x + 1
                   ; ||[ y := x!n_x
                      ; if ¬ Z.y → Z.y := true
                                 ; s, n_y := y:s, 0
                                 ; x := y
                         [] Z.y → skip
                      fi
                   ]||
              od
              ; s := s↓1
              ]||
od
□
```

3. The Deutsch-Schorr-Wait representation of s

The crux of the DSW-algorithm follows from the desire to use as little additional storage as possible. In particular, list s, which can become as long as the total number of nodes, is a candidate for efficient representation. (The variables n_x require only a few bits per variable, if the number of successors per node is sufficiently small.)

From program 2 it follows that:

$\{\ x = s.0 \ \wedge\ y = x!n_x\ \}$

$s := y : s$

$\{\ y = s.0 \ \wedge\ x = s.1 \ \wedge\ y = x!n_x$, hence:

$s.0 = s.1!n_{s.1}$

$\}$

So, by its construction s satisfies

P8: $(\forall i : 0 < i < \#s : s.(i-1) = s.i!n_{s.i})$

Initially, P8 holds because initially $\#s = 1$. So, P8 is invariant. (Obviously, $s := s\downarrow 1$ maintains P8.)

In the DSW-algorithm the redundant information in P8 is exploited in a particular representation of the graph — i.e.: the set of nodes and their successors —, as follows. For each node z there is an array variable $c_z(j : 0 < j \leq N_z)$ satisfying initially and finally:

(6) $(\forall z,j : 0 < j \leq N_z : c_z.j = z!j)$

That is, c_z represents the successors of z. During the computation, this representation is temporarily modified. That is, as an invariant of the program's repetitions, (6) is <u>replaced</u> by P9 $\wedge$ P10 $\wedge$ P11:

P9: $(\forall z,j : 0 < j \leq N_z \ \wedge\ j \neq n_z : c_z.j = z!j)$

P10: $(\forall z : z \notin s\downarrow 1 : c_z.n_z = z!n_z)$

P11: $(\forall i,z : 0 < i < \#s \ \wedge\ z = s.i : c_z.n_z = s.(i+1))$

These invariants express that list s is represented by a modification of the arrays c, in the following way. Because, by P8, $s.i \,!\, n_{s.i} = s.(i-1)$, for $i : 0<i<\#s$, array element $c_z.n_z$, where $z = s.i$, can be used to record $s.(i+1)$ without destruction of information. This is formalised in P11. Invariants P9 and P10 express that all other elements of the arrays c remain unaffected. In order that, in P11, the expression $s.(i+1)$ is also meaningful for $i = \#s-1$, we define $s.(\#s) = \underline{nil}$ where $\underline{nil}$ is a value —also called a virtual root— satisfying:

(7) $\quad (\forall z :: z \neq \underline{nil})$

List s is now completely determined by $s.0$ and $s.1$ and P11. So, apart from the arrays c we only need two additional variables p and q (say) to represent $s.0$ and $s.1$, or $s.0$ only in the case where $s = []$ and, hence, $s.0 = \underline{nil}$:

P12: $\quad p = s.0 \;\wedge\; (s = [] \vee q = s.1)$

Notice that, by (7), we have $s = [] \equiv p = \underline{nil}$. Moreover, $P9 \wedge P10 \wedge s = [] \Rightarrow (6)$: upon termination the graph has been restored.

For the implementation of $s := y:s$ the new representation has the following consequences. P9 does not depend on s; $P10(s := y:s) \Leftarrow P10$ because $z \notin (y:s)\downarrow 1 \Rightarrow z \notin s\downarrow 1$. Furthermore:

$$
\begin{array}{ll}
 & P11(s := y:s) \\
\equiv & \quad \{ \text{substitution} \}
\end{array}
$$

$$
\begin{array}{ll}
& (\forall i,z : 0<i<\#(y:s) \wedge z = (y:s).i : c_z.n_z = (y:s).(i+1)) \\
\equiv & \quad \{ \text{ dummy transformation } i := i+1 \text{ ; def. \# and : } \} \\
& (\forall i,z : 0 \le i < \#s \wedge z = s.i : c_z.n_z = s.(i+1)) \\
\equiv & \quad \{ \text{ range split ; def. } P11 \} \\
& (\forall z : z = s.0 : c_z.n_z = s.1) \wedge P11 \\
\equiv & \quad \{ \text{ 1-pt. rule , } P12 \} \\
& c_p.n_p = q \wedge P11 \ .
\end{array}
$$

Hence, $P11$ is maintained by the assignment $c_p.n_p := q$. Similarly, it can be easily shown that $P12$ is maintained by the (simultaneous) assignment $p, q := y, p$ where $y = p!n_p$ which, by $P10$ and $p \notin s\downarrow 1$, equals $c_p.n_p$. (Notice that the role of variable x in program 2 is now taken over by p.) So, the operation $s := y:s$ can be implemented as:

$$p, q, c_p.n_p := c_p.n_p, p, q$$

For the implementation of $s := s\downarrow 1$ we observe again that $P9$ does not depend on s, and for the calculation of $P10(s := s\downarrow 1)$ we calculate first:

$$
\begin{array}{ll}
& z \notin s\downarrow 2 \\
\equiv & \quad \{ \text{ brute force case analysis } \} \\
& (z \notin s\downarrow 2 \wedge z = s.1) \vee (z \notin s\downarrow 2 \wedge z \neq s.1) \\
\equiv & \quad \{ \text{ by } P4 : z = s.1 \Rightarrow z \notin s\downarrow 2 \} \\
& z = s.1 \vee (z \notin s\downarrow 2 \wedge z \neq s.1) \\
\equiv & \quad \{ z\downarrow 1 = s.1 : s\downarrow 2 \} \\
& z = s.1 \vee z \notin s\downarrow 1 \ .
\end{array}
$$

Hence, we have (assuming $\#s \geq 2$):

$P10\ (s := s{\downarrow}1)$

$\equiv$ { substitution ; $s{\downarrow}1{\downarrow}1 = s{\downarrow}2$ }

$(\forall z : z \notin s{\downarrow}2 : c_z \cdot n_z = z!n_z)$

$\equiv$ { see above ; range split ; $s.1 = q$ (by P12) }

$c_q \cdot n_q = q!n_q \wedge (\forall z : z \notin s{\downarrow}1 : c_z \cdot n_z = z!n_z)$

$\equiv$ { $s.1 = q$, hence $q!n_q = p$ (by P8 and P12) }

$c_q \cdot n_q = p \wedge P10$.

So, P10 is maintained by the assignment $c_q \cdot n_q := p$.
Furthermore :

$P11\ (s := s{\downarrow}1)$

$\equiv$ { substitution }

$(\forall i,z : 0 < i < \#(s{\downarrow}1) \wedge z = (s{\downarrow}1).i : c_z \cdot n_z = (s{\downarrow}1).(i+1))$

$\equiv$ { $(s{\downarrow}1).j = s.(j+1)$ and $\#(s{\downarrow}1) = \#s - 1$ }

$(\forall i,z : 0 < i < \#s - 1 \wedge z = s.(i+1) : c_z \cdot n_z = s.(i+2))$

$\equiv$ { dummy transformation $i := i-1$ }

$(\forall i,z : 1 < i < \#s \wedge z = s.i : c_z \cdot n_z = s.(i+1))$

$\Leftarrow$ { weakening the range }

P11 .

Finally :

$P12\ (s := s{\downarrow}1)$

$\equiv$ { substitution }

$p = (s{\downarrow}1).0 \wedge (s{\downarrow}1 = [] \vee q = (s{\downarrow}1).1)$

$\equiv$ { $(s{\downarrow}1).j = s.(j+1)$ }

$p = s.1 \wedge (s{\downarrow}1 = [] \vee q = s.2)$.

$\equiv$ { P11 }

$p = s.1 \wedge (s{\downarrow}1 = [] \vee q = c_{s.1} \cdot n_{s.1})$.

Hence, P12 is maintained by $p, q := q, c_q \cdot n_q$,

provided that $s\downarrow 1 \neq []$; in the case that $s\downarrow 1 = []$, we have $\#s = 1$ and, hence, $q = \underline{nil}$. The case analysis $s\downarrow 1 \neq [] \ \vee \ s\downarrow 1 = []$ can be avoided by the introduction of an additional arrayvariable $c_{\underline{nil}}\,(j : 0 < j \leq 1)$ and an additional integer variable $n_{\underline{nil}}$, initially $n_{\underline{nil}} = 1$. Then, for both cases, the operation $s := s\downarrow 1$ can be implemented as:

$$p, q, c_q.n_q := q, c_q.n_q, p$$

Thus, we obtain the following, and final, version of our program. Notice that the initial value of $c_{\underline{nil}}$ is irrelevant.

program 3

$Z, n_A, n_{\underline{nil}}, p, q := \{A\}, 0, 1, A, \underline{nil}$
; do $p \neq \underline{nil} \rightarrow \{ p = s.0 \}$
  do $n_p \neq N_p$
    $\rightarrow \ n_p := n_p + 1$
    ; [[$y := c_p.n_p$
      ; if $\neg Z.y \rightarrow Z.y := true$; $n_y := 0$
        ; $p, q, c_p.n_p := c_p.n_p, p, q$
      [] $Z.y \rightarrow$ skip
      fi
    ]]
  od
  ; $p, q, c_q.n_q := q, c_q.n_q, p$
od

□

4. Epilogue

It is no surprise that the last stage of the design is the most laborious one ; in the last stage the most amount of detail is introduced. This exercise shows once more that systematic and manageable derivations of programs involving pointer manipulations are feasible, provided that the introduction of such pointer manipulations is postponed as long as possible and provided that no pictures are drawn. In the case of graph algorithms this boils down to the rule not to let the actual representation of the graph, as a datastructure, enter the discussion in too early a stage of the design.

Eindhoven, 9 march 1992
Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven