Guarded Commands, Nondeterminancy, and Formal Derivation of Programs

E.W. Dijkstra, CACM 18,8 pp.453-457, (Aug. 1975).
A Discipline of Programming, *Prentice-Hall, 1976.*

Concurrent assignment *vs* ***Sequential assignment***

$x, y := y, x$

$x := y; y := x$

$y := x; x := y$

$t := x; x := y; y := t$

$x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n$ *Concurrent assignment statement*

$S_1; S_2; \ldots; S_n$ *Statements List(SL)*

Two ***separators*** *, and ; has* ***different*** *meanings*

Two statements made from guarded commands

guarded_command **::=** *guard* → *guarded_list*
guard **::=** *boolean_expression*
guarded_list **::=** *statement* **{ ;** *statement* **}**
guarded_command_set **::=** *guarded_command* **{ |** *guarded_command* **}**
alternative_construct **::= *if*** *guarded_command_set* ***fi***
repeatative_construct **::= *do*** *guarded_command_set* ***od***
statement **::=** *alternative_construct* **|** *repeatative_construct*
| *concurrent_assignment_statement* **| *skip* | *abort* |** ...

Syntax of ***alternatve*** *and* ***repeatative*** *statements*

if $B_1 \rightarrow SL_1 \mid B_2 \rightarrow SL_2 \mid \ldots \mid B_n \rightarrow SL_n$ ***fi***
do $B_1 \rightarrow SL_1 \mid B_2 \rightarrow SL_2 \mid \ldots \mid B_n \rightarrow SL_n$ ***od***

if fi ≡ ***abort***
do od ≡ ***skip***

Alternative statement and Nondeterminancy

if $x \geq y \rightarrow m := x$
| $x \leq y \rightarrow m := y$
fi.

Nondeterminancy of alternative statement

if $x = y \rightarrow$ "$m := x$" $\vee$ "$m := y$"

Formal Derivation of Programs

$m = max(x, y)$

$R(m, x, y) = ((m = x) \vee (m = y)) \wedge (m \geq x) \wedge (m \geq y)$.

The assignment statement "$m := x$" will make ***true*** *for* $(m = x)$.

weakest preconditon to make R is true after the statement "$m := x$".

$wp(\text{"}m := x\text{"}, R) \equiv R(x, x, y) = (x = x \vee x = y) \wedge (x \geq x) \wedge (x \geq y) = x \geq y$.

if $x \geq y \rightarrow m := x$ ***fi***

$wp(\text{"}m := y\text{"}, R) \equiv R(y, x, y) = (y = x \vee y = y) \wedge (y \geq x) \wedge (y \geq y) = y \geq x$.

if $y \geq x \rightarrow m := y$ ***fi***

if $x \geq y \rightarrow m := x$
$\mid y \geq x \rightarrow m := y$
fi.

Repeatative statement

$q_1, q_2, q_3, q_4 := Q_1, Q_2, Q_3, Q_4;$

do $q_1 > q_2 \rightarrow q_1, q_2 := q_2, q_1$

| $q_2 > q_3 \rightarrow q_2, q_3 := q_3, q_2$

| $q_3 > q_4 \rightarrow q_3, q_4 := q_4, q_3$

od.

do od ≡ **skip**

P *Loop invariance condition*

BB *There exists at least one guard that is true*

$$1 \le {}^{\exists} i \le n: B_i \equiv B_1 \vee B_2 \vee \ldots \vee B_n.$$

$P = 1 \le {}^{\forall} i \le 4$: *$q_i$'s are permutation of Q_i*

$$\neg BB = \neg((q_1 > q_2) \vee (q_2 > q_3) \vee (q_3 > q_4))$$
$$= \neg(q_1 > q_2) \wedge \neg(q_2 > q_3) \wedge \neg(q_3 > q_4))$$
$$= (q_1 \le q_2) \wedge (q_2 \le q_3) \wedge (q_3 \le q_4) = (q_1 \le q_2 \le q_3 \le q_4)$$

$P \wedge \neg BB$.

Given two positive numbers X and Y, find x .э. x = gcd(X, Y)
Loop invariance P: introduce two local variables x and y.
P: [gcd(X, Y) = gcd(x, y)] ∧ (x > 0) ∧ (y > 0)].

Initialization of the loop invariance P
x, y := X, Y

Do something under the loop invariance of P
gcd(x, y) = gcd(x–y, y) = gcd(x, y–x) = gcd(y, x) = ...

wp("x := x–y", P) = [gcd(X, Y) = gcd(x–y, y)] ∧ (x–y > 0) ∧ (y > 0)
= x > y.
do *x > y → x := x–y* ***od***.

wp("y := y–x", P) = [gcd(X, Y) = gcd(x, y–x)] ∧ (x > 0) ∧ (y–x > 0)
= y > x.
do *y > x → y := y–x* ***od***.

```
x, y := X, Y;
do x > y → x := x–y
  | y > x → y := y–x
od
```

```
x :=X; y := Y;
while x ≠ y do if x > y then x := x–y          Ver. A
else y := y–x fi od
while x ≠ y do while x > y do x := x–y od      Ver. B
                    while y > x do y := y–x od
               od
```

```
if X > 0 and Y > 0 →
      x, y := X, Y;
      do x > y → x := x–y
        | y > x → y := y–x
      od
fi
```

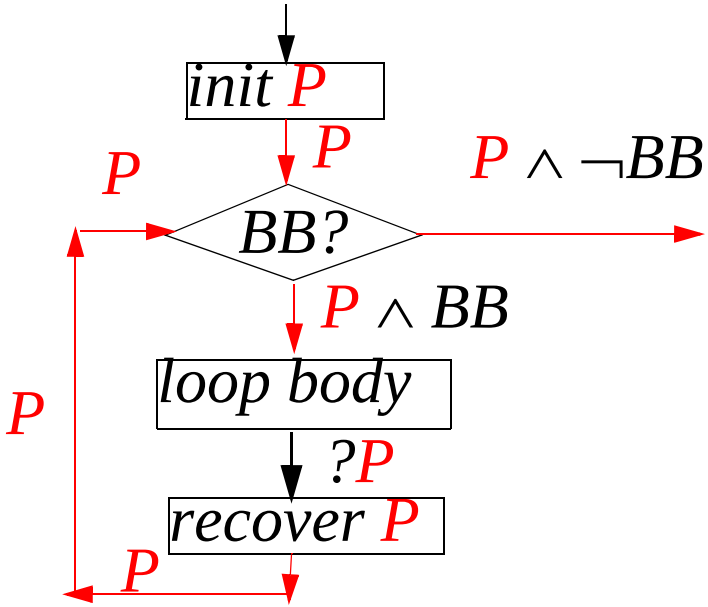
Does the loop ***terminate****?*

|x – y| is a non-negative monotonically decreasing integer function.

Loop terminates when x=y, i.e., |x – y| becomes zero.

*P is true before the loop(**initialization** of P),*
*P **should** remain true(**recover** P) in the loop(loop **invariance**), and*
*P is also true after the loop terminates(loop **terminating**).*
*P should be **weak** enough to be easily **initialized**, whereas*
*__strong__ enough to **fullfill** the final conditon after the loop by*
$P \wedge \neg BB$

for (init; test; update) loop_body
***syntactic sugar** in C*



Another Example

Sum, i := 0, 1;	$P \equiv (Sum = \sum_{k=1}^{i-1} k)$	***initialize** P*
***do** $i \leq 100 \rightarrow$*	$P \equiv (Sum = \sum_{k=1}^{i-1} k)$	*P is still **valid***
Sum := Sum + i;	$P' \equiv (Sum = \sum_{k=0}^{i} k)$	*P is **distroyed***
i := i +1	$P \equiv (Sum = \sum_{k=1}^{i-1} k)$	***recover** P*
od		

$$P \wedge \neg(i \leq 100) \equiv (Sum = \sum_{k=1}^{i-1} k) \wedge (i = 101)$$

$$= (Sum = \sum_{k=1}^{100} k) = (Sum = 5050).$$

101 – i is the nonnegative decreasing functon
loop termination