

An den Programmpunkten 1, 2 und 3 machen wir die Zusicherungen:

$$(1) \quad A \quad \equiv \quad x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$$

$$(2) \quad B \quad \equiv \quad x > 0 \wedge y > 0 \wedge r > x + y$$

$$(3) \quad \text{true}$$

Dann gilt:

$$A \Rightarrow r > 0 \quad \text{und} \quad B \Rightarrow r > x + y$$

Wir überprüfen:

$$\begin{aligned} \mathbf{WP} \llbracket x \neq y \rrbracket (\mathbf{true}, A) &\equiv x = y \vee A \\ &\Leftarrow x > 0 \wedge y > 0 \wedge r = x + y \\ &\equiv C \end{aligned}$$

Wir überprüfen:

$$\begin{aligned} \mathbf{WP}[\mathbf{x} \neq \mathbf{y}](\mathbf{true}, A) &\equiv x = y \vee A \\ &\Leftarrow x > 0 \wedge y > 0 \wedge r = x + y \\ &\equiv C \\ \mathbf{WP}[\mathbf{r} = \mathbf{x+y};](C) &\equiv x > 0 \wedge y > 0 \\ &\Leftarrow B \end{aligned}$$

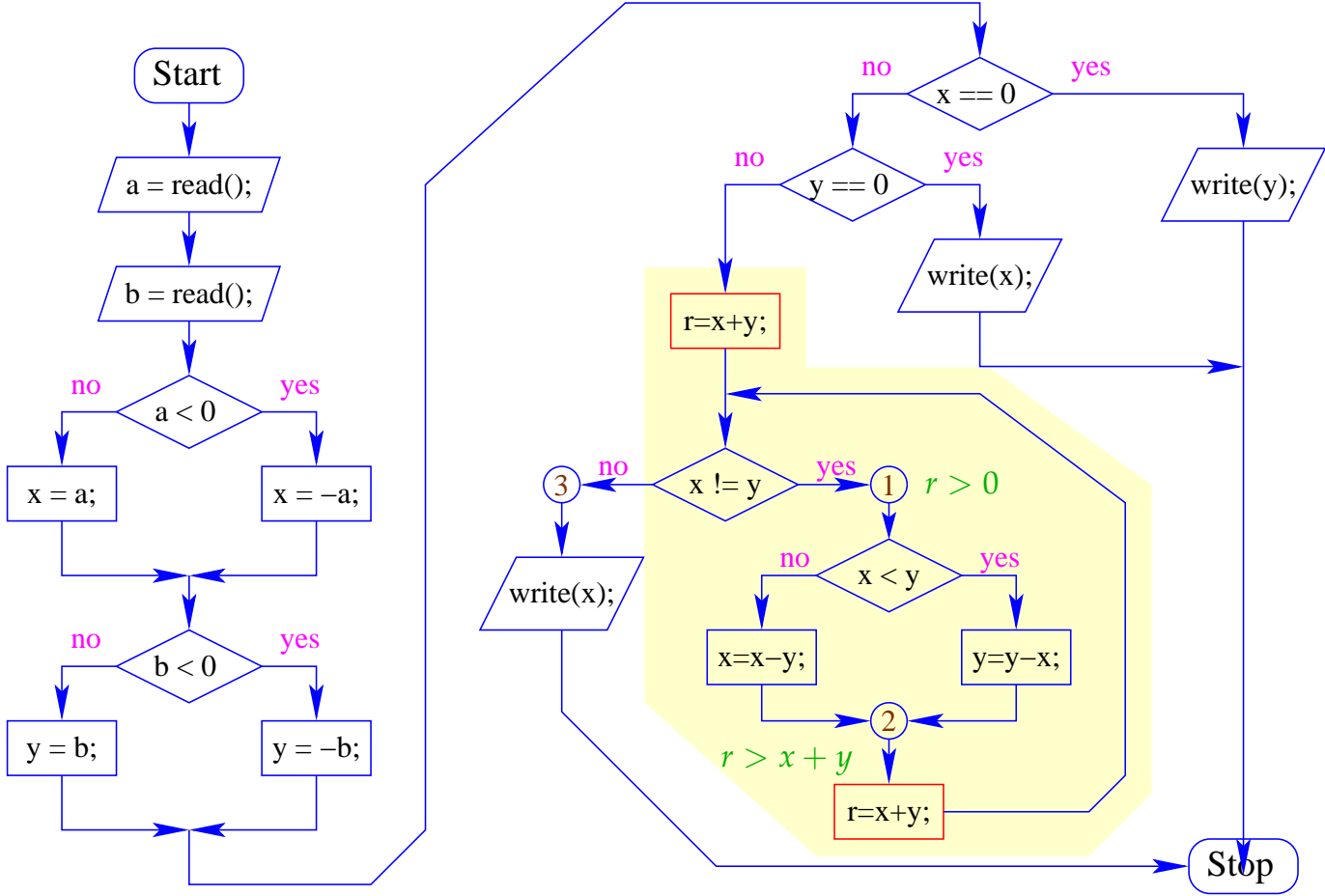
Wir überprüfen:

$$\begin{aligned} \mathbf{WP}[\mathbf{x} \text{ := } \mathbf{y}](\mathbf{true}, \mathbf{A}) &\equiv x = y \vee A \\ &\Leftarrow x > 0 \wedge y > 0 \wedge r = x + y \\ &\equiv C \\ \mathbf{WP}[\mathbf{r} = \mathbf{x+y};](\mathbf{C}) &\equiv x > 0 \wedge y > 0 \\ &\Leftarrow B \\ \mathbf{WP}[\mathbf{x} = \mathbf{x-y};](\mathbf{B}) &\equiv x > y \wedge y > 0 \wedge r > x \\ \mathbf{WP}[\mathbf{y} = \mathbf{y-x};](\mathbf{B}) &\equiv x > 0 \wedge y > x \wedge r > y \end{aligned}$$

Wir überprüfen:

$$\begin{aligned} \mathbf{WP}\llbracket x \neq y \rrbracket(\mathbf{true}, A) &\equiv x = y \vee A \\ &\Leftarrow x > 0 \wedge y > 0 \wedge r = x + y \\ &\equiv C \\ \mathbf{WP}\llbracket r = x+y; \rrbracket(C) &\equiv x > 0 \wedge y > 0 \\ &\Leftarrow B \\ \mathbf{WP}\llbracket x = x-y; \rrbracket(B) &\equiv x > y \wedge y > 0 \wedge r > x \\ \mathbf{WP}\llbracket y = y-x; \rrbracket(B) &\equiv x > 0 \wedge y > x \wedge r > y \\ \mathbf{WP}\llbracket y > x \rrbracket(\dots, \dots) &\equiv (x > y \wedge y > 0 \wedge r > x) \vee \\ &\quad (x > 0 \wedge y > x \wedge r > y) \\ &\Leftarrow x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y \\ &\equiv A \end{aligned}$$

Orientierung:



Weitere Propagation von C durch den Kontrollfluss-Graphen
komplettiert die lokal konsistente Annotation mit Zusicherungen
:-)

Weitere Propagation von C durch den Kontrollfluss-Graphen komplettiert die lokal konsistente Annotation mit Zusicherungen :-)

Wir schließen:

- An den Programmpunkten 1 und 2 gelten die Zusicherungen $r > 0$ bzw. $r > x + y$.
- In jeder Iteration wird r kleiner, bleibt aber stets positiv.
- Folglich wird die Schleife nur endlich oft durchlaufen
 \implies das Programm terminiert :-))

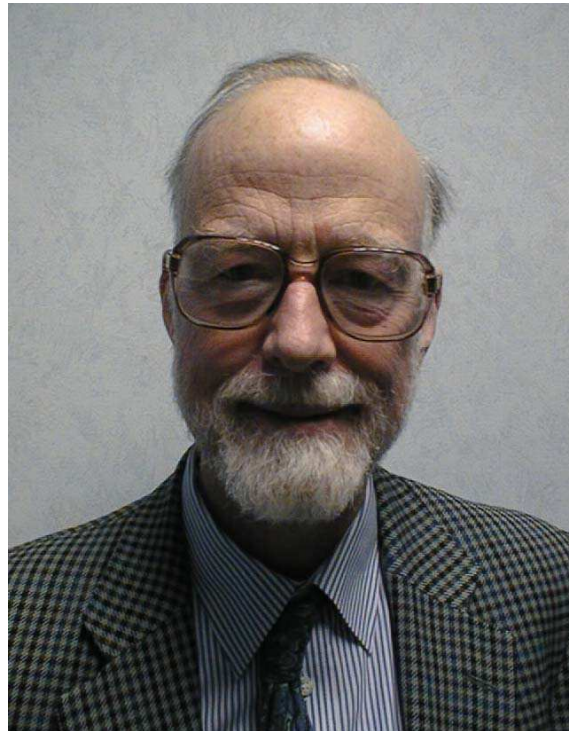
Allgemeines Vorgehen:

- Für jede vorkommende Schleife `while (b) s` erfinden wir eine neue Variable `r`.
- Dann transformieren wir die Schleife in:

```
r = e0;
while (b) {
    assert(r>0);
    s
    assert(r > e1);
    r = e1;
}
```

für geeignete Ausdrücke `e0, e1` :-)

3.5 Der Hoare-Kalkül



Tony Hoare, Microsoft Research, Cambridge

Idee (1):

- Organisiere den Korrektheitsbeweis entsprechend der Struktur des Programms !
- Modularisiere ihn so, dass Beweise für Teilprogramme zu einem Beweis für das gesamte Programm zusammen gesetzt werden können !

Idee (2):

Betrachte Aussagen der Form:

$$\{A\} \ p \ \{B\}$$

... das heißt:

Gilt **vor** der Ausführung des Programmstücks p Eigenschaft A
und terminiert die Programm-Ausführung, dann
gilt **nach** der Ausführung von p Eigenschaft B .

Idee (2):

Betrachte Aussagen der Form:

$$\{A\} \ p \ \{B\}$$

... das heißt:

Gilt **vor** der Ausführung des Programmstücks p Eigenschaft A und terminiert die Programm-Ausführung, dann gilt **nach** der Ausführung von p Eigenschaft B .

A : Vorbedingung

B : Nachbedingung

Beispiele:

$$\{x > y\} \quad z = x - y; \quad \{z > 0\}$$

Beispiele:

$\{x > y\}$ `z = x-y;` $\{z > 0\}$

$\{\mathbf{true}\}$ `if (x<0) x=-x;` $\{x \geq 0\}$

Beispiele:

$\{x > y\}$ `z = x-y;` $\{z > 0\}$

$\{\mathbf{true}\}$ `if (x<0) x=-x;` $\{x \geq 0\}$

$\{x > 7\}$ `while (x!=0) x=x-1;` $\{x = 0\}$

Beispiele:

$\{x > y\}$ `z = x-y;` $\{z > 0\}$

$\{\mathbf{true}\}$ `if (x<0) x=-x;` $\{x \geq 0\}$

$\{x > 7\}$ `while (x!=0) x=x-1;` $\{x = 0\}$

$\{\mathbf{true}\}$ `while (true);` $\{\mathbf{false}\}$

Ableitung korrekter Aussagen:

- Bestimme eine Menge von **Grundaussagen** oder **Axiome**, die gültige Tripel für einfache Anweisungen beschreiben.
- Gib **Regeln** an, wie aus bereits abgeleiteten gültigen Tripeln weitere gültige Tripel hergeleitet werden können ...

Axiome:

leere Anweisung: $\{B\}$; $\{B\}$

Zuweisung: $\{B[e/x]\}$ `x = e;` $\{B\}$

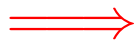
Eingabe: $\{\forall x. B\}$ `x = read();` $\{B\}$

Ausgabe: $\{B\}$ `write(e);` $\{B\}$

Regeln:

Komposition:
$$\frac{\{A\} s \{B\} \quad \{B\} s s \{C\}}{\{A\} s \quad s s \{C\}}$$

- Oberhalb des Strichs stehen die Voraussetzungen für die Regel-Anwendung.
- Unterhalb des Strichs steht die Schlussfolgerung :-)



Den Beweis eines Tripels für das zusammengesetzte Programm $s \quad s s$ setzen wir aus Beweisen für Aussagen über die Programmbestandteile s und $s s$ zusammen :-)

Beispiel:

Für: $x=x+2; z=z+x; i=i+1;$ konstruieren wir:

$$\begin{array}{l} \{z = (i+1)^2 \wedge x = 2i+1\} \text{ i=i+1; } \{z = i^2 \wedge x = 2i-1\} \\ \{z = i^2 \wedge x = 2i+1\} \text{ z=z+x; } \{z = (i+1)^2 \wedge x = 2i+1\} \\ \hline \{z = i^2 \wedge x = 2i-1\} \text{ x=x+2; } \{z = i^2 \wedge x = 2i+1\} \text{ z=z+x; i=i+1; } \{z = i^2 \wedge x = 2i-1\} \\ \hline \{z = i^2 \wedge x = 2i-1\} \text{ x=x+2; z=z+x; i=i+1; } \{z = i^2 \wedge x = 2i-1\} \end{array}$$

Regeln (Forts.):

Bedingung:
$$\frac{\{\neg b \wedge A\} \text{ ss0 } \{B\} \quad \{b \wedge A\} \text{ ss1 } \{B\}}{\{A\} \text{ if (b) } \{\text{ss0}\} \text{ else } \{\text{ss1}\} \{B\}}$$

- Diese Regel entspricht unserer bisherigen Behandlung bedingter Verzweigungen ...
- Falls $\{A_0\} \text{ ss0 } \{B\}$ und $\{A_1\} \text{ ss1 } \{B\}$ gelten, können wir A auch wählen als

$$A \equiv (b \Rightarrow A_0) \wedge (\neg b \Rightarrow A_0) \quad :-)$$

Beispiel:

Wir wollen

$$\{A\} \text{ if } (y > x) \text{ } y=y-x; \text{ else } x=x-y; \{B\}$$

beweisen für:

$$A \equiv x > 0 \wedge y > 0 \wedge x \neq y$$

$$B \equiv x > 0 \wedge y > 0$$

Dazu konstruieren wir:

$$\begin{array}{c} \frac{\{x > 0 \wedge y > x\} \quad y=y-x; \quad \{B\}}{\{A \wedge y > x\} \quad y=y-x; \quad \{B\}} \qquad \frac{\{x > y \wedge y > 0\} \quad x=x-y; \quad \{B\}}{\{A \wedge y \leq x\} \quad x=x-y; \quad \{B\}} \\ \hline \{A\} \quad \text{if } (y > x) \text{ } y=y-x; \text{ else } x=x-y; \quad \{B\} \end{array}$$

Regeln (Forts.):

$$\text{Schleife: } \frac{\{b \wedge I\} \text{ ss } \{I\}}{\{I\} \text{ while } (b) \{ss\} \{\neg b \wedge I\}}$$

- Die Zusicherung I entspricht unserer **Schleifen-Invariante** :-)
- Bei Betreten des Rumpfs wissen wir zusätzlich, dass die Bedingung b gilt :-)
- Bei Verlassen der Schleife wissen wir zusätzlich, dass die Bedingung b **nicht** mehr gilt :-))

Regeln (Forts.):

- Gelegentlich **passen** die Vor- bzw. Nachbedingungen der bereits als korrekt bewiesenen Tripel nicht genau zusammen ...
- Dann kann man die Vorbedingung **verstärken** oder die Nachbedingung **abschwächen**:

Abschwächung:

$$\frac{A \Rightarrow A_1 \quad \{A_1\} \text{ ss } \{B_1\} \quad B_1 \Rightarrow B}{\{A\} \text{ ss } \{B\}}$$

3.6 Behandlung von Prozeduren

Modulare Verifikation können wir benutzen, um die Korrektheit auch von Programmen mit Funktionen nachzuweisen :-)

Vereinfachung:

Wir betrachten nur

- Prozeduren, d.h. statische Methoden ohne Rückgabewerte;
- nur globale Variablen, d.h. alle Variablen sind ebenfalls `static`.

// werden wir später verallgemeinern :-)

Beispiel:

```
int a, b, x, y;

void main () {
    a = read();
    b = read();
    mm();
    write (x-y);
}

void mm() {
    if (a>b) {
        x = a;
        y = b;
    } else {
        y = a;
        x = b;
    }
}
```

Kommentar:

- Der Einfachheit halber haben wir alle Vorkommen von `static` gestrichen :-)
- Die Prozedur-Definitionen sind nicht rekursiv.
- Das Programm liest zwei Zahlen ein.
- Die Prozedur `minmax` speichert die größere in `x`, die kleinere in `y` ab.
- Die Differenz von `x` und `y` wird ausgegeben.
- Wir wollen zeigen, dass gilt:

$$\{a \geq b\} \text{ mm}(); \{a = x\}$$

Vorgehen:

- Für jede Prozedur $f()$ stellen wir ein Tripel bereit:

$$\{A\} f(); \{B\}$$

- Unter dieser **globalen Hypothese** H verifizieren wir, dass sich für jede Prozedurdefinition `void f() { ss }` zeigen lässt:

$$\{A\} \text{ ss } \{B\}$$

- Wann immer im Programm ein Prozeduraufruf vorkommt, benutzen wir dabei die Tripel aus $H \dots$