Exkurs: Aussagenlogik

Aussagen: “Alle Menschen sind sterblich”,

“Sokrates ist ein Mensch”, “Sokrates ist sterblich”

Exkurs: Aussagenlogik

Aussagen: “Alle Menschen sind sterblich”,

“Sokrates ist ein Mensch”, “Sokrates ist sterblich”

$\forall x.\, \mathsf{Mensch}(x) \Rightarrow \mathsf{sterblich}(x)$

$\mathsf{Mensch}(\mathsf{Sokrates}),\ \mathsf{sterblich}(\mathsf{Sokrates})$

Exkurs: Aussagenlogik

Aussagen: “Alle Menschen sind sterblich”,

“Sokrates ist ein Mensch”, “Sokrates ist sterblich”

$\forall\, x.\, \mathsf{Mensch}(x) \Rightarrow \mathsf{sterblich}(x)$

$\mathsf{Mensch}(\mathsf{Sokrates}),\ \mathsf{sterblich}(\mathsf{Sokrates})$

Tautologien: $A \vee \neg A$

$\forall\, x \in \mathbb{Z}.\, x < 0 \vee x = 0 \vee x > 0$

Exkurs: Aussagenlogik

Aussagen: “Alle Menschen sind sterblich”,

“Sokrates ist ein Mensch”, “Sokrates ist sterblich”

$$\forall\, x.\, \mathsf{Mensch}(x) \Rightarrow \mathsf{sterblich}(x)$$

$$\mathsf{Mensch}(\mathsf{Sokrates}),\ \mathsf{sterblich}(\mathsf{Sokrates})$$

Tautologien: $A \vee \neg A$

$$\forall\, x \in \mathbb{Z}.\, x < 0 \vee x = 0 \vee x > 0$$

Gesetze: $\neg\neg A \equiv A$

$$A \wedge A \equiv A$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$

$$A \Rightarrow B \equiv \neg A \vee B$$

Beispiel:

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
   j = read();
   while (j>0) { t = t+1; j = j-1; }
   i = i-1;
}
write(t);
```

- Die gelesene Zahl `i` (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-) die Summe der positiven für j gelesenen Werte

Beispiel:

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
   j = read();
   while (j>0) { t = t+1; j = j-1; }
   i = i-1;
}
write(t);
```

- Die gelesene Zahl `i` (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-) die Summe der positiven für j gelesenen Werte

$\Longrightarrow$ das Programm terminiert immer !!!

Programme nur mit `for`-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
```

// im Rumpf wird `i` nicht modifiziert

... terminieren ebenfalls immer :-))

Programme nur mit `for`-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
// im Rumpf wird i nicht modifiziert
```

... terminieren ebenfalls immer :-))

Frage:

Wie können wir aus dieser Beobachtung eine Methode machen, die auf beliebige Schleifen anwendbar ist ?

Idee:

- Weise nach, dass jede Scheife nur endlich oft durchlaufen wird ...
- Finde für jede Schleife eine Kenngröße r, die zwei Eigenschaften hat:

  (1) Wenn immer der Rumpf betreten wird, ist $r > 0$;

  (2) Bei jedem Schleifen-Durchlauf wird r kleiner :-)

- Transformiere das Programm so, dass es neben der normalen Programmausführung zusätzlich die Kenngrößen r mitberechnet.
- Verifiziere, dass (1) und (2) gelten :-)

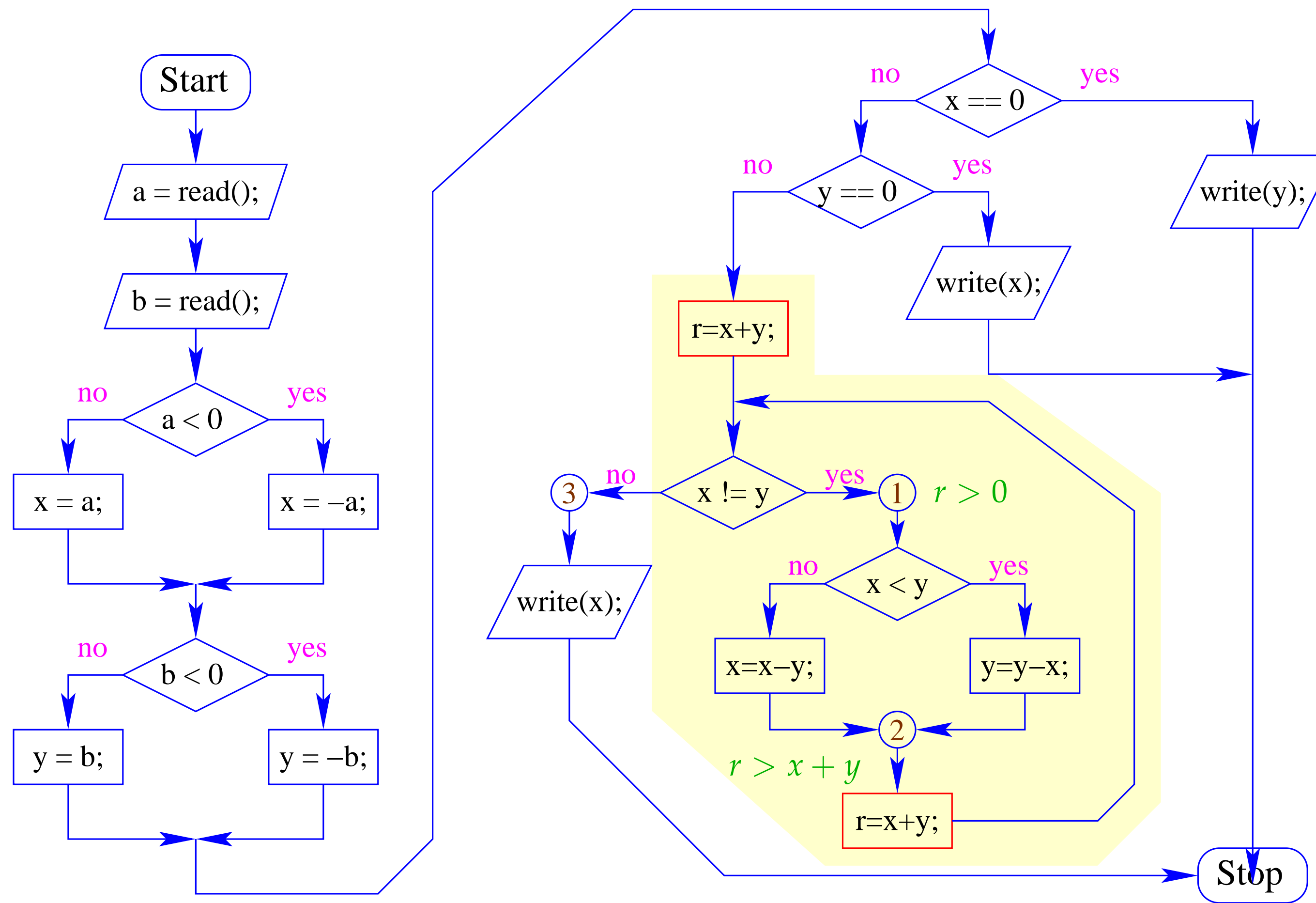
Beispiel: Sicheres ggT-Programm

```
int a, b, x, y;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
     else {
        while (x != y)
           if (y > x) y = y-x;
           else       x = x-y;
        write(x);
}
```

Wir wählen: $r = x + y$

Transformation:

```
int a, b, x, y, r;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
     else { r = x+y;
        while (x != y) {
           if (y > x) y = y-x;
           else       x = x-y;
           r = x+y; }
        write(x);
}
```



Start
a = read();
b = read();
a < 0
no
yes
x = a;
x = −a;
b < 0
no
yes
y = b;
y = −b;
x == 0
no
yes
y == 0
no
yes
write(y);
write(x);
r=x+y;
x != y
no
yes
3
1
r > 0
x < y
no
yes
x=x−y;
y=y−x;
2
r > x + y
r=x+y;
write(x);
Stop

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

$$
\begin{array}{lllr}
(1) & A & \equiv & x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y \\
(2) & B & \equiv & x > 0 \wedge y > 0 \wedge r > x + y \\
(3) & \textbf{true} & &
\end{array}
$$

Dann gilt:

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

Wir überprüfen:

$$
\begin{aligned}
\mathbf{WP}[\![\texttt{x != y}]\!](\mathbf{true}, A) &\equiv x = y \lor A \\
&\Leftarrow x > 0 \land y > 0 \land r = x + y \\
&\equiv C
\end{aligned}
$$

Wir überprüfen:

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

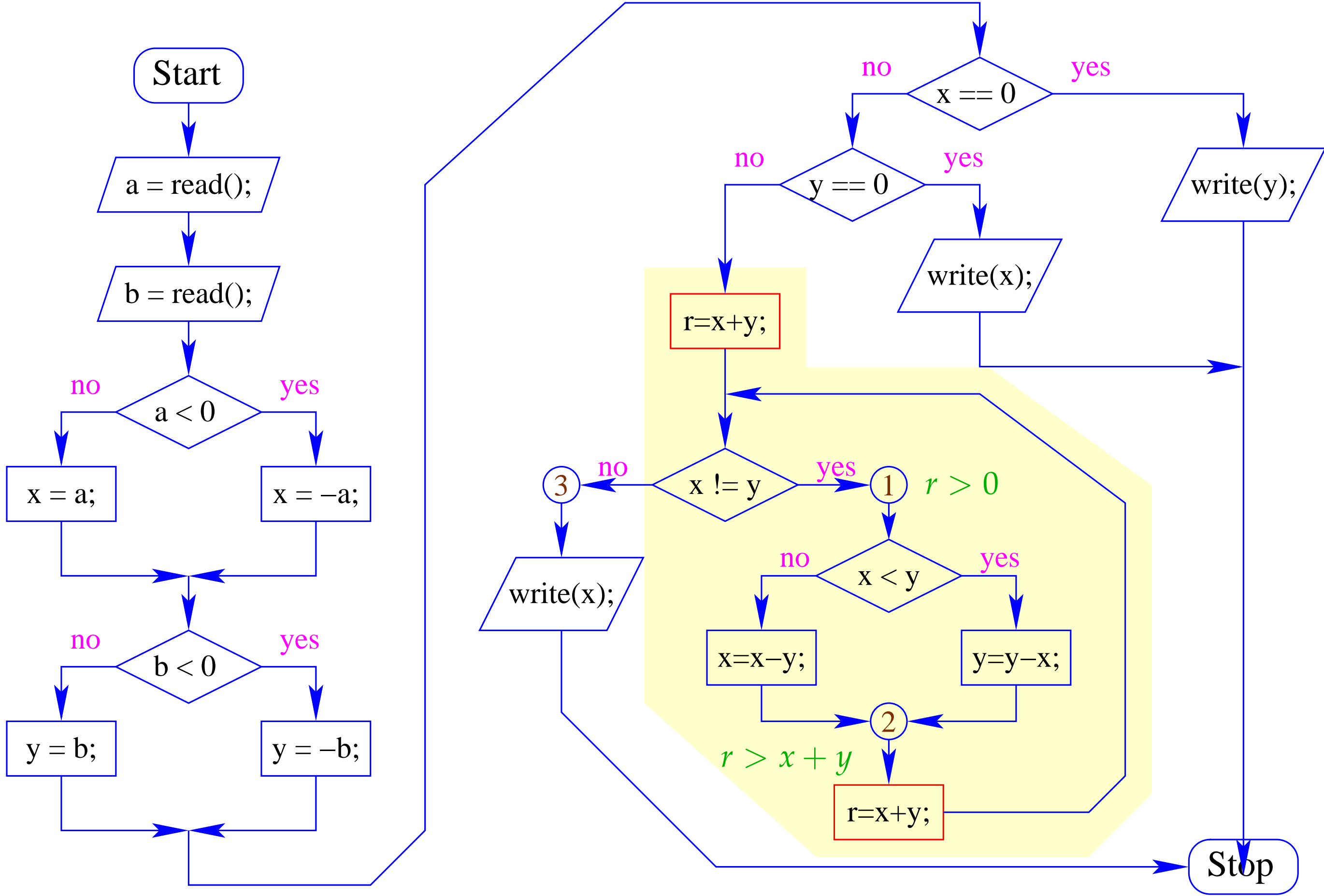
Wir überprüfen:

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

Wir überprüfen:

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

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

  $\Longrightarrow$ das Programm terminiert :-))

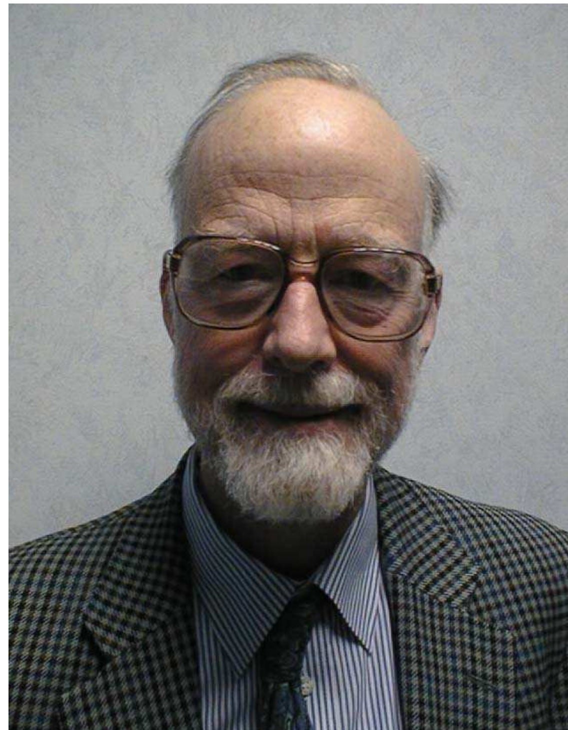
Allgemeines Vorgehen:

- 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` :-)

1.5 Modulare Verification und Prozeduren



Tony Hoare, Microsoft Research, Cambridge

Idee:

- Modularisiere den Korrektheitsbeweis so, dass Teilbeweise für wiederkehrende Aufgaben wiederverwendet werden können.
- Betrachte Aussagen der Form:

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

... 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:

- Modularisiere den Korrektheitsbeweis so, dass Teilbeweise für wiederkehrende Aufgaben wiederverwendet werden können.
- Betrachte Aussagen der Form:

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

... 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.

$$\begin{array}{lll} A & : & \text{Vorbedingung} \\ B & : & \text{Nachbedingung} \end{array}$$

Beispiele:

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

Beispiele:

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

$$\{\textbf{true}\} \quad \texttt{if (x<0) x=-x;} \quad \{x \geq 0\}$$

Beispiele:

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

$$\{\textbf{true}\} \quad \texttt{if (x<0) x=-x;} \quad \{x \geq 0\}$$

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

Beispiele:

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

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

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

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

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 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 Einfachkeit 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:

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

Vorgehen:

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

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

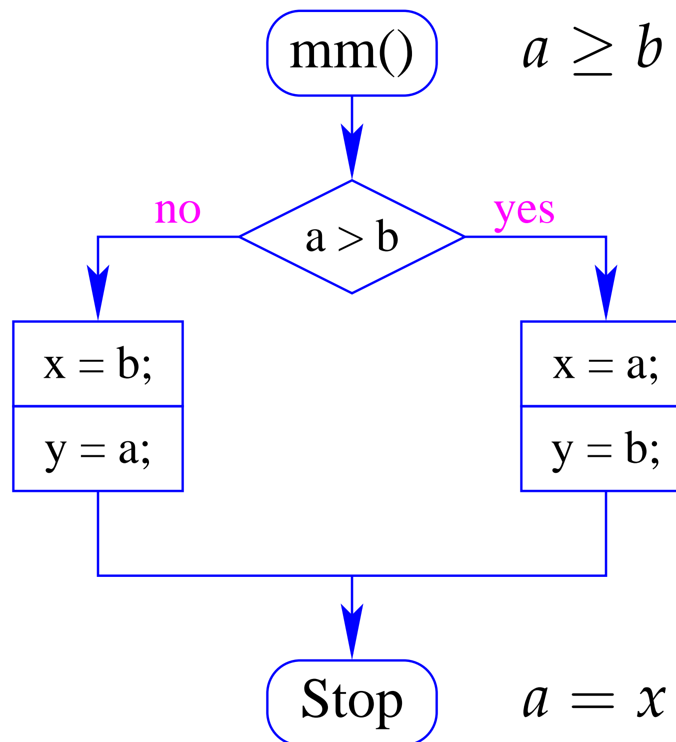
- Unter dieser globalen Hypothese H verifizieren wir, dass sich für jede Prozedurdefinition `void f() { ss }` zeigen lässt:

$$\{A\}\ \texttt{ss}\ \{B\}$$

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

... im Beispiel:

Wir überprüfen:



... im Beispiel:

Wir überprüfen: