Script generated by TTT

Title: Info2 (06.11.2015)

Date: Fri Nov 06 08:34:03 CET 2015

Duration: 89:30 min

Pages: 50

Beispiele

- Das ggT-Programm terminiert nur für Eingaben a, b mit: $a > 0$ und $b > 0$.
- Das Quadrier-Programm terminiert nur für Eingaben $a \geq 0$.
- `while (true) ;` terminiert nie.
- Programme ohne Schleifen terminieren immer!

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!

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!

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);
}
```

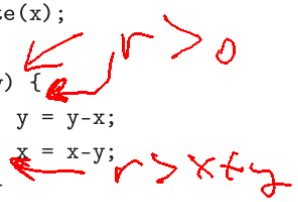

Beispiel: Sicheres ggT-Programm

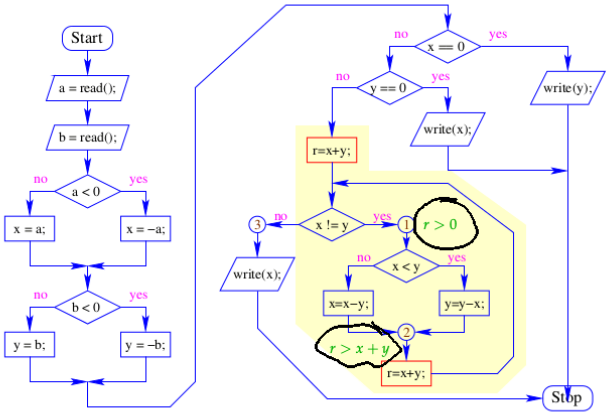
```
int a, b, x, y;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
     else {
        while (x != y)
           if (y > x) y = y-x;
           else       x = x-y;
        write(x);
}
```


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);
}
```





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

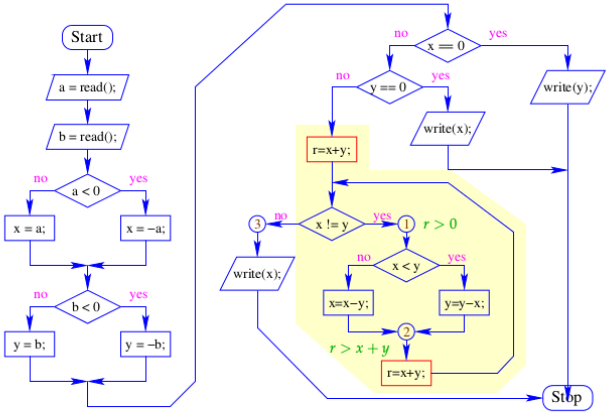
(1) $A \equiv x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$

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

(3) **true**

Dann gilt:

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



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

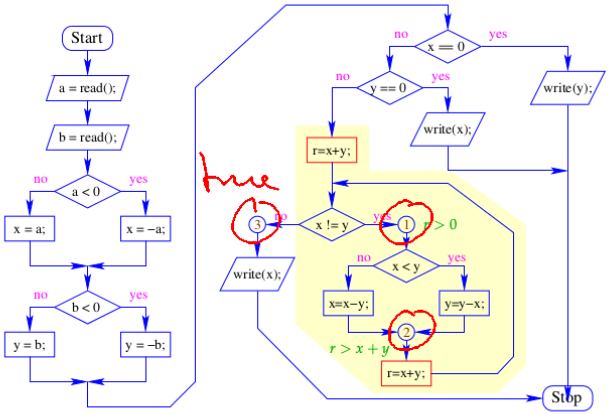
(1) $A \equiv x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$

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

(3) **true**

Dann gilt:

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



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

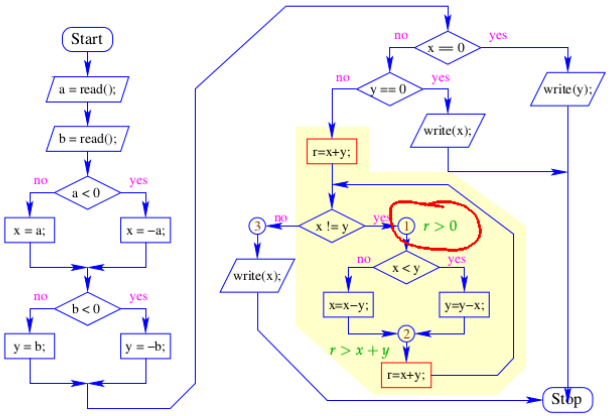
(1) $A \quad \equiv \quad x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$

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

(3) **true**

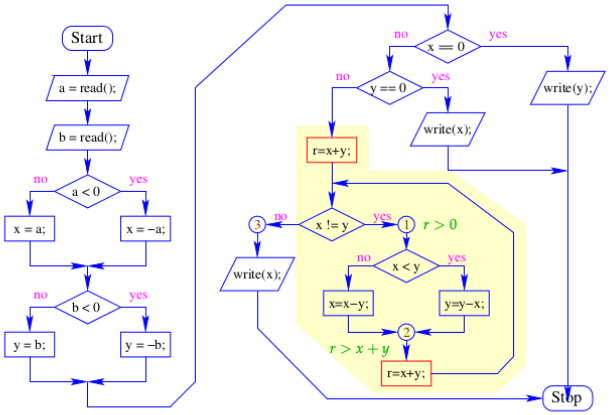
Dann gilt:

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



Wir überprüfen:

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



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

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

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

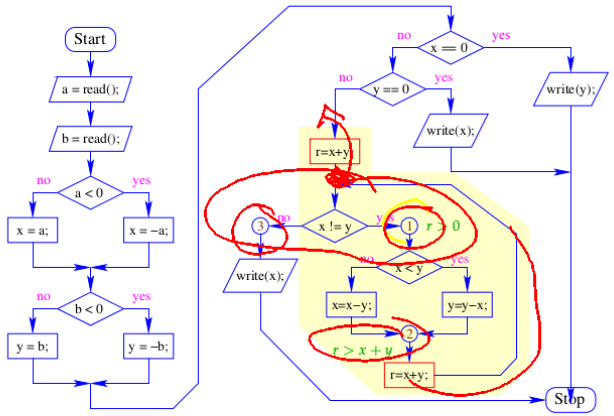
(3) **true**

Dann gilt:

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

Wir überprüfen:

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



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

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

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

(3) **true**

Dann gilt:

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

Wir überprüfen:

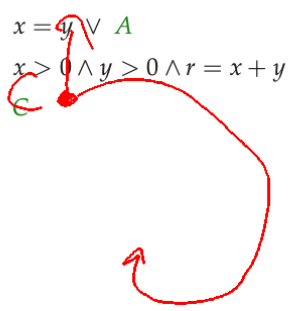
$$\begin{array}{lcl} \mathbf{WP}[\![\texttt{x != y}]\!](\mathbf{true}, A) & \equiv & x = y \vee A \\ & \Leftarrow & x > 0 \wedge y > 0 \wedge r = x + y \\ & \equiv & C \end{array}$$

Wir überprüfen:

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

Wir überprüfen:

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



Wir überprüfen:

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

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

(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) **true**

Dann gilt:

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

Wir überprüfen:

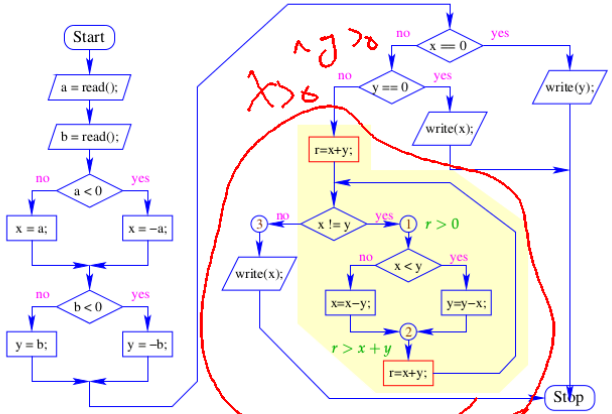
$$
\begin{array}{lcl}
\mathbf{WP}[\![\texttt{x != y}]\!](\mathbf{true}, A) & \equiv & x = y \vee A \\
& \Leftarrow & x > 0 \wedge y > 0 \wedge r = x + y \\
& \equiv & C \\
\mathbf{WP}[\![\texttt{r = x+y;}]\!](C) & \equiv & x > 0 \wedge y > 0 \\
& \Leftarrow & B
\end{array}
$$

Wir überprüfen:

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

Wir überprüfen:

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



Wir überprüfen:

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

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:

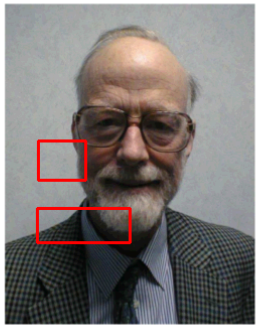
- An den Programmpunkten 1 und 2 gelten die Zusicherungen $r > 0$ bzw. $r > x + y$.
- In jeder Iteration wird r kleiner, bleibt aber stets positiv.
- Folglich wird die Schleife nur endlich oft durchlaufen

  $\Longrightarrow$ das Programm terminiert!

1.5 Modulare Verification und Prozeduren

Tony Hoare, Microsoft Research, Cambridge

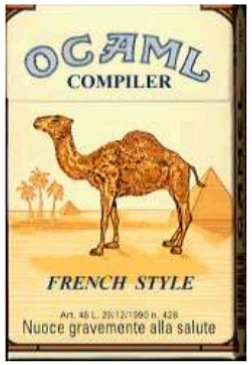
1.5 Modulare Verification und Prozeduren

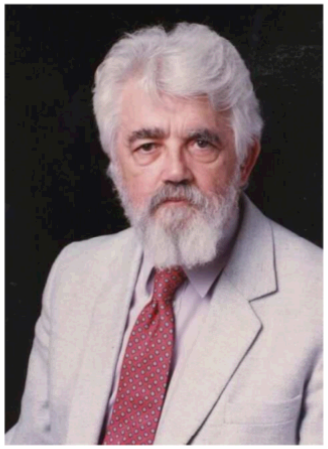


Tony Hoare, Microsoft Research, Cambridge

96

Funktionale Programmierung





John McCarthy, Stanford

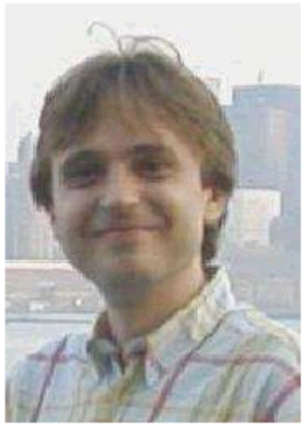
133

Robin Milner, Edinburgh

134

Robin Milner, Edinburgh

134



Xavier Leroy, INRIA, Paris

135

2 Grundlagen

- Interpreter-Umgebung
- Ausdrücke
- Wert-Definitionen
- Komplexere Datenstrukturen
- Listen
- Definitionen (Forts.)
- Benutzer-definierte Datentypen

2.1 Die Interpreter-Umgebung

Der Interpreter wird mit `ocaml` aufgerufen...

```
seidl@linux:~> ocaml
        Objective Caml version 4.02.3
#
```

Definitionen von Variablen, Funktionen, ... können direkt eingegeben werden.

Alternativ kann man sie aus einer Datei einlesen:

```
# #use "Hallo.ml";;
```


2.2 Ausdrücke

```
# 3+4;;
- : int = 7
# 3+
  4;;
- : int = 7
#
```

→ Bei `#` wartet der Interpreter auf Eingabe.

→ Das `;;` bewirkt Auswertung der bisherigen Eingabe.

→ Das Ergebnis wird berechnet und mit seinem Typ ausgegeben.

Vorteil: Das Testen von einzelnen Funktionen kann stattfinden, ohne jedesmal neu zu übersetzen!

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

```
# -3.0/.4.0;;
- : float = -0.75
# "So"^" "^"geht"^" "^"das";;
- : string = "So geht das"
# 1>2 || not (2.0<1.0);;
- : bool = true
```


2.3 Wert-Definitionen

Mit `let` kann man eine Variable mit einem Wert belegen.

Die Variable behält diesen Wert für immer!

```
# let seven = 3+4;;
val seven : int = 7
# seven;;
- : int = 7
```

Achtung: Variablen-Namen werden klein geschrieben !!!