Script generated by TTT

Title: Seidl: Info2 (11.11.2016)

Date: Fri Nov 11 08:29:10 CET 2016

Duration: 84:22 min

Pages: 48

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!

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!

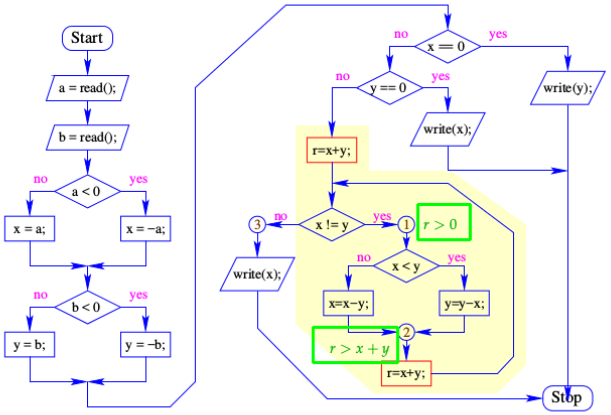
Beispiel: Sicheres ggT-Programm

```
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);
}
```

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

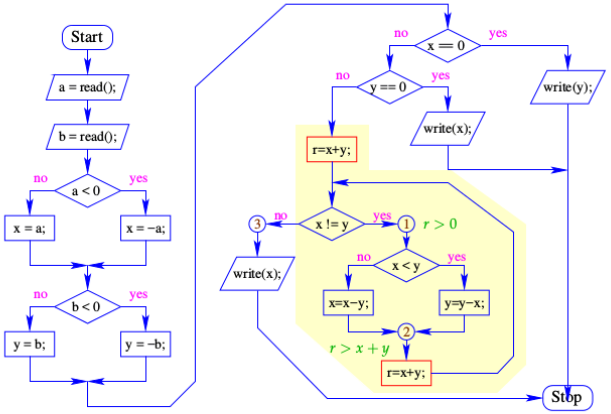
(1) $A \equiv x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$

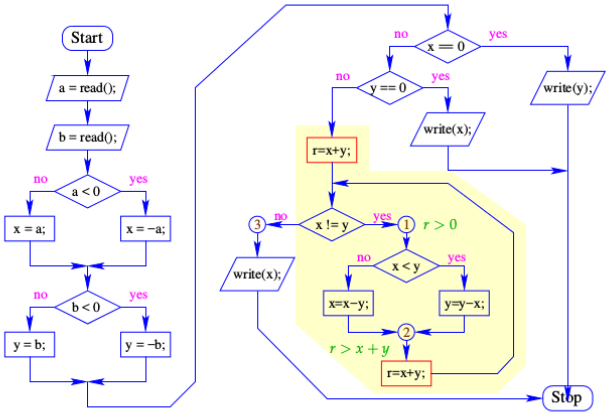
(2) $B \equiv x > 0 \wedge y > 0 \wedge r > x + y$

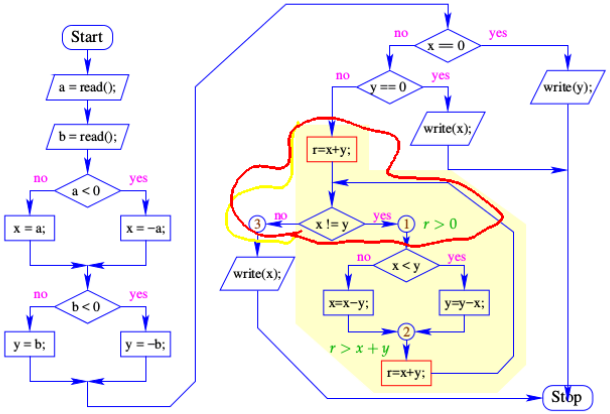
(3) **true**

Dann gilt:

$A \Rightarrow r > 0$ und $B \Rightarrow r > x + y$







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:

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

Wir überprüfen:

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

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

(1) $A \equiv x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$
(2) $B \equiv x > 0 \wedge y > 0 \wedge r > x + y$
(3) **true**

Dann gilt:

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

Wir überprüfen:

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

92

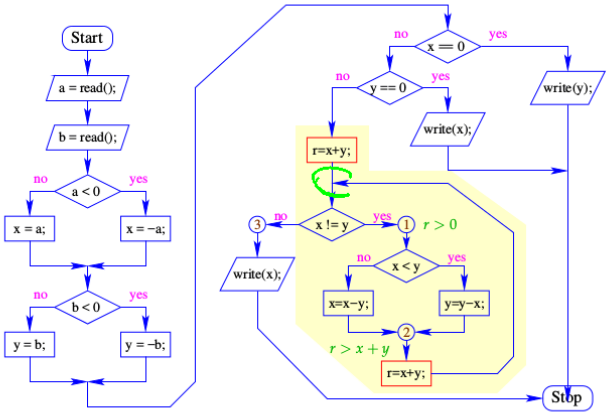
An den Programmpunkten 1, 2 und 3 machen wir die Zusicherungen:

(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:

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

91



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:

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

Wir überprüfen:

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

$x > 0 \wedge y > 0 \wedge$
$x + y = x + y$

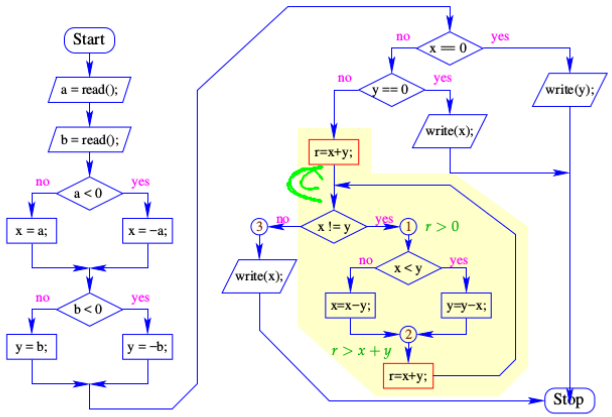
Wir überprüfen:

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

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.

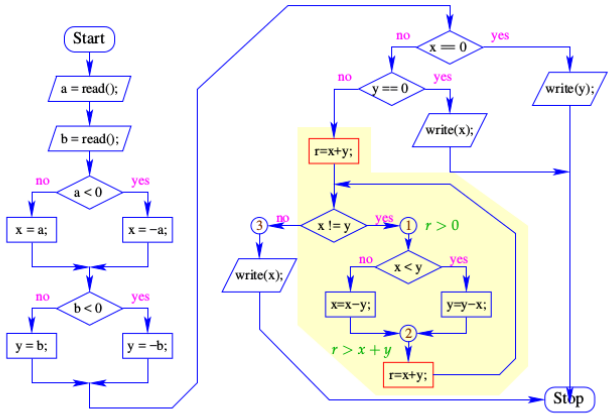
Wir schließen:

- 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!

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

Orientierung



Allgemeines Vorgehen

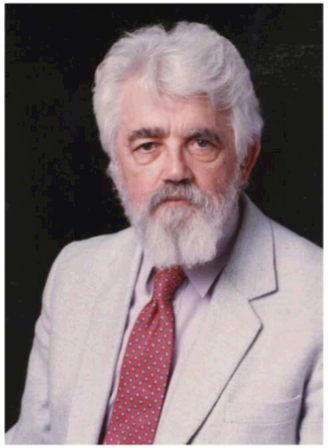
- Für jede vorkommende Schleife `while (b) s` erfinden wir eine neue Variable `r`.
- Dann transformieren wir die Schleife in:

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

für geeignete Ausdrücke `e0`, `e1`.

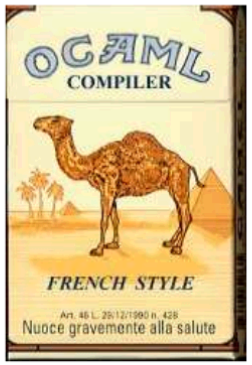
1.5 Modulare Verification und Prozeduren

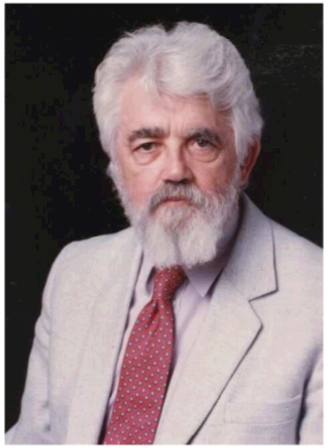
Tony Hoare, Microsoft Research, Cambridge



John McCarthy, Stanford

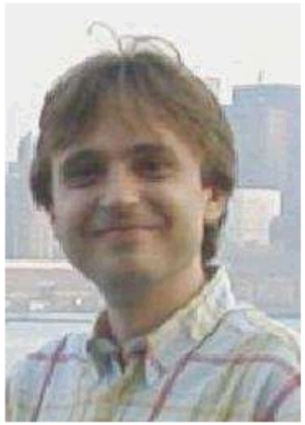
Funktionale Programmierung





John McCarthy, Stanford

Robin Milner, Edinburgh



Xavier Leroy, INRIA, Paris

2 Grundlagen

2.1 Die Interpreter-Umgebung

Der Interpreter wird mit `utop` aufgerufen...

```
seidl@linux:~> utop
        Objective Caml version 4.04.0
#
```

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

Alternativ kann man sie aus einer Datei einlesen:

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

2.1 Die Interpreter-Umgebung

Der Interpreter wird mit `utop` aufgerufen...

```
seidl@linux:~> utop
        Objective Caml version 4.04.0
#
```

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!

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!

Vordefinierte Konstanten und Operatoren

Typ	Konstanten: Beispiele	Operatoren
int	`0 3 -7`	`+ - * / mod`
float	`-3.0 7.0`	`+. -. *. /.`
bool	`true false`	`not || &&`
string	`"hallo"`	`^`
char	`'a' 'b'`	

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
```


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
```


Eine erneute Definition für seven weist nicht seven einen neuen Wert zu, sondern erzeugt eine neue Variable mit Namen seven.

```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```

Die alte Definition wurde unsichtbar (ist aber trotzdem noch vorhanden!

Offenbar kann die neue Variable auch einen anderen Typ haben.

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 !!!

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 !!!

Eine erneute Definition für seven weist nicht seven einen neuen Wert zu, sondern erzeugt eine neue Variable mit Namen seven.

```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```

Die alte Definition wurde unsichtbar (ist aber trotzdem noch vorhanden!

Offenbar kann die neue Variable auch einen anderen Typ haben.

Eine erneute Definition für seven weist nicht seven einen neuen Wert zu, sondern erzeugt eine neue Variable mit Namen seven.

```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```

Die alte Definition wurde unsichtbar (ist aber trotzdem noch vorhanden!

Offenbar kann die neue Variable auch einen anderen Typ haben.

2.4 Komplexere Datenstrukturen

- Paare:

```
# (3,4);;
- : int * int = (3, 4)
# (1=2,"hallo");;
- : bool * string = (false, "hallo")
```

- Tupel:

```
# (2,3,4,5);;
- : int * int * int * int = (2, 3, 4, 5)
# ("hallo",true,3.14159);;
-: string * bool * float = ("hallo", true, 3.14159)
```