

Teil 9: Modellierung von Semantik

- Modellierung von Programmiersprachen-Semantik
- Projekt: Datenflussanalyse

Modellierung von Semantik (1)

Operationelle Semantik:

Beschreibung der Programmausführung durch abstrakte Maschine

Ingredientien:

- Zustandsbegriff
- Programmiersprache (Ausdrücke, Anweisungen, ...)
- Ausführungsrelation

Modellierung von Semantik (2)

Im folgenden:

- Einfache imperative Sprache (“while-Sprache”)
- Keine komplexe Speicherstruktur (Pointer etc.)
- Kein komplexer Kontrollfluß (Prozeduren, Exceptions etc.)

Es geht auch besser, z.B. Modellierung der Semantik von Java

↪ **Semantik-Vorlesung nächstes Semester**

Zustände

Speicherzellen: uninterpretiert

typedcl *loc*

Werte: hier nur nat. Zahlen

Zustand:

types

state = "loc \Rightarrow nat"

Programmiersprache: Ausdrücke

datatype *binop* = *BPlus* / *BMinus* / *BEq*

datatype *expr*

= *N nat* — constant

/ *Var loc* — variable

/ *Op binop expr expr*

Programmiersprache: Anweisungen

datatype *com*

= *SKIP*

| *Assign loc expr* ("_ ::= _ ")

| *Semi com com* ("_; _")

| *Cond expr com com* ("IF _ THEN _ ELSE _")

| *While expr com* ("WHILE _ DO _")

Beispiel:

Abstrakte Syntax:

```
"WHILE (Var y) DO
  (x ::= (Op BPlus (Var y) (N 2)));
  y ::= (Op BPlus (Var y) (N 1)))"
```

... steht für:

```
WHILE y DO {
  x = y + 2;
  y = y + 1;
}
```

Evaluierungs- und Ausführungsrelationen

Evaluierung von Ausdrücken:

consts

`evale :: "[expr, state] ⇒ nat"`

... zu definieren

Ausführungsrelation für Anweisungen: (warum keine Funktion?)

consts `evalc :: "(com × state × state) set"`

Geschrieben als:

translations `"⟨c, s⟩ →c s'" == "(c, s, s') ∈ evalc"`

Ausführungsrelation: einige Regeln

“Big-Step”-Semantik

inductive evalc intros

Assign: " $\langle x := a, s \rangle \longrightarrow_c s(x := (\text{evale } a \ s))$ "

Semi: " $\llbracket \langle c_0, s \rangle \longrightarrow_c s''; \langle c_1, s'' \rangle \longrightarrow_c s' \rrbracket$
 $\implies \langle c_0; c_1, s \rangle \longrightarrow_c s'$ "

WhileFalse: " $\text{evale } b \ s = 0$
 $\implies \langle \text{WHILE } b \ \text{DO } c, s \rangle \longrightarrow_c s$ "

WhileTrue: " $\llbracket \text{evale } b \ s \neq 0; \langle c, s \rangle \longrightarrow_c s''; \langle \text{WHILE } b \ \text{DO } c, s'' \rangle \longrightarrow_c s' \rrbracket$
 $\implies \langle \text{WHILE } b \ \text{DO } c, s \rangle \longrightarrow_c s'$ "

Ausführungsrelation: Beispiel

Anfangszustand: Alle Variablen auf 0

lemma " $m \neq n \implies p \neq m \implies p \neq n \implies$
 $\langle (m ::= (N\ 3));$
 $(n ::= (N\ 2));$
 $(WHILE\ (Var\ n)\ DO$
 $(p ::= (Op\ BPlus\ (Var\ p)\ (Var\ m)));$
 $n ::= (Op\ BMinus\ (Var\ n)\ (N\ 1))))))$,
 $\lambda\ v.\ 0 \rangle \longrightarrow_c\ ?s'$ "

- Modellierung von Programmiersprachen-Semantik
- **Projekt: Datenflussanalyse**

Lebendige Variablen: Begriff

Variable v ist **lebendig** an einem Programmpunkt P , wenn

- in jedem weiteren möglichen Programmverlauf auf die Variable lesend zugegriffen wird
- ohne daß v vorher überschrieben worden ist.

Bsp.:

$x = 3;$ (1)

$y = 5;$ (2)

$y = x + 3;$ (3)

$x = x + y;$ (4)

x lebendig (mindestens) nach (1), (2), (3)

y lebendig (mindestens) nach (3)

y nicht lebendig nach (2)

Lebendigkeitsanalyse: Verwendung

Compileroptimierung:

Zuweisungen an nicht-lebendige Variablen können gestrichen werden (“dead code elimination”)

$x = 3;$ (1)

$y = 5;$ (2)

$y = x + 3;$ (3)

$x = x + y;$ (4)

$x = 3;$ (1)

$y = x + 3;$ (3)

$x = x + y;$ (4)

Korrektheitsnachweis?

Lebendige Variablen: Verfeinerung

Frage: Ist x lebendig nach (4)?

· · ·
 $x = x + y;$ (4)
· · ·

Antwort:

- *Ja*, wenn x noch verwendet wird,
z.B. in `return(x + y)`
- *Nein*, wenn x nicht mehr verwendet wird,
z.B. in `return(y)`

↪ Betrachte Lebendigkeit bzgl. Menge **relevanter Variablen**

Äquivalenz bzgl. relevanter Variablen (1)

Äquivalenz zweier Funktionen auf Menge A :

```
equiv_on :: "[ 'a set, 'a ⇒ 'b, 'a ⇒ 'b ] ⇒ bool"
```

```
"equiv_on A f1 f2 == ∀ x ∈ A. f1 x = f2 x"
```

[Funktionen hier: Programmzustände]

Äquivalenz bzgl. relevanter Variablen (2)

<i>Original</i>		<i>Optimierung 1</i>		<i>Optimierung 2</i>
$x = 3;$	(1)	$x = 3;$	(1)	$x = 3;$
$y = 5;$	(2)			
$y = x + 3;$	(3)	$y = x + 3;$	(3)	$y = x + 3;$
$x = x + y;$	(4)	$x = x + y;$	(4)	
<i>s_orig</i>		<i>s_opt1</i>		<i>s_opt2</i>

Es gilt:

equiv_on {x,y} s_orig s_opt1

equiv_on {y} s_orig s_opt2

Es gilt nicht:

equiv_on {x} s_orig s_opt2

Lebendige Variablen: Komplikationen (1)

Ist y lebendig nach (1),
da y in (2) nicht gebraucht wird?
Ist folgende Optimierung korrekt?

Original

```
y = 5;          (1)
IF (x > 0)
THEN x = x + 1  (2)
ELSE x = x + y  (3)
return(x)
```

Optimierung

```
IF (x > 0)
THEN x = x + 1  (2)
ELSE x = x + y  (3)
return(x)
```

Lebendige Variablen: Komplikationen (2)

Ist x lebendig nach (3),
da x nach Ende der Schleife nicht mehr gebraucht wird?
Ist folgende Optimierung korrekt?

<i>Original</i>		<i>Original</i>	
$x = 2;$	(1)	$x = 2;$	(1)
$y = 5;$	(2)	$y = 5;$	(2)
WHILE $x > 0$ DO		WHILE $x > 0$ DO	
$x = x - 1;$	(3)	SKIP;	(3)
return(y)		return(y)	

Datenflußanalyse (1)

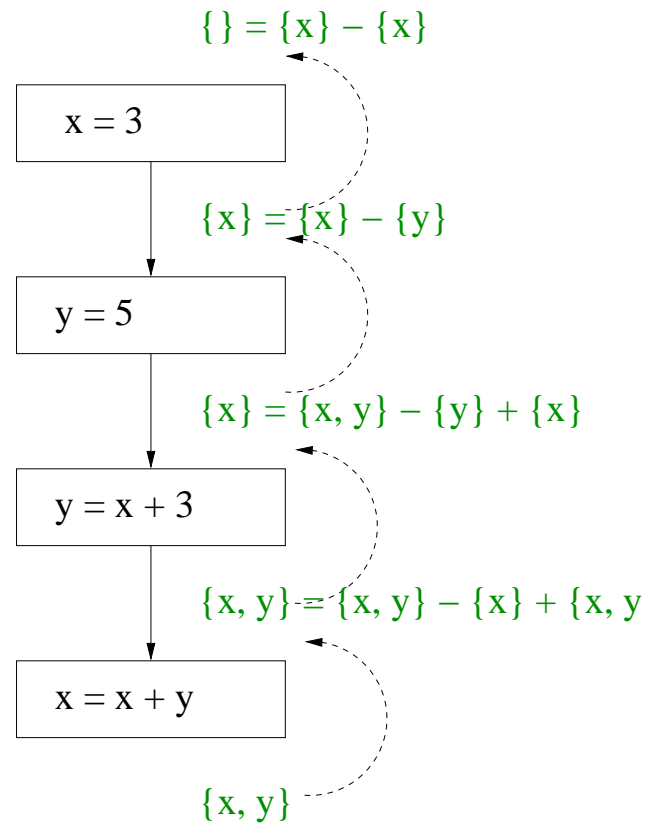
Beachte alle Pfade im Kontrollflußgraphen!

Vorgehen: Traversiere Kontrollflußgraphen von hinten nach vorne

- Bekannt: Lebendige Variablen A *nach* Anweisung
- Berechne: Lebendige Variablen B *vor* Anweisung

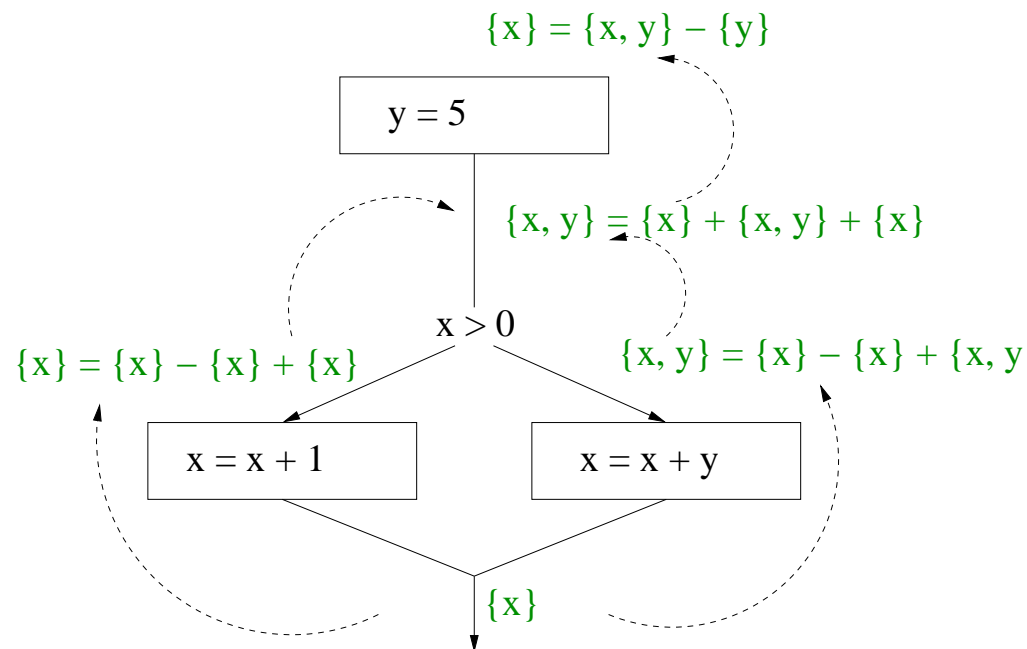
Datenflußanalyse (2)

Kontrollfluß ohne Verzweigung



Datenflußanalyse (3)

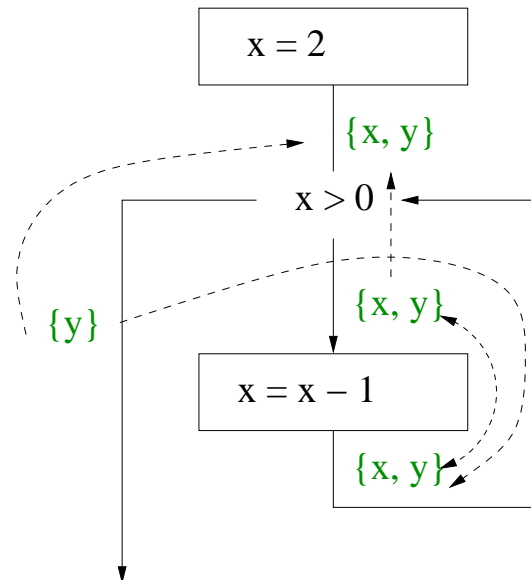
Kontrollfluß mit Verzweigung (Conditional)



Prinzip: Schätze Menge lebendiger Variablen *konservativ* ab
 “Lieber zu viele Variablen als lebendig bezeichnen als zu wenige”

Datenflußanalyse (4)

Schleife



Frage: Warum muss x in der Schleife lebendig sein?

Analyse von Schleifen (1)

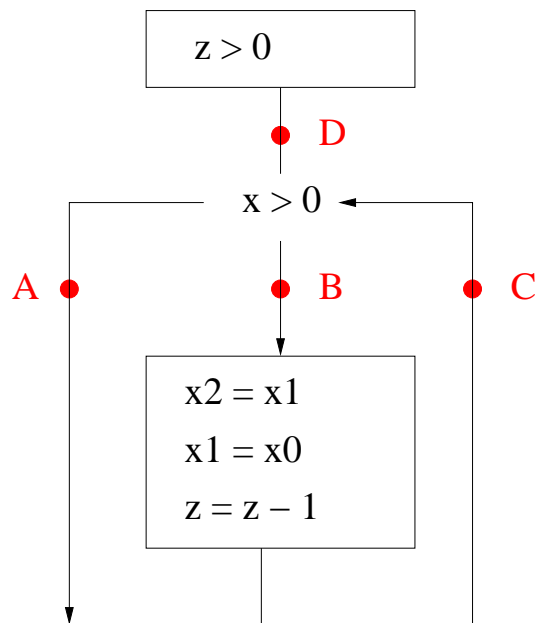
Warum genügt einmalige Traversierung nicht?

Bsp: Kann Zuweisung (2) gelöscht werden?

```
WHILE z > 0 DO  
  x2 = x1;      (1)  
  x1 = x0;      (2)  
  z = z - 1;    (3)  
return(x2)
```

Analyse von Schleifen (2)

Datenfluß-Gleichungen:



Bestimme kleinste Lösung von:

$$\{x_2\} \subseteq A$$

$$B \subseteq C - \{x_2\} \cup \{x_0, x_1\}$$

$$C \subseteq A \cup B \cup \{z\}$$

$$D \subseteq A \cup B \cup \{z\}$$

Damit insbesondere:

$$\{x_0, x_1\} \subseteq C$$

Analyse von Schleifen (3)

Vereinfachung:

Gleichungen:

$$B = A \cup FV_c \cup FV_e$$

$$D = A \cup B \cup FV_e$$

wobei

FV_c die freien Variablen des Schleifenkörpers

FV_e die freien Variablen der Schleifenbedingung

Damit: Analyse etwas ungenauer, aber immer noch korrekt

Zu diesem Thema interessantes SEP zu vergeben!

Datenflußanalyse und Optimierung

Aufgabe: Definition einer Funktion

$optimLV :: "[com, loc set] \Rightarrow com \times (loc set)"$

Sei

- c ein zu optimierendes Programm
- A die Menge der lebendigen Variablen *nach* Ausführung von c

Dann gilt für $optimLV\ c\ A = (c', B)$

- c' ist das optimierte Programm
- B die Menge der lebendigen Variablen *vor* Ausführung von c

Korrektheitsaussage

Das nicht-optimierte Programm c
 und das optimierte Programm c'
 sind äquivalent bezüglich der Menge lebendiger Variablen A .

lemma *optim_correct*: "
 $\llbracket \langle c, s \rangle \longrightarrow_c s'; \text{optimLV } c \ A = (c', B); \langle c', s \rangle \longrightarrow_c so' \rrbracket \implies$
 $\text{equiv_on } A \ s' \ so'$ "

Beweis

- Induktion über Ausführungsrelation des nicht-optimierten Programms
- “Nachziehen” des optimierten Programms

Teilfall Komposition mit c als $c1; c2$

