

## Kapitel III

# Programmanalyse und Typsysteme

# Motivation

Verifikationsmethoden wie Model Checking untersuchen den vollständigen Zustandsraum eines Systems.

- Das beschränkt die Verifikation auf (Teil-)Systeme mit nicht zu großem Zustandsraum.
- Der Zustandsraum kann durch Abstraktion verkleinert werden; dann muss aber auch die Abstraktion verifiziert werden.

In diesem Kapitel befassen wir uns mit Methoden der Programmiersprachentheorie zur Analyse von Software realistischer Größe.

# Motivation

## Statische Analyse von Programmen realistischer Größe

Schwächere Eigenschaften können automatisch überprüft werden.

- Abwesenheit von Laufzeitfehlern wie Division durch Null, falscher Speicherzugriff, ungefangene Exceptions, Verletzung von Ressourcenschranken und -protokollen (z.B. Dateizugriff).
- Datenflusseigenschaften zur Programmoptimierung

Solche Eigenschaften sind z.B. im Compilerbau nützlich.

Kompliziertere Eigenschaften erfordern manuelle Hilfe durch

- Angabe von Typannotaten,
- Spezifikation von Zusicherungen und Invarianten,
- formale Beweise.

# Inhalt Kapitel III

- Induktive Definitionen
- Programmanalyse für imperative Programme
  - Spezifikation einer While-Sprache
  - Datenflussanalyse
- Fixpunkttheorie
- Programmanalyse und Typsysteme
- Typ- und Effektsysteme
  - Funktionale Sprache
  - Typinferenz
  - Polymorphie
  - Kontrollflussanalyse

# Induktive Definitionen

In der Programmiersprachentheorie werden viele Konzepte durch induktive Definitionen formalisiert.

Das übliche Format für induktive Definitionen sind *Inferenzregeln*.

Beispiele aus der Vorlesung:

$$\frac{e_1 \longrightarrow \text{fun}_\pi x \rightarrow e'_1 \quad e_2 \longrightarrow v_2 \quad e'_1[x \mapsto v_2] \longrightarrow v}{e_1 e_2 \longrightarrow v}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2 \ \& \ \varphi_2 \quad \Gamma \vdash e_2 : \tau_1 \ \& \ \varphi_3}{\Gamma \vdash e_1 e_2 : \tau_2 \ \& \ \varphi_1 \cup \varphi_2 \cup \varphi_3}$$

Ein **Urteil** ist ein formaler Ausdruck, der eine bestimmte Aussage ausdrückt (meist eine Eigenschaft bestimmter Objekte).

- $n \text{ even}$  “Die natürliche Zahl  $n$  ist gerade.”
- $n \text{ odd}$  “Die natürliche Zahl  $n$  ist ungerade.”
- $\text{sum}(m, n) = r$  “Die Summe der nat. Zahlen  $m$  und  $n$  ist die nat. Zahl  $r$ .”
- $\phi \vdash \psi$  “Jede Belegung, welche die aussagenlogischen Formeln  $\phi$  wahr macht, macht auch  $\psi$  wahr.”
- $e: X$  “Der Programmausdruck  $e$  hat Typ  $X$ .”

Urteile sind als *rein syntaktische* Ausdrücke ohne inhärente Bedeutung zu verstehen (oben sind “*sum*” und “ $\vdash$ ” nur Symbole).

# Inferenzregeln

Die Bedeutung von Urteilen wird durch *Inferenzregeln* festgelegt.

Eine Inferenzregel hat die Form:

$$(\text{NAME}) \frac{U_1 \quad U_2 \quad \dots \quad U_n}{U}$$

- Die Urteile  $U_1, \dots, U_n$  heißen *Prämissen*.
- Das Urteil  $U$  heißt *Konklusion*.
- Inferenzregeln ohne Prämissen (d.h. mit  $n = 0$ ) heißen *Axiome*.

Bedeutung der Regel: Wenn die Prämissen alle zutreffen, dann auch die Konklusion.

# Inferenzregeln: Beispiele

Gerade/ungerade:

$$\begin{array}{lcl}
 (\text{EVENZ}) \frac{}{0 \text{ even}} & (\text{ODDS}) \frac{n \text{ even}}{n + 1 \text{ odd}} & (\text{EVENS}) \frac{n \text{ odd}}{n + 1 \text{ even}}
 \end{array}$$

Summen:

$$\begin{array}{lcl}
 (\text{SUMZ}) \frac{}{\text{sum}(n, 0) = n} & (\text{SUMS}) \frac{\text{sum}(n, m) = r}{\text{sum}(n, m + 1) = r + 1}
 \end{array}$$

Aussagenlogik:

$$\begin{array}{cccc}
 \frac{}{\phi \vdash \phi} & \frac{\phi \vdash \psi_1 \quad \phi \vdash \psi_2}{\phi \vdash \psi_1 \wedge \psi_2} & \frac{\phi \vdash \psi_1}{\phi \vdash \psi_1 \vee \psi_2} & \frac{\phi \vdash \psi_2}{\phi \vdash \psi_1 \vee \psi_2}
 \end{array}$$

...



# Inferenzregeln (genau)

Inferenzregeln enthalten meist Metavariablen und sind eigentlich als Regelschemata zu verstehen.

Gemeint sind alle Regeln, die man durch Einsetzung beliebiger passender(!) Ausdrücke für die Metavariablen erhält.

Beispiel:

$$\frac{sum(n, m) = r}{sum(n, m + 1) = r + 1}$$

steht für

$$\frac{sum(0, 0) = 0}{sum(0, 1) = 1}, \frac{sum(0, 0) = 1}{sum(0, 1) = 2}, \dots, \frac{sum(5, 6) = 18}{sum(5, 7) = 19}, \dots$$

Für  $m, n$  und  $r$  dürfen nur nat. Zahlen eingesetzt werden, da die Urteile in der Regel nur für nat. Zahlen definiert sind.

# Inferenzregeln (genau)

Beispiel:

$$\frac{\phi \vdash \psi_1 \quad \phi \vdash \psi_2}{\phi \vdash \psi_1 \wedge \psi_2}$$

steht für alle Regeln, die man durch Einsetzung von konkreten aussagenlogischen Formeln für  $\phi$ ,  $\psi_1$  und  $\psi_2$  erhält, z.B.

$$\frac{\top \vdash \top \quad \top \vdash \top}{\top \vdash \top \wedge \top}, \frac{(A \Rightarrow B) \vdash (C \wedge B) \quad (A \Rightarrow B) \vdash \perp}{(A \Rightarrow B) \vdash (C \wedge B) \wedge \perp}, \dots$$

Hier dürfen nur aussagenlogische Formeln eingesetzt werden, da die Urteile nur dafür definiert sind.

## Inferenzregeln: Seitenbedingungen

Die möglichen Werte der Metavariablen in schematischen Regeln werden manchmal durch Seitenbedingungen eingeschränkt.

**Beispiel:** Die Regel (SUMS) könnte auch so geschrieben werden.

$$\text{(SUMS)} \frac{\text{sum}(n, m) = r}{\text{sum}(n, m') = r'} m' = m + 1, r' = r + 1$$

Die Metavariablen dürfen nur so instantiiert werden, dass alle Seitenbedingungen erfüllt sind.

(Seitenbedingungen werden manchmal auch wie Prämissen über den Strich geschrieben.)

# Herleitungen

Der Begriff der *Herleitung für ein Urteil  $U$*  ist induktiv wie folgt definiert.

- Jede Regel der Form (d.h. ohne Prämisse)

$$(R) \frac{}{U}$$

ist eine Herleitung für ihre Konklusion  $U$ .

- Gibt es eine Regel

$$(S) \frac{U_1 \quad U_2 \quad \dots \quad U_n}{U}$$

und sind  $\Pi_1, \dots, \Pi_n$  jeweils Herleitungen für  $U_1, \dots, U_n$ , dann ist

$$(S) \frac{\Pi_1 \quad \Pi_2 \quad \dots \quad \Pi_n}{U}$$

eine Herleitung für  $U$ .

- Nichts weiter ist eine Herleitung.

# Herleitungen: Beispiele

Beispiele:

$$\begin{array}{c}
 \frac{}{A \vdash A} \quad \frac{}{A \vdash A} \quad \frac{}{A \vdash A} \\
 \frac{}{A \vdash A} \quad \frac{}{A \vdash A} \quad \frac{}{A \vdash B \vee A} \\
 \frac{}{A \vdash A} \quad \frac{}{A \vdash A \wedge (B \vee A)} \\
 \frac{}{A \vdash A \wedge (A \wedge (B \vee A))}
 \end{array}
 \quad
 \begin{array}{c}
 (\text{SUMZ}) \frac{}{sum(3, 0) = 3} \\
 (\text{SUMS}) \frac{}{sum(3, 1) = 4} \\
 (\text{SUMS}) \frac{}{sum(3, 2) = 5}
 \end{array}$$

Kein Beispiel:

$$\begin{array}{c}
 (\text{SUMS}) \frac{sum(3, 0) = 4}{sum(3, 1) = 5} \\
 (\text{SUMS}) \frac{sum(3, 1) = 5}{sum(3, 2) = 6}
 \end{array}$$

(Regelnamen und Seitenbedingungen werden oft nicht ausgeschrieben.)

# Induktive Definition

Inferenzregeln sind induktive Definitionen von Urteilen.

**Induktionsprinzip:** Um zu zeigen, dass alle herleitbaren Urteile eine bestimmte Eigenschaft haben, genügt es für alle Regeln zu zeigen: Wenn die Prämissen der Regel die Eigenschaft haben, dann auch die Konklusion.

**Beispiel:** Wenn  $\text{sum}(m, n) = r$  herleitbar ist, dann gilt  $r = m + n$ .

- Regel (SUMZ)

$$\frac{}{\text{sum}(n, 0) = n}$$

Es gilt  $n = n + 0$ , also hat die Konklusion die Eigenschaft.

- Regel (SUMS)

$$\frac{\text{sum}(n, m) = r}{\text{sum}(n, m + 1) = r + 1}$$

Angenommen die Prämisse hat die Eigenschaft,

d.h.  $r = m + n$ . Dann gilt  $r + 1 = m + n + 1 = (m + 1) + n$ , also hat auch die Konklusion die Eigenschaft.

# Induktive Definitionen

Inferenzregeln sind als reine Spezifikation von Urteilen zu verstehen.

- Inferenzregeln spezifizieren was herleitbar ist, aber nicht unbedingt wie.
- Die Frage, ob ein Urteil herleitbar ist, kann je nach Art der Regeln sehr leicht oder sehr schwer sein.
- Man verwendet Inferenzregeln, um möglichst einfach zu spezifizieren, was eine Analyse leisten soll, ohne sich dabei bereits auf einen Analysealgorithmus festzulegen.
- Beispiel: Das Urteil  $\text{sum}(n, m) = r$  sagt, was eine Summe ist; verschiedene Implementierungen der Addition realisieren diese Spezifikation, z.B.
  - Ripple-Carry Adder
  - Lookahead-Carry-Adder

# Einfache While-Sprache

## Arithmetische Ausdrücke:

$$a, a_1, a_2 ::= x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$$

## Boolesche Ausdrücke:

$$b, b_1, b_2 ::= \text{true} \mid \text{false} \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \mid a_1 < a_2 \mid a_1 = a_2$$

## Programmstücke (statements):

$$\begin{aligned} S, S_1, S_2 ::= & [x := a]^\ell \mid [\text{skip}]^\ell \mid S_1; S_2 \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \\ & \mid \text{while } [b]^\ell \text{ do } S \end{aligned}$$

Hierbei  $x$  läuft über Programmvariablen,  $n$  über Integerkonstanten,  $\ell$  über Labels (konkret: natürliche Zahlen), welche elementare Programmteile eindeutig markieren sollen.

Konvention:  $S_1; S_2; S_3$  steht für  $S_1; (S_2; S_3)$ .



# Beispielprogramm

$$[y:=x]^1; [z:=1]^2; \text{while } [1 < y]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5); [y:=0]^6$$

Alternative Notation:

```

1: y:=x;
2: z:=1;
3: while 1<y do (
4:     z:=z*y;
5:     y:=y-1);
6: y:=0
    
```

# Operationelle Semantik

Die operationelle Semantik spezifiziert wie sich While-Programme bei der Ausführung verhalten.

Ein *Programmzustand* ( $\sigma$ ) ist eine endliche Abbildung von Programmvariablen auf ganze Zahlen.

**Beispiel:**  $\sigma = [x \mapsto 3, y \mapsto 4]$ .

Wenn  $\sigma$  alle Variablen in einem arithmetischen Ausdruck  $a$  auf Werte abbildet, dann schreiben wir  $\llbracket a \rrbracket \sigma$  für den Wert des Ausdrucks mit den entsprechenden Belegungen. Sonst ist  $\llbracket a \rrbracket \sigma$  undefiniert.

**Beispiel:**  $\llbracket x + 4 * y \rrbracket \sigma = 19$

Analog für Boolesche Ausdrücke.

**Beispiel:**  $\llbracket x + 4 * y < 18 \rrbracket \sigma = \text{false}$

# Operationelle Semantik

Die operationelle Semantik ist durch zwei Urteile gegeben.

- $\langle S, \sigma \rangle \rightarrow \sigma'$  “Die Ausführung von  $S$  im Startzustand  $\sigma$  ist nach einem Schritt beendet und endet im Zustand  $\sigma'$ .”
- $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$  “Die Ausführung von  $S$  im Anfangszustand  $\sigma$  führt nach einem Schritt zum Zwischenzustand  $\sigma'$ ; es bleibt danach noch die Anweisung  $S'$  abzuarbeiten.”

# Regeln für die operationelle Semantik

$$(ASS) \frac{}{\langle [x := a]^\ell, \sigma \rangle \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma]}$$

$$(SKIP) \frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma}$$

$$(SEQ1) \frac{\langle S_1, \sigma \rangle \rightarrow \langle S'_1, \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S'_1; S_2, \sigma' \rangle}$$

$$(SEQ2) \frac{\langle S_1, \sigma \rangle \rightarrow \sigma'}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \sigma' \rangle}$$

Beachte: (ASS) ist nur anwendbar, wenn  $\llbracket a \rrbracket \sigma$  definiert ist.

# Regeln für die operationelle Semantik, Forts.

$$(IFT) \frac{}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle} \llbracket b \rrbracket \sigma = \text{true}$$

$$(IFF) \frac{}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle} \llbracket b \rrbracket \sigma = \text{false}$$

$$(WHILET) \frac{}{\langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \langle S; \text{while } b \text{ do } S, \sigma \rangle} \llbracket b \rrbracket \sigma = \text{true}$$

$$(WHILEF) \frac{}{\langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \sigma} \llbracket b \rrbracket \sigma = \text{false}$$

# Beispiel

Schreibe  $S$  als Abkürzung für folgendes Programm.

$$\text{while } [0 < x]^1 \text{ do } ([y := y - 1]^2; [x := x - 1]^3)$$

Dann kann man folgende Urteile herleiten:

- $\langle S, \rho_1 \rangle \rightarrow \langle ([y := y - 1]^2; [x := x - 1]^3); S, \rho_1 \rangle$
- $\langle ([y := y - 1]^2; [x := x - 1]^3); S, \rho_1 \rangle \rightarrow \langle [x := x - 1]^3; S, \rho_2 \rangle$
- $\langle [x := x - 1]^3; S, \rho_2 \rangle \rightarrow \langle S, \rho_3 \rangle$
- $\langle S, \rho_3 \rangle \rightarrow \rho_3$

wobei  $\rho_1 := [x \mapsto 1, y \mapsto 5]$  und  $\rho_2 := [x \mapsto 1, y \mapsto 4]$  und  $\rho_3 := [x \mapsto 0, y \mapsto 4]$ .

Herleitung für das zweite Urteil:

$$\begin{array}{c}
 \text{(ASS)} \frac{}{\langle [y := y - 1]^2, \rho_1 \rangle \rightarrow \rho_2} \\
 \text{(SEQ2)} \frac{}{\langle [y := y - 1]^2; [x := x - 1]^3, \rho_1 \rangle \rightarrow \langle [x := x - 1]^3, \rho_2 \rangle} \\
 \text{(SEQ1)} \frac{}{\langle ([y := y - 1]^2; [x := x - 1]^3); S, \rho_1 \rangle \rightarrow \langle [x := x - 1]^3; S, \rho_2 \rangle}
 \end{array}$$

# Operationelle Semantik

Die operationelle Semantik ist eine kompakte Spezifikation der Programmausführung.

- Die Inferenzregeln selbst spezifizieren noch keinen Algorithmus zur Programmausführung.
- Ein Compiler/Interpreter implementiert sollte die operationelle Semantik möglichst effizient implementieren.
- Die operationelle Semantik ist Spezifikation des Compilers/Interpreters.
- Man kann einen einfachen Interpreter für die Sprache von der operationellen Semantik ableiten.

Für gegebene  $S$  und  $\sigma$  versucht man einen Herleitungsbaum für  $\langle S, \sigma \rangle \rightarrow X$  von unten nach oben aufzubauen und damit  $X$  zu bestimmen.

# Small-Step- und Big-Step-Semantik

Die Urteile  $\langle S, \sigma \rangle \rightarrow \sigma'$  und  $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$  machen Aussagen über einen einzigen Schritt der Programmausführung. Das nennt man eine *Small-Step-Semantik*.

Interessiert man sich nur für die Ausführung des gesamten Programms, kann man auch ein Urteil für eine *Big-Step-Semantik* definieren:

$$(BIG1) \frac{\langle S, \sigma \rangle \rightarrow \sigma'}{\langle S, \sigma \rangle \Downarrow \sigma'}$$

$$(BIG2) \frac{\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \quad \langle S', \sigma' \rangle \Downarrow \sigma''}{\langle S, \sigma \rangle \Downarrow \sigma''}$$

Das Urteil  $\langle S, \sigma \rangle \Downarrow \sigma'$  sagt: Wenn man  $S$  mit Anfangszustand  $\sigma$  laufen lässt, dann terminiert die Programmausführung im Endzustand  $\sigma'$ .



# Datenflussanalyse

Analyse der möglichen Werte der Variablen in einem Programm

- statisch, d.h. zur Compilierzeit
- approximativ; allgemein ist die Frage welche Werte eine Variable annehmen kann unentscheidbar (Satz von Rice)

Wir betrachten:

- Liveness
- Reaching Definitions
- Available Expressions

Typische Anwendungen:

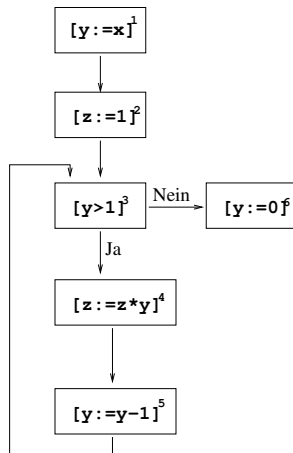
- Programmoptimierung in Compilern
- Verifikation einfacher Korrektheitseigenschaften: keine NullPointerExceptions, vollständige Bereinigung von CGI-Parametern, ...

# Kontrollflussgraph

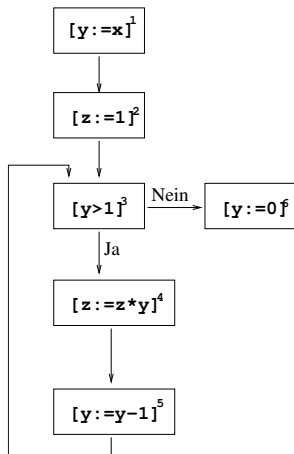
Der *Kontrollflussgraph* eines Programms enthält alle potentiellen Ausführungsfolgen der elementaren Anweisungen eines Programms.

```

[y:=x]1;
[z:=1]2;
while [1<y]3 do (
    [z:=z*y]4;
    [y:=y-1]5
); [y:=0]6
    
```



# Kontrollflussgraph



- Knoten des Kontrollflussgraphen sind die elementaren Anweisungen, also die, die ein Label tragen;
- Knoten, die bei der Ausführung unmittelbar aufeinanderfolgen *können*, werden durch Kanten verbunden.
- Bei Fallunterscheidungen (in if und while) geht man unabhängig von der Bedingung davon aus, dass beide Fälle auftreten können.
- Bei Fallunterscheidungen können die Kanten mit Antworten beschriftet werden.

# Liveness von Variablen

Eine Programmvariable ist an einem bestimmten Programmpunkt in einer Programmausführung *live*, wenn ihr Wert später noch gelesen wird.

Anders gesagt, kann man eine Variable, die *nicht* live ist, ungestraft überschreiben.

## Anwendungen:

- Zuweisungen zu Variablen, die nach der Zuweisung nicht live sind, sind redundant und können eliminiert werden.
- Zwei Variablen, die nie gleichzeitig live sind, können zu einer einzigen verschmolzen werden.

## Liveness von Variablen: Anwendungen

**Beispiel:** Im linken Programm sind  $x$  und  $y$  niemals gleichzeitig live. Die beiden Variablen können zu einer verschmolzen werden (siehe rechts).

|                       |                       |
|-----------------------|-----------------------|
| $[y:=0]^1;$           |                       |
| while $[x<10]^2$ do ( | while $[x<10]^2$ do ( |
| $[y:=x+1]^3;$         | $[x:=x+1]^3;$         |
| $[z:=z+y]^4;$         | $[z:=z+x]^4;$         |
| $[x:=2*y]^5$          | $[x:=2*x]^5$          |
| ); $[r:=z]^6$         | ); $[r:=z]^6$         |

Solches Verschmelzen ist für Compiler wichtig, da Prozessoren nur wenige Register haben (z.B. 8 Register auf x86).

# Liveness-Analyse

Wir wollen für jeden Programmpunkt  $\ell$  zwei Mengen bestimmen:

- $LV_{entry}(\ell)$ : Variablen, die am Eingang von  $\ell$  live sind oder möglicherweise live sein könnten.
- $LV_{exit}(\ell)$ : Variablen, die am Ausgang von  $\ell$  live sind oder möglicherweise live sein könnten.

Beispiel:

$[y:=0]^1; \text{while } [x < 10]^2 \text{ do } ([y:=x+1]^3; [z:=z+y]^4; [x:=2*y]^5); [r:=z]^6$

| $\ell$ | $LV_{entry}$ | $LV_{exit}$ |
|--------|--------------|-------------|
| 1      | $\{x, z\}$   | $\{x, z\}$  |
| 2      | $\{x, z\}$   | $\{x, z\}$  |
| 3      | $\{x, z\}$   | $\{y, z\}$  |
| 4      | $\{y, z\}$   | $\{y, z\}$  |
| 5      | $\{y, z\}$   | $\{x, z\}$  |
| 6      | $\{z\}$      | $\emptyset$ |

# Liveness-Analyse

Vom Kontrollflussgraphen des gegebenen Programms können wir hinreichende Gleichungen für die  $LV$ -Mengen aufstellen.

- Wir interessieren uns für eine möglichst genaue Analyse, in der die  $LV$ -Mengen möglichst klein sind.  
Es wäre korrekt (aber nicht interessant) für jede  $LV$ -Menge die Menge aller Variablen zu wählen.
- Für jedes  $\ell$  liefert die Anweisung mit Label  $\ell$  eine Gleichung für  $LV_{entry}(\ell)$ .

Die Anweisung  $[y := x+1]^3$  liefert zum Beispiel

$$LV_{entry}(3) = \{x\} \cup (LV_{exit}(3) \setminus \{y\})$$

- Die Menge  $LV_{exit}(\ell)$  sollte gleich der Vereinigung der Entry-Mengen aller Nachfolgerknoten von  $\ell$  im Kontrollflussgraphen sein.

# Gleichungssystem für das Beispielprogramm

$[y:=0]^1; \text{while } [x<10]^2 \text{ do } ([y:=x+1]^3; [z:=z+y]^4; [x:=2*y]^5); [r:=z]^6$

$$\begin{aligned}
 LV_{entry}(1) &= LV_{exit}(1) \setminus \{y\} \\
 LV_{entry}(2) &= LV_{exit}(2) \cup \{x\} \\
 LV_{entry}(3) &= (LV_{exit}(3) \setminus \{y\}) \cup \{x\} \\
 LV_{entry}(4) &= (LV_{exit}(4) \setminus \{z\}) \cup \{z, y\} \\
 LV_{entry}(5) &= (LV_{exit}(5) \setminus \{x\}) \cup \{y\} \\
 LV_{entry}(6) &= (LV_{exit}(6) \setminus \{r\}) \cup \{z\} \\
 LV_{exit}(1) &= LV_{entry}(2) \\
 LV_{exit}(2) &= LV_{entry}(3) \cup LV_{entry}(6) \\
 LV_{exit}(3) &= LV_{entry}(4) \\
 LV_{exit}(4) &= LV_{entry}(5) \\
 LV_{exit}(5) &= LV_{entry}(2) \\
 LV_{exit}(6) &= \emptyset
 \end{aligned}$$



## Kleinste Lösung

- Wir interessieren uns für die kleinste Lösung dieser Gleichungen. (d.h. die Lösung, die alle unbekannten Mengen möglichst klein macht)
- Jede Lösung des Gleichungssystems ist korrekt in dem Sinne, dass alle live Variablen erfasst sind.
- Es gibt Lösungen, die mehr live Variablen aufführen, als tatsächlich vorhanden sind, z.B. bleiben die Gleichungen gültig, wenn man die Variable  $r$  zu allen Mengen außer denen für Label 6 hinzunimmt.
- Auch die kleinste Lösung kann überflüssige Einträge enthalten.  
Beispiel:

$$(\text{while } [\text{true}]^1 \text{ do } [\text{skip}]^2); [r := x]^3$$

Hier gilt selbst für die kleinste Lösung  $LV_{entry}(1) = \{x\}$ , obwohl tatsächlich nie wieder von  $x$  gelesen wird.

# Berechnung der kleinsten Lösung

Benutze eine Tabelle mit einem Eintrag für jede gesuchte Menge.

- Initialisiere alle Einträge mit  $\emptyset$
- Betrachte dann jeden Tabelleneintrag, berechne die rechte Seite der Gleichung für diesen Eintrag mit den aktuellen Werten in der Tabelle und aktualisiere den Tabelleneintrag mit dem Ergebnis.
- Wiederhole bis sich nichts mehr ändert.

Im Beispiel:

| $\ell$ | $LV_{exit}(\ell)$ | $LV_{entry}(\ell)$ |                    | $\ell$ | $LV_{exit}(\ell)$ | $LV_{entry}(\ell)$ |                    |
|--------|-------------------|--------------------|--------------------|--------|-------------------|--------------------|--------------------|
| 6      | $\emptyset$       | $\emptyset$        | $\rightsquigarrow$ | 6      | $\emptyset$       | $\{z\}$            | $\rightsquigarrow$ |
| 5      | $\emptyset$       | $\emptyset$        |                    | 5      | $\emptyset$       | $\{y, z\}$         |                    |
| 4      | $\emptyset$       | $\emptyset$        |                    | 4      | $\{y, z\}$        | $\{y, z\}$         |                    |
| 3      | $\emptyset$       | $\emptyset$        |                    | 3      | $\{y, z\}$        | $\{x, z\}$         |                    |
| 2      | $\emptyset$       | $\emptyset$        |                    | 2      | $\{x, z\}$        | $\{x, z\}$         |                    |
| 1      | $\emptyset$       | $\emptyset$        |                    | 1      | $\{x, z\}$        | $\{x, z\}$         |                    |

# Berechnung der kleinsten Lösung

| $\ell$       | $LV_{exit}(\ell)$ | $LV_{entry}(\ell)$ |            | $\ell$ | $LV_{exit}(\ell)$ | $LV_{entry}(\ell)$ |
|--------------|-------------------|--------------------|------------|--------|-------------------|--------------------|
| 6            | $\emptyset$       | $\{z\}$            |            | 6      | $\emptyset$       | $\{z\}$            |
| 5            | $\{x, z\}$        | $\{y, z\}$         |            | 5      | $\{x, z\}$        | $\{y, z\}$         |
| $\leadsto$ 4 | $\{y, z\}$        | $\{y, z\}$         | $\leadsto$ | 4      | $\{y, z\}$        | $\{y, z\}$         |
| 3            | $\{y, z\}$        | $\{x, z\}$         |            | 3      | $\{y, z\}$        | $\{x, z\}$         |
| 2            | $\{x, z\}$        | $\{x, z\}$         |            | 2      | $\{x, z\}$        | $\{x, z\}$         |
| 1            | $\{x, z\}$        | $\{x, z\}$         |            | 1      | $\{x, z\}$        | $\{x, z\}$         |

Ein  $\leadsto$ -Schritt entspricht hier einem Durchgang aller Einträge von links nach rechts und von oben nach unten.

## Berechnung der kleinsten Lösung

Andere Durchlaufreihenfolgen liefern das gleiche Ergebnis, können aber mehr (oder auch weniger) Iterationen brauchen.

Bei der Liveness-Analyse sollte man möglichst rückwärts, also entgegen des Programmflusses vorgehen.

# Live-Variablen Gleichungen

Allgemein können die Liveness-Datenflussgleichungen für ein gegebenes Programm nach folgendem Schema aufgestellt werden.

$$\begin{aligned} LV_{exit}(\ell) &= \bigcup_{\ell': \ell \rightarrow \ell'} LV_{entry}(\ell') \\ LV_{entry}(\ell) &= (LV_{exit}(\ell) \setminus kill_{LV}(B^\ell)) \cup gen_{LV}(B^\ell) \end{aligned}$$

Hierbei ist  $B^\ell$  das durch  $\ell$  beschriftete Programmstück (“Block”), sowie:

$$\begin{aligned} kill_{LV}([x:=a]^\ell) &= \{x\} \\ kill_{LV}([\text{skip}]^\ell) &= \emptyset \\ kill_{LV}([b]^\ell) &= \emptyset \\ gen_{LV}([\text{skip}]^\ell) &= \emptyset \\ gen_{LV}([x:=a]^\ell) &= FV(a) \\ gen_{LV}([b]^\ell) &= FV(b) \end{aligned}$$

Es bezeichnet  $FV(a)$  die Menge aller Variablen, die im Ausdruck  $a$  vorkommen.

# Live-Variablen Gleichungen

Die Gleichungen drücken aus

- an welche Programmpunkten Variablen gelesen werden und somit direkt live sind; und
- welche Variablen in den Vorgängern einer elementaren Anweisung live sein können, wenn sie in einem Nachfolger live sind.

Für die *LV*-Mengen gilt “kleiner = genauer”; wir berechnen also die kleinste Lösung des Gleichungssystems.

Die kleinste Lösung kann auch allgemein wie im Beispiel berechnet werden: Tabelle mit  $\emptyset$  initialisieren, Gleichungen iterieren bis sich nichts mehr ändert

# Reaching Definitions

## Definition

Eine Zuweisung  $[x := a]^{\ell}$  *erreicht* einen bestimmten Programmpunkt  $p$ , wenn es eine Programmausführung gibt, die  $p$  über die Wertzuweisung erreicht, so dass die Variable  $x$  zwischen der Wertzuweisung und  $p$  nicht verändert wird.

## Beispiel:

$[y := x]^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } ([z := z * y]^4; [y := y - 1]^5); [y := 0]^6$

- Die Zuweisung 1 erreicht 3.
- Die Zuweisung 4 erreicht 3.
- Die Zuweisung 2 erreicht 5 nicht.

## Reaching Definitions: Anwendungen

- **Verifikation:** Wird Variable  $x$  stets initialisiert?  
 Füge eine neue Zuweisung  $[x := ?]^\ell$  an den Anfang des Programms an. Diese Zuweisung darf keine Programmstelle erreichen, in der  $x$  gelesen wird.
- **Optimierung: Konstantenpropagation**  
 Wenn die einzige Zuweisung für  $x$ , die einen Programmpunkt erreicht, eine Konstanten-Zuweisung  $[x := c]^\ell$  ist, dann können wir an diesem Programmpunkt  $x$  durch  $c$  ersetzen.
- **Optimierung: Hoisting**  
 Wenn die Anweisung  $[x := y * z]$  nur von Zuweisungen für  $y$  und  $z$  erreicht wird, die vor der `while`-Schleife stehen, dann können wir sie aus der Schleife herausziehen.

|   |   |
|---|---|
| while b do (<br>$x := y * z;$<br>...)<br> | $x := y * z;$<br>while b do (<br>...)<br> |
|---|---|



# Reaching Definitions

Für jeden Programmpunkt  $\ell$  bestimmen wir zwei Mengen:

- $RD_{entry}(\ell)$ : Zuweisungen, die den Eingang von  $\ell$  erreichen oder möglicherweise erreichen könnten.
- $RD_{exit}(\ell)$ : Zuweisungen, die den Ausgang von  $\ell$  erreichen oder möglicherweise erreichen könnten.

Je kleiner diese Mengen gewählt werden können, desto genauer ist die Analyse.

Notation: Wir schreiben kurz  $\ell$  für eine Zuweisung  $[x := a]^\ell$ .

# Reaching Definitions: Datenflussgleichungen

Allgemein können die RD-Gleichungen für ein gegebenes Programm nach folgendem Schema aufgestellt werden.

$$\begin{aligned} RD_{entry}(\ell) &= \bigcup_{\ell': \ell' \rightarrow \ell} RD_{exit}(\ell') \\ RD_{exit}(\ell) &= (RD_{entry}(\ell) \setminus kill_{RD}(B^\ell)) \cup gen_{RD}(B^\ell) \end{aligned}$$

$$\begin{aligned} kill_{RD}([x:=a]^\ell) &= \{\ell' \mid \ell' \text{ ist eine Zuweisung an } x\} \\ kill_{RD}([skip]^\ell) &= \emptyset \\ kill_{RD}([b]^\ell) &= \emptyset \\ gen_{RD}([skip]^\ell) &= \emptyset \\ gen_{RD}([x:=a]^\ell) &= \{\ell\} \\ gen_{RD}([b]^\ell) &= \emptyset \end{aligned}$$

Annahme: Keine Gleichung erreicht den Anfangspunkt des Programms.

# Reaching Definitions: Datenflussgleichungen

Die Gleichungen drücken aus

- welche Definitionen im Programm gemacht werden; und
- wie Definitionen von den Vorgängern zu den Nachfolgern einer elementaren Anweisung propagiert werden.

Auch für die *RD*-Mengen gilt “kleiner = genauer”.

Die kleinste Lösung des Gleichungssystems kann mit dem gleichen Verfahren wie für Liveness bestimmt werden.

Information wird hier vorwärts und nicht rückwärts propagiert. Die Berechnung geht am schnellsten, wenn man die Gleichungen in Reihenfolge des Programmflusses durchgeht.

# Reaching Definitions: Beispiel

$[y:=x]^1; [z:=1]^2; \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5); [y:=0]^6$

$$RD_{\text{entry}}(1) = \emptyset$$

$$RD_{\text{entry}}(2) = RD_{\text{exit}}(1)$$

$$RD_{\text{entry}}(3) = RD_{\text{exit}}(2)$$

$$\cup RD_{\text{exit}}(5)$$

$$RD_{\text{entry}}(4) = RD_{\text{exit}}(3)$$

$$RD_{\text{entry}}(5) = RD_{\text{exit}}(4)$$

$$RD_{\text{entry}}(6) = RD_{\text{exit}}(3)$$

$$RD_{\text{exit}}(1) = (RD_{\text{entry}}(1) \setminus \{1, 5, 6\}) \cup \{1\}$$

$$RD_{\text{exit}}(2) = (RD_{\text{entry}}(2) \setminus \{2, 4\}) \cup \{2\}$$

$$RD_{\text{exit}}(3) = RD_{\text{entry}}(3)$$

$$RD_{\text{exit}}(4) = (RD_{\text{entry}}(4) \setminus \{2, 4\}) \cup \{4\}$$

$$RD_{\text{exit}}(5) = (RD_{\text{entry}}(5) \setminus \{1, 5, 6\}) \cup \{5\}$$

$$RD_{\text{exit}}(6) = (RD_{\text{entry}}(6) \setminus \{1, 5, 6\}) \cup \{6\}$$

# Berechnung der kleinsten Lösung

$[y:=x]^1; [z:=1]^2; \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5); [y:=0]^6$

| $\ell$ | $RD_{entry}(\ell)$ | $RD_{exit}(\ell)$ |                    | $\ell$ | $RD_{entry}(\ell)$ | $RD_{exit}(\ell)$ |                    |
|--------|--------------------|-------------------|--------------------|--------|--------------------|-------------------|--------------------|
| 1      | $\emptyset$        | $\emptyset$       | $\rightsquigarrow$ | 1      | $\emptyset$        | $\{1\}$           | $\rightsquigarrow$ |
| 2      | $\emptyset$        | $\emptyset$       |                    | 2      | $\{1\}$            | $\{1, 2\}$        |                    |
| 3      | $\emptyset$        | $\emptyset$       |                    | 3      | $\{1, 2\}$         | $\{1, 2\}$        |                    |
| 4      | $\emptyset$        | $\emptyset$       |                    | 4      | $\{1, 2\}$         | $\{1, 4\}$        |                    |
| 5      | $\emptyset$        | $\emptyset$       |                    | 5      | $\{1, 4\}$         | $\{4, 5\}$        |                    |
| 6      | $\emptyset$        | $\emptyset$       |                    | 6      | $\{1, 2, 4, 5\}$   | $\{2, 4, 6\}$     |                    |

# Berechnung der kleinsten Lösung, Forts.

$$[y:=x]^1; [z:=1]^2; \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5); [y:=0]^6$$

| $\ell$       | $RD_{entry}(\ell)$ | $RD_{exit}(\ell)$ |            | $\ell$ | $RD_{entry}(\ell)$ | $RD_{exit}(\ell)$ |
|--------------|--------------------|-------------------|------------|--------|--------------------|-------------------|
| 1            | $\emptyset$        | $\{1\}$           |            | 1      | $\emptyset$        | $\{1\}$           |
| 2            | $\{1\}$            | $\{1, 2\}$        |            | 2      | $\{1\}$            | $\{1, 2\}$        |
| $\leadsto$ 3 | $\{1, 2, 4, 5\}$   | $\{1, 2, 4, 5\}$  | $\leadsto$ | 3      | $\{1, 2, 4, 5\}$   | $\{1, 2, 4, 5\}$  |
| 4            | $\{1, 2, 4, 5\}$   | $\{1, 4, 5\}$     |            | 4      | $\{1, 2, 4, 5\}$   | $\{1, 4, 5\}$     |
| 5            | $\{1, 4, 5\}$      | $\{4, 5\}$        |            | 5      | $\{1, 4, 5\}$      | $\{4, 5\}$        |
| 6            | $\{1, 2, 4, 5\}$   | $\{2, 4, 6\}$     |            | 6      | $\{1, 2, 4, 5\}$   | $\{2, 4, 6\}$     |

# Available Expressions

Wir interessieren uns jetzt dafür, welche arithmetischen Ausdrücke an einem bestimmten Programmpunkt *verfügbar* (available) sind, also nicht neu berechnet werden müssen, wenn man bereit ist, berechnete Ausdrücke in einer Tabelle abzuspeichern.

$$[x:=a+b]^1; [y:=a*b]^2; \text{while } [y>a+b]^3 \text{ do } ([a:=a+1]^4; [x:=a+b]^5)$$

Beim Test  $[y>a+b]^3$  ist der Ausdruck  $a+b$  verfügbar, müsste also dort nicht neu berechnet werden. Man könnte das Programm also durch folgende effizientere Version ersetzen:

$$[mem:=a+b]^{1a}; [x:=mem]^1; [y:=a*b]^2; \\ \text{while } [y>mem]^3 \text{ do } ([a:=a+1]^4; [mem:=a+b]^{5a}; [x:=mem]^5)$$

# Available Expressions

Gegeben sei ein Programm, in dem wir die Verfügbarkeit von Ausdrücken analysieren wollen.

Schreibe  $AExp$  für die Menge aller arithmetischer Ausdrücke, die im gegebenen Program vorkommen (evtl. als Teilausdruck eines größeren arithmetischen Ausdrucks), abgesehen von Konstanten und Variablen.

Für jeden Programmpunkt  $\ell$  berechnen wir zwei Mengen:

- $AE_{entry}(\ell) \subseteq AExp$ : Eine Menge arithmetischer Ausdrücke, die sicher am Eingang von  $\ell$  verfügbar sind.
- $AE_{exit}(\ell) \subseteq AExp$ : Eine Menge arithmetischer Ausdrücke, die sicher am Ausgang von  $\ell$  verfügbar sind.

Dieses Mal entsprechen größere Mengen genauerer Information.



# Datenflussgleichungen

$$\begin{aligned}
 AE_{entry}(\ell) &= \begin{cases} \emptyset & \ell \text{ ist Einstiegslabel} \\ \bigcap_{\ell': \ell' \rightarrow \ell} AE_{exit}(\ell') & \text{sonst} \end{cases} \\
 AE_{exit}(\ell) &= (AE_{entry}(\ell) \setminus kill_{AE}(B^\ell)) \cup gen_{AE}(B^\ell)
 \end{aligned}$$

wobei

$$\begin{aligned}
 kill_{AE}([x := a]^\ell) &= \{a' \in AExp \mid x \in FV(a')\} \\
 kill_{AE}([\text{skip}]^\ell) &= \emptyset \\
 kill_{AE}([b]^\ell) &= \emptyset \\
 gen_{AE}([x := a]^\ell) &= \{a' \in AExp \mid a' \text{ Teilausdruck von } a \text{ und } x \notin FV(a')\} \\
 gen_{AE}([\text{skip}]^\ell) &= \emptyset \\
 gen_{AE}([b]^\ell) &= \{a' \in AExp \mid a' \text{ Teilausdruck von } b\}
 \end{aligned}$$

(Ein Teilausdruck entspricht einem Teilbaum des Syntaxbaums.)

# Datenflussgleichungen im Beispiel

$[x:=a+b]^1; [y:=a*b]^2; \text{while } [y>a+b]^3 \text{ do } ([a:=a+1]^4; [x:=a+b]^5)$

| $\ell$ | $kill_{AE}$         | $gen_{AE}$  |
|--------|---------------------|-------------|
| 1      | $\emptyset$         | $\{a+b\}$   |
| 2      | $\emptyset$         | $\{a*b\}$   |
| 3      | $\emptyset$         | $\{a+b\}$   |
| 4      | $\{a+b, a*b, a+1\}$ | $\emptyset$ |
| 5      | $\emptyset$         | $\{a+b\}$   |

$$\begin{aligned}
 AE_{entry}(1) &= \emptyset \\
 AE_{entry}(2) &= AE_{exit}(1) \\
 AE_{entry}(3) &= AE_{exit}(2) \cap AE_{exit}(5) \\
 AE_{entry}(4) &= AE_{exit}(3) \\
 AE_{entry}(5) &= AE_{exit}(4)
 \end{aligned}$$

$$\begin{aligned}
 AE_{exit}(1) &= AE_{entry}(1) \cup \{a+b\} \\
 AE_{exit}(2) &= AE_{entry}(2) \cup \{a*b\} \\
 AE_{exit}(3) &= AE_{entry}(3) \cup \{a+b\} \\
 AE_{exit}(4) &= AE_{entry}(4) \setminus \{a+b, a*b, a+1\} \\
 AE_{exit}(5) &= AE_{entry}(5) \cup \{a+b\}
 \end{aligned}$$

$$AExp = \{a+b, a*b, a+1\}$$

# Größte Lösung

Auch hier ist jede Lösung der Datenflussgleichung gerechtfertigt, allerdings sind wir an möglichst großen  $AE$ -Mengen interessiert. Wir berechnen daher die *größte* Lösung. Man erhält sie wie die kleinste Lösung, initialisiert allerdings die dynamischen Mengen jeweils mit der Menge *aller Ausdrücke* (die im Programm vorkommen) anstatt mit  $\emptyset$ .

Man erhält:

| $\ell$ | $AE_{entry}(\ell)$ | $AE_{exit}(\ell)$ |
|--------|--------------------|-------------------|
| 1      | $AExp$             | $AExp$            |
| 2      | $AExp$             | $AExp$            |
| 3      | $AExp$             | $AExp$            |
| 4      | $AExp$             | $AExp$            |
| 5      | $AExp$             | $AExp$            |

$\rightsquigarrow \dots \rightsquigarrow$

| $\ell$ | $AE_{entry}(\ell)$ | $AE_{exit}(\ell)$ |
|--------|--------------------|-------------------|
| 1      | $\emptyset$        | $\{a+b\}$         |
| 2      | $\{a+b\}$          | $\{a+b, a*b\}$    |
| 3      | $\{a+b\}$          | $\{a+b\}$         |
| 4      | $\{a+b\}$          | $\emptyset$       |
| 5      | $\emptyset$        | $\{a+b\}$         |

# Größte vs. kleinste Lösung

Im Beispiel stimmen größte und kleinste Lösung zufällig überein. Das liegt daran, dass die Schleife alle Ausdrücke “kilt”.

Anders bei diesem Beispiel:

$$[x:=a+b]^1; \text{ while } [x>y]^2 \text{ do } [x:=x-1]^3$$

Hier gelten die Gleichungen

$$\begin{aligned} AE_{entry}(1) &= \emptyset \\ AE_{entry}(2) &= AE_{exit}(1) \cap AE_{exit}(3) \\ AE_{entry}(3) &= AE_{exit}(2) \end{aligned}$$

$$\begin{aligned} AE_{exit}(1) &= (AE_{entry}(1) \setminus \{x-1\}) \cup \{a+b\} \\ AE_{exit}(2) &= AE_{entry}(2) \\ AE_{exit}(3) &= AE_{entry}(3) \end{aligned}$$

Die größte Lösung liefert  $AE_{exit}(3) = \{a+b\}$  während bei der kleinsten Lösung  $AE_{exit}(3) = \emptyset$  gilt.

# Fixpunkttheorie

Verschiedene Definitionen und Algorithmen der Vorlesung sind Anwendungsbeispiele für *Fixpunkte* und ihre Berechnung.

- Datenflussanalyse: Iteration zur Berechnung der kleinsten bzw. größten Lösung
- Model-Checking: Semantik von CTL, Labelling-Algorithmus, symbolisches Model-Checking

Mit etwas *Fixpunkttheorie* können die Eigenschaften solcher Verfahren allgemein verstanden werden, z.B.:

- Warum finden die iterativen Algorithmen für die Datenflussanalyse die kleinste bzw. die größte Lösung?
- Ist bei der Iteration die Reihenfolge wichtig?

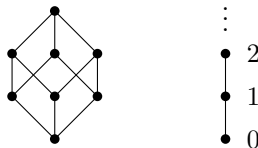
# Halbordnung

## Definition

Eine *Halbordnung* (partial order) ist eine Menge  $L$  mit einer binären Relation  $\sqsubseteq$ , die folgende Eigenschaften hat.

- Reflexivität:  $\forall x \in L. x \sqsubseteq x$
- Transitivität:  $\forall x, y, z \in L. (x \sqsubseteq y) \wedge (y \sqsubseteq z) \implies x \sqsubseteq z.$
- Antisymmetrie:  $\forall x, y \in L. (x \sqsubseteq y) \wedge (y \sqsubseteq x) \implies x = y.$

Darstellung als *Hasse-Diagramme*, z.B.:



Die Knoten im Hasse-Diagramm sind die Elemente von  $L$ ; eine Kante von  $x \in L$  zu einem höher liegenden  $y \in L$  drückt  $x \sqsubseteq y$  aus. Alle Kanten, die aus Reflexivität oder Transitivität folgen, werden weggelassen.

# Obere und untere Schranken

Sei  $(L, \sqsubseteq)$  eine Halbordnung.

## Definition (obere Schranke)

Ein Element  $y \in L$  ist eine **obere Schranke** für eine Menge  $X \subseteq L$  falls  $\forall x \in X. x \sqsubseteq y$  gilt.

## Definition (kleinste obere Schranke)

Ein Element  $y \in L$  ist eine **kleinste obere Schranke** für eine Menge  $X \subseteq L$  falls gilt:

1.  $y$  ist eine obere Schranke für  $X$
2. Jede obere Schranke  $z$  für  $X$  ist größer-gleich  $y$ , d.h.  $y \sqsubseteq z$ .

Analog für (größte) untere Schranken.

# Vollständiger Verband

## Definition (Vollständiger Verband)

Eine Halbordnung  $(L, \sqsubseteq)$  heißt *vollständiger Verband*, wenn jede Teilmenge  $X \subseteq L$  sowohl eine kleinste obere Schranke in  $L$  als auch eine größte untere Schranke in  $L$  hat.

Schreibe  $\bigsqcup X$  für die kleinste obere Schranke von  $X$  und  $\bigsqcap X$  für die größte untere Schranke  $X$ .

Abkürzungen:

$$\begin{aligned}\perp &:= \bigsqcap L & \top &:= \bigsqcup L \\ x \wedge y &:= \bigsqcap \{x, y\} & x \vee y &:= \bigsqcup \{x, y\}\end{aligned}$$

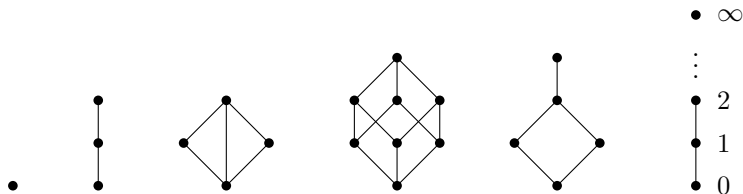
( $\perp$  und  $\top$  sind kleinstes und größtes Element des Verbands.)

Die *Höhe* eines Verbandes ist die Länge des längsten Pfades von  $\perp$  nach  $\top$  im Hasse-Diagramm.

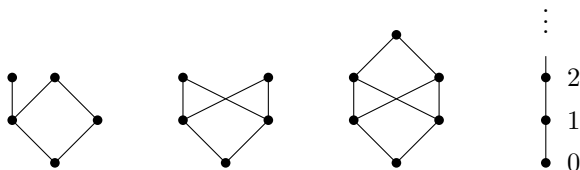


# Beispiele

## Vollständige Verbände:



## Keine vollständigen Verbände:

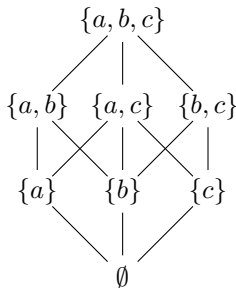


# Beispiele

Für jede Menge  $U$  ist der *Potenzmengenverband*  $(\mathcal{P}(U), \subseteq)$  ein vollständiger Verband.

In diesem Fall gilt  $\perp = \emptyset$ ,  $\top = U$ ,  $\sqcup X = \bigcup X$ ,  $\sqcap X = \bigcap X$ .

Beispiel für  $U = \{1, 2, 3\}$ :



# Monotone Funktionen

Sei  $(L, \sqsubseteq)$  eine Halbordnung.

## Definition (Monotonie)

Eine Funktion  $f: L \rightarrow L$  heißt *monoton* wenn gilt:

$$\forall x, y \in L. x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$$

Beispiele für  $(\mathcal{P}(\mathbb{N}), \subseteq)$ :

- $f(X) = X \cup \{1\}$  ist monoton.
- $f(X) = X \setminus \{1\}$  ist monoton.
- Sind  $f$  und  $g$  monoton, so ist auch  $h(X) = g(f(X))$  monoton.
- $f(X) = \mathbb{N} \setminus X$  ist nicht monoton.

# Fixpunkte

## Definition (Fixpunkt)

Ein Element  $x \in L$  mit der Eigenschaft  $f(x) = x$  heißt *Fixpunkt von  $f$* .

Beispiele für  $(\mathcal{P}(\mathbb{N}), \subseteq)$ :

- Jede Menge, die 1 enthält ist ein Fixpunkt von  $f(X) = X \cup \{1\}$ .
- Jede Menge, die 1 nicht enthält ist ein Fixpunkt von  $f(X) = X \setminus \{1\}$ .
- $f(X) = \mathbb{N} \setminus X$  hat keinen Fixpunkt.

# Fixpunkte

## Theorem (Kleene)

*In einem vollständigen Verband endlicher Höhe hat jede monotone Funktion  $f$  einen Fixpunkt. Der kleinste Fixpunkt ist*

$$\text{lfp}(f) := \bigsqcup \{f^i(\perp) \mid i \in \mathbb{N}\} .$$

Wegen der Monotonie von  $f$  gilt:

$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq f^3(\perp) \sqsubseteq \dots$$

Da der Verband endliche Höhe hat, muss es ein  $k$  geben mit  $f^k(\perp) = f^{k+1}(\perp)$ , d.h.  $f^k(\perp)$  ist ein Fixpunkt von  $f$ .

Das ist der kleinste Fixpunkt. Sei  $x$  ein beliebiger Fixpunkt. Man zeigt durch Induktion über  $i$ , dass  $f^i(\perp) \sqsubseteq f^i(x)$  für alle  $i$  gilt. Da  $x$  ein Fixpunkt ist, gilt  $f^k(x) = x$ , also  $f^k(\perp) \sqsubseteq f^k(x) = x$ . Der Fixpunkt  $f^k(\perp)$  ist kleiner-gleich  $x$ .

# Fixpunkte

## Theorem (Kleene)

*In einem vollständigen Verband endlicher Höhe hat jede monotone Funktion  $f$  einen Fixpunkt. Der größte Fixpunkt ist*

$$gfp(f) := \bigsqcap \{f^i(\top) \mid i \in \mathbb{N}\} .$$

Wegen der Monotonie von  $f$  gilt:

$$\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsubseteq f^3(\top) \sqsubseteq \dots$$

Da der Verband endliche Höhe hat, muss es ein  $k$  geben mit  $f^k(\top) = f^{k+1}(\top)$ , d.h.  $f^k(\top)$  ist ein Fixpunkt von  $f$ .

Das ist der größte Fixpunkt. Sei  $x$  ein beliebiger Fixpunkt. Man zeigt durch Induktion über  $i$ , dass  $f^i(\top) \sqsupseteq f^i(x)$  für alle  $i$  gilt. Da  $x$  ein Fixpunkt ist, gilt  $f^k(x) = x$ , also  $f^k(\top) \sqsupseteq f^k(x) = x$ . Der Fixpunkt  $f^k(\top)$  ist größer-gleich  $x$ .

# Fixpunkte

## Theorem (Knaster-Tarski)

*Sei  $(L, \sqsubseteq)$  ein vollständiger Verband und  $f : L \rightarrow L$  monoton. Dann bildet die Menge dieser Fixpunkte selbst einen vollständigen Verband.*

- Der kleinste Fixpunkt ist gegeben durch  $\bigcap \{x \mid F(x) \sqsubseteq x\}$   
(Beweis an der Tafel).
- Der größte Fixpunkt ist gegeben durch  $\bigcup \{x \mid x \sqsubseteq F(x)\}$   
(Beweis an der Tafel).

# Anwendung: Datenflussanalyse

Die Gleichungssysteme für die verschiedenen Datenflussanalysen haben alle die Form

$$X_1 = F_1(X_1, \dots, X_n)$$

$$X_2 = F_2(X_1, \dots, X_n)$$

$$\dots$$

$$X_n = F_n(X_1, \dots, X_n)$$

wobei die  $X_i$  alle Elemente eines Verbands  $U$  und die  $F_i$  monotone Funktionen auf  $U$  sind.

- **Liveness:**  $U = (\mathcal{P}(V), \subseteq)$ , wobei  $V$  die Menge aller Variablen ist, die im gegebenen Programms vorkommen.
- **Reaching Definitions:**  $U = (\mathcal{P}(L), \subseteq)$ , wobei  $L$  die Menge aller Labels des gegebenen Programms ist.
- **Available Expressions:**  $U = (\mathcal{P}(AExp), \subseteq)$ .



# Anwendung: Datenflussanalyse

Das Gleichungssystem kann als eine einzige Gleichung in einem Produktverband von Tupeln formuliert werden.

Angenommen  $V_1, \dots, V_n$  sind vollständige Verbände.

Dann ist auch die Menge

$$V_1 \times \dots \times V_n := \{(v_1, \dots, v_n) \mid v_1 \in V_1, \dots, v_n \in V_n\}$$

ein vollständiger Verband, wenn man die Ordnung punktweise definiert:

$$(v_1, \dots, v_n) \sqsubseteq (w_1, \dots, w_n) \iff v_1 \sqsubseteq w_1 \wedge \dots \wedge v_n \sqsubseteq w_n$$

Es gilt  $\perp = (\perp, \dots, \perp)$ .

## Anwendung: Datenflussanalyse

Das Gleichungssystem kann als eine einzige Gleichung in einem Verband von Tuplen formuliert werden.

Durch

$$F(X_1, \dots, X_n) := (F_1(X_1, \dots, X_n), \dots, F_n(X_1, \dots, X_n))$$

wird eine Funktion  $F: U \times \dots \times U \rightarrow U \times \dots \times U$  definiert.

Aus der Monotonie der  $F_i$  folgt Monotonie von  $F$ .

Man erhält die kleinste (bzw. größte) Lösung des Gleichungssystems (als Vektor von Mengen) durch Berechnung von  $lfp(F)$  (bzw.  $gfp(F)$ ).

# Fixpunktiteration

Berechnung von  $lfp(F)$ :

```
 $x := (\perp, \dots, \perp);$   
repeat  
   $\text{old} := x;$   
   $x := F(x);$   
until  $x = \text{old}$ 
```

Korrektheit folgt aus  $lfp(F) = \bigsqcup \{F^i(\perp) \mid i \in \mathbb{N}\}.$

# Fixpunktiteration

Die Algorithmen zur Datenflussanalyse entsprachen folgendem etwas besseren Algorithmus.

```

 $x_1 := \perp; \dots; x_n := \perp;$ 
repeat
     $\text{old}_1 := x_1; \dots; \text{old}_n := x_n;$ 
     $x_1 := F_1(x_1, \dots, x_n); \dots; x_n := F_n(x_1, \dots, x_n)$ 
until  $x_1 = \text{old}_1 \wedge \dots \wedge x_n = \text{old}_n$ 
    
```

Diesen erhält man mit einer verbesserten Wahl von  $F$ , in der immer schon die “neuesten” Werte für  $X_i$  benutzt werden.

Beispiel für  $n = 2$ :

$$F'(X_1, X_2) := (F_1(X_1, X_2), F_2(F_1(X_1, X_2), X_2))$$

Es ist nicht schwer zu zeigen, dass  $F$  und  $F'$  die gleichen Fixpunkte haben.

## Anwendung: CTL

Die Temporaloperatoren in CTL entsprechen kleinsten und größten Fixpunkten.

Gegeben sei ein endliches Transitionssystem  $(S, \rightarrow)$ .  
Definiere:

$$\text{EX}(B) := \{s \in S \mid \exists s' \in B. s \rightarrow s'\}$$

$$\text{AX}(B) := \{s \in S \mid \forall s' \in S. s \rightarrow s' \implies s' \in B\}$$

Im Potenzmengenverband auf  $S$  haben wir:

- $\text{EF}(B) = \text{lfp}(F)$  für  $F(C) = B \cup \text{EX}(C)$ : Menge aller Zustände, von denen aus ein Pfad existiert, der irgendwann  $B$  erreicht.

$$\emptyset \subseteq B \cup \text{EX}(\emptyset) \subseteq B \cup \text{EX}(B \cup \text{EX}(\emptyset)) \subseteq \dots$$

## Anwendung: CTL, Forts.

- $AG(B) = gfp(F)$  für  $F(C) = B \cap AX(C)$ : Menge aller Zustände, so dass alle ausgehenden Pfade stets in  $B$  bleiben.

$$S \supseteq B \cap AX(S) \supseteq B \cap AX(B \cap AX(S)) \supseteq \dots$$

- $AF(B) = lfp(F)$  für  $F(C) = B \cup AX(C)$ : Menge aller Zustände, so dass alle ausgehenden Pfade irgendwann  $B$  erreichen.
- $EG(B) = gfp(F)$  für  $F(C) = B \cap EX(C)$ : Menge aller Zustände, von denen aus ein Pfad existiert, der stets in  $B$  bleibt.
- $E[-U-]$  und  $A[-U-]$ : Übung

Labelling-Algorithmus und symbolisches Modelchecking sind Beispiele der Fixpunktiteration.

# Programmanalyse und Typsysteme

Ein Typsystem weist verschiedenen Teilen eines Programms einen Typ zu.

Ein Typ erfasst eine bestimmte Eigenschaft, die ein Programmteil zur Laufzeit hat.

Beispiel: In der While-Sprache kann man arithmetischen Ausdrücken den Typ `int` geben, um auszudrücken, dass der Wert zur Laufzeit eine Zahl ist (und nicht etwa ein Boolescher Wert).

Typen können auch kompliziertere Eigenschaften erfassen. Wir erfassen Reaching-Definitions durch ein Typsystem.

## Typsystem für Reaching Definitions

Information über Reaching-Definitions lässt sich durch ein Typsystem erfassen.

Wir verwenden ein *Typurteil* der Form

$$\vdash S : D_1 \rightarrow D_2$$

(lies: “ $S$  hat Typ  $D_1 \rightarrow D_2$ ”), wobei  $S$  ein Programmstück ist und  $D_1, D_2$  Mengen von Zuweisungen sind, wie bei der Reaching-Definition-Analyse.

Die gemeinte Bedeutung von  $\vdash S : D_1 \rightarrow D_2$  ist:

*“Wenn allein die Definitionen aus  $D_1$  das Entry-Label von  $S$  erreichen können, dann sind die Definitionen, die ein Exit-Label von  $S$  erreichen können, alle in  $D_2$  enthalten.”*



## Typregeln für das Typurteil

$$\text{(ASS)} \frac{}{\vdash [x:=a]^\ell: D \rightarrow (D \setminus \{\ell' \mid \ell' \text{ ist Zuweisung an } x\}) \cup \{\ell\}}$$

$$\text{(SKIP)} \frac{}{\vdash [\text{skip}]^\ell: D \rightarrow D}$$

$$\text{(SEQ)} \frac{\vdash S_1: D_1 \rightarrow D_2 \quad \vdash S_2: D_2 \rightarrow D_3}{\vdash S_1; S_2: D_1 \rightarrow D_3}$$

$$\text{(IF)} \frac{\vdash S_1: D_1 \rightarrow D_2 \quad \vdash S_2: D_1 \rightarrow D_2}{\vdash \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2: D_1 \rightarrow D_2}$$

$$\text{(WHILE)} \frac{\vdash S: D \rightarrow D}{\vdash \text{while } [b]^\ell \text{ do } S: D \rightarrow D}$$

$$\text{(SUB)} \frac{\vdash S: D_1 \rightarrow D_2}{\vdash S: D'_1 \rightarrow D'_2} D'_1 \subseteq D_1, D_2 \subseteq D'_2$$

## Beispiel

Wir geben eine Typherleitung für unser Beispielprogramm an:

$[y:=x]^1; [z:=1]^2; \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5); [y:=0]^6$

Schreibe kurz  $D$  für  $\{1, 5, 2, 4\}$ .

Erster Teil der Herleitung:

$$\begin{array}{c}
 \text{(ASS)} \frac{}{\vdash [z:=z*y]^4: D \rightarrow D \setminus \{2\}} \quad \text{(ASS)} \frac{}{\vdash [y:=y-1]^5: D \setminus \{2\} \rightarrow D \setminus \{2, 1\}} \\
 \text{(SEQ)} \frac{}{\vdash [z:=z*y]^4; [y:=y-1]^5: D \rightarrow D \setminus \{2, 1\}} \\
 \text{(SUB)} \frac{}{\vdash [z:=z*y]^4; [y:=y-1]^5: D \rightarrow D} \\
 \text{(WHILE)} \frac{}{\vdash \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5): D \rightarrow D}
 \end{array}$$

## Beispielherleitung

Wenn wir wissen, dass  $\vdash S : D \rightarrow D$  für  $D = \{1, 5, 2, 4\}$  herleitbar ist, dann können wir auch einen Typ für

$$[y:=x]^1; [z:=1]^2; S; [y:=0]^6$$

herleiten.

$$\begin{array}{c}
 \vdash [y:=x]^1 : \emptyset \rightarrow \{1\} \quad \frac{\vdash [z:=1]^2 : \{1\} \rightarrow \{1, 2\}}{\vdash [z:=1]^2 : \{1\} \rightarrow D} \quad \frac{\vdash S : D \rightarrow D \quad \vdash [y:=0]^6 : D \rightarrow D \cup \{6\}}{\vdash S; [y:=0]^6 : D \rightarrow D \cup \{6\}} \\
 \hline
 \vdash [y:=x]^1; [z:=1]^2; S; [y:=0]^6 : \emptyset \rightarrow D \cup \{6\}
 \end{array}$$

Das ist ein Beispiel für *Kompositionalität*: Der Typ eines Programmstück hängt nur von den Typen seiner Teilausdrücke ab. Im Beispiel müssen wir  $S$  nicht kennen, nur seinen Typ.

## Relation zur RD-Analyse

Ist  $RD_{entry/exit}(-)$  Lösung des RD-Gleichungssystems, so lässt sich für jedes Programmstück  $S$  das Typurteil

$$\vdash S : RD_{entry}(init(S)) \rightarrow \bigcup_{\ell \in final(S)} RD_{exit}(\ell)$$

herleiten.

Hier bezeichnet  $init(S)$  das Eingangslabel von  $S$  und  $final(S)$  die Ausgangslabels von  $S$ . Wenn etwa  $S$  ein if-then-else ist, dann gibt es mehr als ein Ausgangslabel.

Das Typsystem kann als Spezifikation der “Reaching-Definitions” verstanden werden, die RD-Analyse als Algorithmus, der herleitbare Urteile berechnet.

## Typprüfung und Typinferenz

Die Frage, ob ein gegebenes Typurteil  $\vdash S: D_1 \rightarrow D_2$  herleitbar ist, heißt *Typprüfung* (type checking).

Die Frage, für ein gegebenes Programm  $S$  einen (möglichst informativen) Typen  $D_1 \rightarrow D_2$  zu berechnen, so dass  $\vdash S: D_1 \rightarrow D_2$  herleitbar ist, heißt *Typinferenz* (type inference).

Man kann direkt einen Algorithmus für Typinferenz konstruieren.

## Ansatz zur Typinferenz

**Beispiel:** Typinferenz für `while [y>1]3 do ([z:=z*y]4; [y:=y-1]5).`

Wenn es eine Typherleitung gibt, dann gibt es eine Herleitung folgender Form:

$$\begin{array}{c}
 \text{(ASS)} \frac{}{\vdash [z:=z*y]^4 : D_7 \rightarrow D_7 \setminus \{2\} \cup \{4\}} \quad \text{(ASS)} \frac{}{\vdash [y:=y-1]^5 : D_8 \rightarrow D_8 \setminus \{1, 6\} \cup \{5\}} \\
 \text{(SUB)} \frac{}{\vdash [z:=z*y]^4 : D_4 \rightarrow D_6} \quad \text{(SUB)} \frac{}{\vdash [y:=y-1]^5 : D_6 \rightarrow D_5} \\
 \text{(SEQ)} \frac{}{\vdash [z:=z*y]^4; [y:=y-1]^5 : D_4 \rightarrow D_5} \\
 \text{(SUB)} \frac{}{\vdash [z:=z*y]^4; [y:=y-1]^5 : D_3 \rightarrow D_3} \\
 \text{(WHILE)} \frac{}{\vdash \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5) : D_3 \rightarrow D_3} \\
 \text{(SUB)} \frac{}{\vdash \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5) : D_1 \rightarrow D_2}
 \end{array}$$

(Aufbau von unten nach oben; in jedem Schritt kann neben (SUB) nur eine einzige Regel angewendet werden; keine oder mehr als eine Anwendung von (SUB) können durch eine einzige ersetzt werden.)

## Ansatz zur Typinferenz

Die Mengen  $D_1, \dots, D_8$  sind noch unbestimmt und die Herleitung ist nur dann korrekt, wenn alle Nebenbedingungen erfüllt sind.

Können wir die Mengen  $D_1, \dots, D_8$  so wählen, dass die Nebenbedingungen aller verwendeten Regeln erfüllt sind?

Im Beispiel müssen dazu die Nebenbedingungen für die Anwendungen von (sub) alle erfüllt sein:

$$\begin{array}{ll} D_4 \subseteq D_7 & (D_7 \setminus \{2\}) \cup \{4\} \subseteq D_6 \\ D_6 \subseteq D_8 & (D_9 \setminus \{1, 6\}) \cup \{5\} \subseteq D_5 \\ D_3 \subseteq D_4 & D_5 \subseteq D_3 \\ D_1 \subseteq D_3 & D_3 \subseteq D_2 \end{array}$$

Gesucht ist eine möglichst kleine Lösung dieser Mengeninklusionen.

## Ansatz zur Typinferenz

Nach dem Satz von Knaster-Tarski ist die kleinste Lösung von  $F(X) = X$  auch die kleinste Lösung von  $F(X) \sqsubseteq X$ .

Die Inklusionen der vorigen Folie können als  $F(X) \sqsubseteq X$  ausgedrückt werden, wenn man  $F(D_1, \dots, D_8)$  als

$$(\emptyset, D_3, D_1 \cup D_5, D_3, (D_9 \setminus \{1, 6\}) \cup \{5\}, (D_7 \setminus \{2\}) \cup \{4\}, D_4, D_6)$$

definiert. (Die beiden Inklusionen  $D_1 \subseteq D_3$  und  $D_5 \subseteq D_3$  wurden zu  $D_1 \cup D_5 \subseteq D_3$  zusammengefasst).

Die kleinste Lösung für  $F(X) = X$  (und damit für  $F(X) \sqsubseteq X$ ) kann dann durch Fixpunktiteration von  $F$  berechnet werden. Das entspricht der RD-Analyse.



## Korrektheit des Typsystems

Wir kommen nun zum Korrektheitsbeweis des *RD*-Typsystems. Dazu erweitern wir die operationelle Semantik, so dass die den aktuellen Programmpunkt tatsächlich erreichenden Zuweisungen aufgezeichnet werden.

- Ein Programmzustand  $\sigma$  ist nicht nur durch eine endliche Abbildung von Variablen auf Zahlen gegeben. Es kommt noch eine Menge von Zuweisungen (wie in der *RD*-Analyse) hinzu.
- Mit  $\sigma.D$  bezeichnen wir diese Menge von Zuweisungen.
- Die Regel (ASS) der operationellen Semantik ist wie folgt anzupassen:

$$\text{(ASS)} \frac{}{\langle [x := a]^\ell, \sigma \rangle \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma][D \mapsto D']}$$

wobei  $D'$  für  $\sigma.D \setminus \{\ell' \mid \ell' \text{ ist Zuweisung und } x\} \cup \{\ell\}$  steht.

- Alle anderen Regeln bleiben unverändert.

## Korrektheit des Typsystems für $RD$

### Korrektheit des $RD$ -Typsystems (bzgl. Big-Step Semantik)

Angenommen  $\vdash S : D_1 \rightarrow D_2$  ist herleitbar und  $\sigma$  ist ein Programmzustand mit  $\sigma.D \subseteq D_1$ .

Dann folgt aus  $\langle S, \sigma \rangle \Downarrow \sigma'$  auch  $\sigma'.D \subseteq D_2$ .

### Korrektheit des $RD$ -Typsystems (bzgl. Small-Step Semantik)

Angenommen  $\vdash S : D_1 \rightarrow D_2$  ist herleitbar und  $\sigma$  ist ein Programmzustand mit  $\sigma.D \subseteq D_1$ .

- Wenn  $\langle S, \sigma \rangle \rightarrow \sigma'$  dann  $\sigma'.D \subseteq D_2$ .
- Wenn  $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$  dann gibt es  $D_m$ , sodass  $\sigma'.D \subseteq D_m$  und  $\vdash S' : D_m \rightarrow D_2$ .

Der Beweis erfolgt durch Induktion über die Herleitung von  $\vdash S : D_1 \rightarrow D_2$ .

## Zusammenfassung: Typsystem für Programmanalyse

- Herleitbare Typurteile werden durch Typisierungsregeln induktiv definiert.
- Meist gibt es eine Typregel für jedes syntaktische Konstrukt und darüberhinaus Anpassungsregeln, wie (SUB).
- Typurteile können semantisch interpretiert werden. Korrektheit besagt, dass jedes herleitbare Typurteil in diesem Sinne semantisch gültig ist.

## Zusammenfassung: Typsystem für Programmanalyse, Forts.

- Durch Rückwärtsanwendung der Typregeln (“Skelett-Herleitung”) können Bedingungen (“Constraints”) generiert werden und durch deren Lösung Typherleitungen automatisch gefunden werden (Typinferenz).
- Ergebnisse einer Programmanalyse können oft in Typherleitungen übersetzt werden.
- Typsysteme erleichtern die Formulierung von semantischen Korrektheitsaussagen.

# Typ- und Effektsysteme

**Typsysteme** eignen sich vor allem zur Analyse von Sprachen mit (first-class) Funktionen.

- funktionale Sprachen SML, OCaml, Haskell, ...
- Selbst untypisierte Sprachen wie Javascript werden mit Typsystemen analysiert.

**Beispiel:** Flow (Facebook) für JavaScript

- Ausdrucksstarke Typsysteme finden Anwendungen in systemnaher Programmierung.

**Beispiel:** Rust (Mozilla)

Durch statische Typanalyse kann automatisch eine bestimmte Klasse von Laufzeitfehlern ausgeschlossen werden (z.B. illegale Speicherzugriffe).

Verschiedene Typsysteme liefern verschiedene Garantien.

# Typ- und Effektsysteme

Im folgenden Abschnitt betrachten wir eine einfache funktionale Sprache.

- Typsystem mit Polymorphie
- Hindley-Milner Typinferenze
- Erweiterungen um Exceptions, Referenzen

Verschiedene Programmanalysen für diese Sprache können durch Typsysteme mit *Effektannotaten* erfasst werden.

- Kontrollflussanalyse
- Exceptions
- Speicherzugriffe

# Funktionale Sprache

## Terme:

$$\begin{aligned}
 e, e_1, e_2 ::= & \ x \mid n \mid \text{true} \mid \text{false} \mid e_1 \text{ op } e_2 \\
 & \mid \text{fun } x \rightarrow e \mid e_1 \ e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \\
 & \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{recfun } f \ x \rightarrow e \\
 \text{op} ::= & \ + \mid - \mid * \mid / \mid < \mid = \mid \& \mid | \mid \dots
 \end{aligned}$$

Hierbei läuft  $x$  über Programmvariablen und  $n$  über ganze Zahlen.

**Konvention:**  $e_1 \ e_2 \ e_3$  steht für  $(e_1 \ e_2) \ e_3$

## Beispielterme:

- `fun x -> x+3`
- `(fun x -> x+3) 5`
- `fun x -> if x=0 then true else false`
- `(fun x -> (x 3)+(x 2)) (fun y -> y*2)`
- `recfun f x -> if x=0 then 1 else x * f (x-1)`

# Beispiele

## Fibonacci, rekursiv:

```
let fib = recfun f x ->
    if x < 2 then 1 else f (x-1) + f (x-2) in
fib 12
```

## Fibonacci, mit Akkumulator:

```
let fib = fun x ->
    let fibacc =
        recfun fa x -> fun y1 -> fun y2 ->
            if x = 0 then y1 else fa (x-1) y2 (y1 + y2) in
        fibacc x 1 1 in
fib 12
```



# Variablen, Substitution

## Gebundene und freie Variablen:

In den Termen

$$\text{fun } x \rightarrow e \quad \text{let } x = e_1 \text{ in } e \quad \text{recfun } f \ x \rightarrow e$$

wird die Variable  $x$  deklariert. Diese Deklaration *bindet* alle freien Vorkommen von Variable  $x$  in  $e$ . Eine Variable ist *frei*, wenn sie nicht gebunden ist. In `recfun` wird analog auch noch  $f$  in  $e$  gebunden.

## Substitution:

Schreibe  $e[x \mapsto e']$  für das Ergebnis der Substitution (Ersetzung) aller freien Vorkommen von  $x$  in  $e$  durch  $e'$ . Dabei werden gebundene Variablen geeignet umbenannt, so dass keine freie Variable in  $e'$  gebunden wird.

Beispiele:

- $(\text{fun } x \rightarrow y \ x)[y \mapsto (\text{fun } z \rightarrow z * 2)] = (\text{fun } x \rightarrow (\text{fun } z \rightarrow z * 2) \ x)$
- $(\text{fun } x \rightarrow y \ x)[y \mapsto (\text{fun } z \rightarrow x * 2)] = (\text{fun } u \rightarrow (\text{fun } z \rightarrow x * 2) \ u)$

# Operationelle Semantik

Bisher haben die Programmterme keine Bedeutung. Wir müssen spezifizieren, welches Ergebnis ihre Auswertung liefern soll.

Dazu definieren wir eine operationelle Semantik (big-step).

Das Urteil

$$e \Downarrow v$$

drückt aus, dass der Programmterm  $e$  bei vollständiger Ausführung den Wert  $v$  als Endergebnis liefert.

Für den Wert  $v$  gibt es dabei folgende Möglichkeiten:

- Boolesche Konstanten: `true`, `false`
- Zahlkonstanten: `0`, `1`, ...
- Funktionswerte, d.h. Terme der Form: `fun x -> e`

# Operationelle Semantik: Regeln

(CONST)  $\frac{}{c \Downarrow c}$   $c$  ist Konstante, d.h. true, false oder eine Zahl

(FN)  $\frac{}{\text{fun } x \rightarrow e \Downarrow \text{fun } x \rightarrow e}$

(APP)  $\frac{e_1 \Downarrow \text{fun } x \rightarrow e'_1 \quad e_2 \Downarrow v_2 \quad e'_1[x \mapsto v_2] \Downarrow v}{e_1 e_2 \Downarrow v}$

(IF1)  $\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}$

(IF2)  $\frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}$

Diese Regeln erfassen eine Call-by-Value-Auswertungsstrategie.

# Operationelle Semantik: Regeln, Forts.

$$(\text{LET}) \frac{e_1 \Downarrow v_1 \quad e_2[x \mapsto v_1] \Downarrow v}{\text{let } x=e_1 \text{ in } e_2 \Downarrow v}$$

$$(\text{RECFUN}) \frac{}{\text{recfun } f \ x \rightarrow e \Downarrow \text{fun } x \rightarrow e[f \mapsto (\text{recfun } f \ x \rightarrow e)]}$$

$$(\text{OP}) \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \text{ op } e_2 \Downarrow v} \quad v = v_1 \text{ op } v_2$$

In (OP) bezeichnet  $v_1 \text{ op } v_2$  das entsprechende Ergebnis des Operators  $op$ . Z.B.  $3+4 = 7$ .

# Einfaches Typsystem

Die Menge der *Typen* wird durch folgende Grammatik erzeugt.

$$\tau, \tau_2, \tau_2 ::= \alpha \mid \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2$$

Hierbei läuft  $\alpha$  über Typvariablen. Wir benutzen die Buchstaben  $\alpha$ ,  $\beta$ ,  $\gamma$  für Typvariablen.

Ein *Typkontext*  $\Gamma$  ist eine endliche Abbildung, die endlich vielen Variablen je einen Typ zuweist, z.B.:  $[x \mapsto \text{int}, y \mapsto (\text{int} \rightarrow \text{bool})]$ .

Das *Typurteil*

$$\Gamma \vdash e : \tau$$

sagt aus: Wenn man für Variablen die von  $\Gamma$  zugewiesenen Typen annimmt, dann hat der Term  $e$  den Typ  $\tau$ .

# Einfaches Typsystem, Forts.

## Beispiele für (korrekte) Typurteile:

$$\vdash \text{fun } x \rightarrow x + 3 : \text{int} \rightarrow \text{int}$$

$$\vdash (\text{fun } x \rightarrow x + 3) 5 : \text{int}$$

$$\vdash \text{fun } x \rightarrow \text{if } x = 0 \text{ then true else false} : \text{int} \rightarrow \text{bool}$$

$$[x \mapsto (\text{int} \rightarrow \text{int})] \vdash (x 3) + (x 2) : \text{int}$$

## Notation:

- $\Gamma(x)$  ist der Typ, auf den  $\Gamma$  die Variable  $x$  abbildet.
- $\Gamma[x \mapsto \tau]$  ist die Abbildung, die  $x$  auf  $\tau$  abbildet und jede andere Variable  $y$  auf  $\Gamma(y)$ .

# Einfaches Typsystem, Forts.

$$(TT) \frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad (FF) \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$(CON) \frac{}{\Gamma \vdash n : \text{int}} \quad (VAR) \frac{}{\Gamma \vdash x : \tau} \Gamma(x) = \tau$$

$$(FN) \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

$$(APP) \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$(IF) \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

Beispiel:  $\vdash (\text{fun } x \rightarrow x) (\text{fun } y \rightarrow y) : \tau \rightarrow \tau$  ist herleitbar für alle  $\tau$ .

# Einfaches Typsystem, Forts.

$$(\text{LET}) \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

$$(\text{RECFUN}) \frac{\Gamma[f \mapsto (\tau_1 \rightarrow \tau_2)][x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \text{recfun } f \ x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

$$(\text{OP}) \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \text{ op } e_2 : \tau_3} (*)$$

(\*) In Regel (OP) müssen  $\tau_1, \tau_2$  und  $\tau_3$  der Operation  $op$  entsprechen:

| $op$     | +   | -   | =    | <    | &    | ... |
|----------|-----|-----|------|------|------|-----|
| $\tau_1$ | int | int | int  | int  | bool | ... |
| $\tau_2$ | int | int | int  | int  | bool | ... |
| $\tau_3$ | int | int | bool | bool | bool | ... |