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
  - Seiteneffekte
- Programmlogik

Induktive Definitionen

In der Programmiersprachentheorie werden viele Konzepte durch induktive Definitionen formalisiert.

Das übliche Format für induktive Definitionen sind *Inferenzregeln*.

Beispiele aus der Vorlesung:

$$\frac{e_1 \longrightarrow \mathsf{fun}_\pi\ x \texttt{->}\ e_1' \qquad e_2 \longrightarrow v_2 \qquad e_1'[x \mapsto v_2] \longrightarrow v}{e_1\ e_2 \longrightarrow v}$$

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

Urteile

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

Beispiele:

- n *even* — "Die natürliche Zahl n ist gerade."
- n *odd* — "Die natürliche Zahl n ist ungerade."
- *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 ϕ wahr macht, macht auch ψ 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:

$$(\textsc{name})\ \frac{U_1 \qquad U_2 \qquad \ldots \qquad U_n}{U}$$

- Die Urteile $U_1, \ldots, U_n$ 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:

$$\text{(EvenZ)}\ \frac{}{0\ \mathit{even}} \qquad \text{(OddS)}\ \frac{n\ \mathit{even}}{n+1\ \mathit{odd}} \qquad \text{(EvenS)}\ \frac{n\ \mathit{odd}}{n+1\ \mathit{even}}$$

Summen:

$$\text{(SumZ)}\ \frac{}{\mathit{sum}(n,0) = n} \qquad \text{(SumS)}\ \frac{\mathit{sum}(n,m) = r}{\mathit{sum}(n,m+1) = r+1}$$

Aussagenlogik:

$$\frac{}{\phi \vdash \phi} \qquad \frac{\phi \vdash \psi_1 \qquad \phi \vdash \psi_2}{\phi \vdash \psi_1 \wedge \psi_2} \qquad \frac{\phi \vdash \psi_1}{\phi \vdash \psi_1 \vee \psi_2} \qquad \frac{\phi \vdash \psi_2}{\phi \vdash \psi_1 \vee \psi_2}$$

$$\ldots$$

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:

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

steht für

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

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:

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

steht für alle Regeln, die man durch Einsetzung von konkreten aussagenlogischen Formeln für ϕ, ψ_1 und ψ_2 erhält, z.B.

$$\frac{\top \vdash \top \qquad \top \vdash \top}{\top \vdash \top \wedge \top}, \frac{(A \Rightarrow B) \vdash (C \wedge B) \qquad (A \Rightarrow B) \vdash \bot}{(A \Rightarrow B) \vdash (C \wedge B) \wedge \bot}, \ldots$$

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.

$$(\textsc{SumS})\ \frac{\mathit{sum}(n,m) = r}{\mathit{sum}(n,m') = r'}\ m' = m+1,\ r' = r+1$$

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)

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

  ist eine Herleitung für ihre Konklusion U.
- Gibt es eine Regel

  $$(\mathsf{S})\ \frac{U_1 \qquad U_2 \qquad \ldots \qquad U_n}{U}$$

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

  $$(\mathsf{S})\ \frac{\Pi_1 \qquad \Pi_2 \qquad \ldots \qquad \Pi_n}{U}$$

  eine Herleitung für U.
- Nichts weiter ist eine Herleitung.

Herleitungen: Beispiele

Beispiele:

$$\dfrac{\dfrac{}{A \vdash A} \qquad \dfrac{\dfrac{}{A \vdash A} \qquad \dfrac{}{A \vdash B \vee A}}{A \vdash A \wedge (B \vee A)}}{A \vdash A \wedge (A \wedge (B \vee A))}$$

$$\begin{array}{l}(\textsc{SumZ})\ \dfrac{}{sum(3,0) = 3} \\ (\textsc{SumS})\ \dfrac{sum(3,0) = 3}{sum(3,1) = 4} \\ (\textsc{SumS})\ \dfrac{sum(3,1) = 4}{sum(3,2) = 5}\end{array}$$

Kein Beispiel:

$$\begin{array}{l}(\textsc{SumS})\ \dfrac{sum(3,0) = 4}{sum(3,1) = 5} \\ (\textsc{SumS})\ \dfrac{sum(3,1) = 5}{sum(3,2) = 6}\end{array}$$

(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 $\mathsf{sum}(m, n) = r$ herleitbar ist, dann gilt $r = m + n$.

- Regel (SumZ)

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

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

- Regel (SumS)

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

  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 $\mathit{sum}(n, m) = r$ 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:

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

Programmstücke (statements):

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

Hierbei x läuft über Programmvariablen, n über Integerkonstanten, ℓ 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

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

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* (σ) ist eine endliche Abbildung von Programmvariablen auf ganze Zahlen.

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

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

Beispiel: $[\![x + 4 * y]\!]\sigma = 19$

Analog für Boolesche Ausdrücke.

Beispiel: $[\![x + 4 * y < 18]\!]\sigma = \mathit{false}$

Operationelle Semantik

Die operationelle Semantik ist durch zwei Urteile gegeben.

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

Regeln für die operationelle Semantik

$$(\textsc{ass})\ \frac{}{\langle [x := a]^\ell, \sigma\rangle \to \sigma[x \mapsto [\![a]\!]\sigma]}$$

$$(\textsc{skip})\ \frac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma}$$

$$(\textsc{seq1})\ \frac{\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \to \langle S_1'; S_2, \sigma'\rangle}$$

$$(\textsc{seq2})\ \frac{\langle S_1, \sigma\rangle \to \sigma'}{\langle S_1; S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle}$$

Beachte: (ASS) ist nur anwendbar, wenn $[\![a]\!]\sigma$ definiert ist.

Regeln für die operationelle Semantik, Forts.

$$(\textsc{IfT})\ \frac{}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \langle S_1, \sigma\rangle}\ [\![b]\!]\sigma = \mathit{true}$$

$$(\textsc{IfF})\ \frac{}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \langle S_2, \sigma\rangle}\ [\![b]\!]\sigma = \mathit{false}$$

$$(\textsc{WhileT})\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \to \langle S; \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle}\ [\![b]\!]\sigma = \mathit{true}$$

$$(\textsc{WhileF})\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \to \sigma}\ [\![b]\!]\sigma = \mathit{false}$$

Beispiel

Schreibe S als Abkürzung für folgendes Programm.

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

Dann kann man folgende Urteile herleiten:

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

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:

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

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 σ versucht man einen Herleitunsbaum für $\langle S, \sigma \rangle \to X$ von unten nach oben aufzubauen und damit X zu bestimmen.

Small-Step- und Big-Step-Semantik

Die Urteile $\langle S, \sigma \rangle \to \sigma'$ und $\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$ 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:

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

$$\text{(BIG2)}\ \frac{\langle S, \sigma \rangle \to \langle S', \sigma' \rangle \qquad \langle S', \sigma' \rangle \Downarrow \sigma''}{\langle S, \sigma \rangle \Downarrow \sigma''}$$

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

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

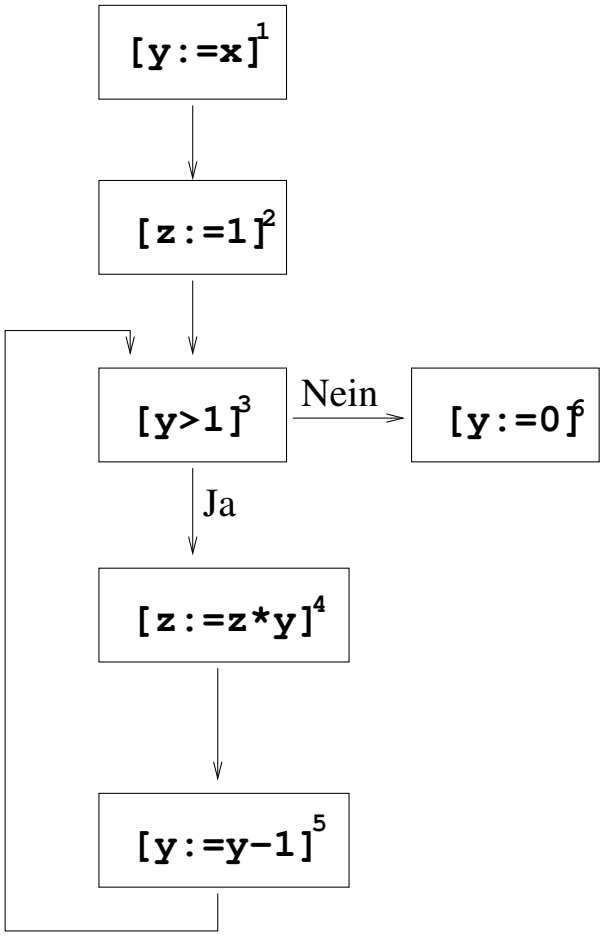
Typische Anwendungen:

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

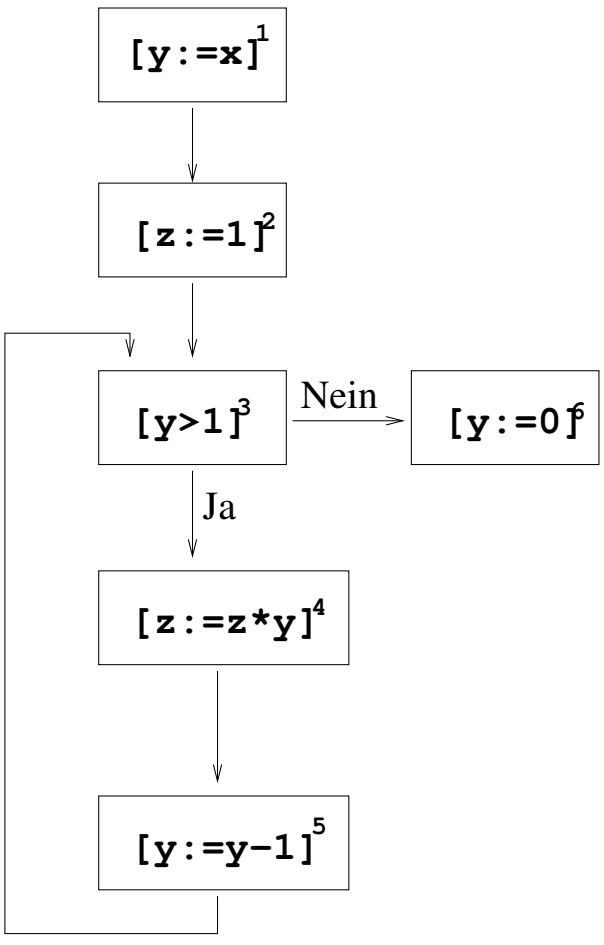
Kontrollflussgraph

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

$$
\begin{aligned}
&[\mathtt{y:=x}]^1;\\
&[\mathtt{z:=1}]^2;\\
&\mathtt{while}\ [\mathtt{1<y}]^3\ \mathtt{do}\ (\\
&\quad [\mathtt{z:=z*y}]^4;\\
&\quad [\mathtt{y:=y-1}]^5\\
&);\ [\mathtt{y:=0}]^6
\end{aligned}
$$



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 (
   [y:=x+1]^3;
   [z:=z+y]^4;
   [x:=2*y]^5
); [r:=z]^6
```

```
while [x<10]^2 do (
   [x:=x+1]^3;
   [z:=z+x]^4;
   [x:=2*x]^5
); [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 ℓ zwei Mengen bestimmen:

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

Beispiel:

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

ℓ	LV_{entry}	LV_{exit}
1	$\{\mathtt{x},\mathtt{z}\}$	$\{\mathtt{x},\mathtt{z}\}$
2	$\{\mathtt{x},\mathtt{z}\}$	$\{\mathtt{x},\mathtt{z}\}$
3	$\{\mathtt{x},\mathtt{z}\}$	$\{\mathtt{y},\mathtt{z}\}$
4	$\{\mathtt{y},\mathtt{z}\}$	$\{\mathtt{y},\mathtt{z}\}$
5	$\{\mathtt{y},\mathtt{z}\}$	$\{\mathtt{x},\mathtt{z}\}$
6	$\{\mathtt{z}\}$	$\varnothing$

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 ℓ liefert die Anweisung mit Label ℓ eine Gleichung für $LV_{entry}(\ell)$.

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

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

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

Gleichungssystem für das Beispielprogramm

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

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

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:

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

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

Berechnung der kleinsten Lösung

Benutzte eine Tabelle mit einem Eintrag für jede gesuchte Menge.

- Initalisiere alle Einträge mit $\varnothing$
- 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:

ℓ	$LV_{exit}(\ell)$	$LV_{entry}(\ell)$
6	$\varnothing$	$\varnothing$
5	$\varnothing$	$\varnothing$
4	$\varnothing$	$\varnothing$
3	$\varnothing$	$\varnothing$
2	$\varnothing$	$\varnothing$
1	$\varnothing$	$\varnothing$

$\rightsquigarrow$

ℓ	$LV_{exit}(\ell)$	$LV_{entry}(\ell)$
6	$\varnothing$	$\{z\}$
5	$\varnothing$	$\{y, z\}$
4	$\{y, z\}$	$\{y, z\}$
3	$\{y, z\}$	$\{x, z\}$
2	$\{x, z\}$	$\{x, z\}$
1	$\{x, z\}$	$\{x, z\}$

$\rightsquigarrow$

Berechnung der kleinsten Lösung

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

$\rightsquigarrow$

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

Ein $\rightsquigarrow$-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-Analysis sollte man möglichst rückwärts, also entgegen des Programmflusses vorgehen.

Live-Variablen Gleichungen

Allgemein können die Liveness-Datenflussgleichen für ein gegegebenes Programm nach folgendem Schema aufgestellt werden.

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

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

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

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 $\varnothing$ initialisieren, Gleichungen iterieren bis sich nichts mehr ändert

Reaching Definitions

Definition

Eine Zuweisung $[\mathtt{x:=a}]^\ell$ *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:

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

- 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 Zuweiung 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 (          x:=y*z;
  x:=y*z;             while b do (
  ...)                  ...)
```

Reaching Definitions

Für jeden Programmpunkt ℓ bestimmen wir zwei Mengen:

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

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

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

Reaching Definitions: Datenflussgleichungen

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

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

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

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

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

$$RD_{entry}(1) = \varnothing$$
$$RD_{entry}(2) = RD_{exit}(1)$$
$$RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$$
$$RD_{entry}(4) = RD_{exit}(3)$$
$$RD_{entry}(5) = RD_{exit}(4)$$
$$RD_{entry}(6) = RD_{exit}(3)$$

$$RD_{exit}(1) = (RD_{entry}(1) \setminus \{1,5,6\}) \cup \{1\}$$
$$RD_{exit}(2) = (RD_{entry}(2) \setminus \{2,4\}) \cup \{2\}$$
$$RD_{exit}(3) = RD_{entry}(3)$$
$$RD_{exit}(4) = (RD_{entry}(4) \setminus \{2,4\}) \cup \{4\}$$
$$RD_{exit}(5) = (RD_{entry}(5) \setminus \{1,5,6\}) \cup \{5\}$$
$$RD_{exit}(6) = (RD_{entry}(6) \setminus \{1,5,6\}) \cup \{6\}$$

Berechnung der kleinsten Lösung

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

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\varnothing$	$\varnothing$
2	$\varnothing$	$\varnothing$
3	$\varnothing$	$\varnothing$
4	$\varnothing$	$\varnothing$
5	$\varnothing$	$\varnothing$
6	$\varnothing$	$\varnothing$

$\rightsquigarrow$

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\varnothing$	$\{1\}$
2	$\{1\}$	$\{1,2\}$
3	$\{1,2\}$	$\{1,2\}$
4	$\{1,2\}$	$\{1,4\}$
5	$\{1,4\}$	$\{4,5\}$
6	$\{1,2,4,5\}$	$\{2,4,6\}$

$\rightsquigarrow$

Berechnung der kleinsten Lösung, Forts.

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

$\rightsquigarrow$

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

$\rightsquigarrow$

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\emptyset$	$\{1\}$
2	$\{1\}$	$\{1,2\}$
3	$\{1,2,4,5\}$	$\{1,2,4,5\}$
4	$\{1,2,4,5\}$	$\{1,4,5\}$
5	$\{1,4,5\}$	$\{4,5\}$
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.

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

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

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

Available Expressions

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

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

Für jeden Programmpunkt ℓ berechnen wir zwei Mengen:

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

Dieses Mal entsprechen größere Mengen genauerer Information.

Datenflussgleichungen

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

wobei

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

(Ein Teilausdruck entspricht einem Teilbaum des Syntaxbaums.)

Datenflussgleichungen im Beispiel

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

ℓ	$kill_{AE}$	gen_{AE}
1	$\varnothing$	{a+b}
2	$\varnothing$	{a*b}
3	$\varnothing$	{a+b}
4	{a+b, a*b, a+1}	$\varnothing$
5	$\varnothing$	{a+b}

$$
\begin{array}{lcl}
AE_{entry}(1) & = & \varnothing \\
AE_{entry}(2) & = & AE_{exit}(1) \\
AE_{entry}(3) & = & AE_{exit}(2) \cap AE_{exit}(5) \\
AE_{entry}(4) & = & AE_{exit}(3) \\
AE_{entry}(5) & = & AE_{exit}(4)
\end{array}
$$

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

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

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 $\varnothing$.

Man erhält:

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

$\rightsquigarrow \cdots \rightsquigarrow$

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

Größte vs. kleinste Lösung

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

Anders bei diesem Beispiel:

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

Hier gelten die Gleichungen

$$\begin{array}{|l|}\hline AE_{entry}(1) = \emptyset \\ AE_{entry}(2) = AE_{exit}(1) \cap AE_{exit}(3) \\ AE_{entry}(3) = AE_{exit}(2) \\ \hline \end{array}$$

$$\begin{array}{|l|}\hline AE_{exit}(1) = (AE_{entry}(1) \setminus \{\mathsf{x-1}\}) \cup \{\mathsf{a+b}\} \\ AE_{exit}(2) = AE_{entry}(2) \\ AE_{exit}(3) = AE_{entry}(3) \\ \hline \end{array}$$

Die größte Lösung liefert $AE_{exit}(3) = \{\mathsf{a+b}\}$ 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?

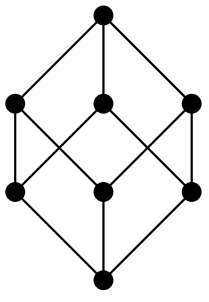
Halbordnung

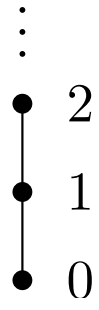
Definition

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-Diagram 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:

$$\bot := \bigsqcap L \qquad \top := \bigsqcup L$$

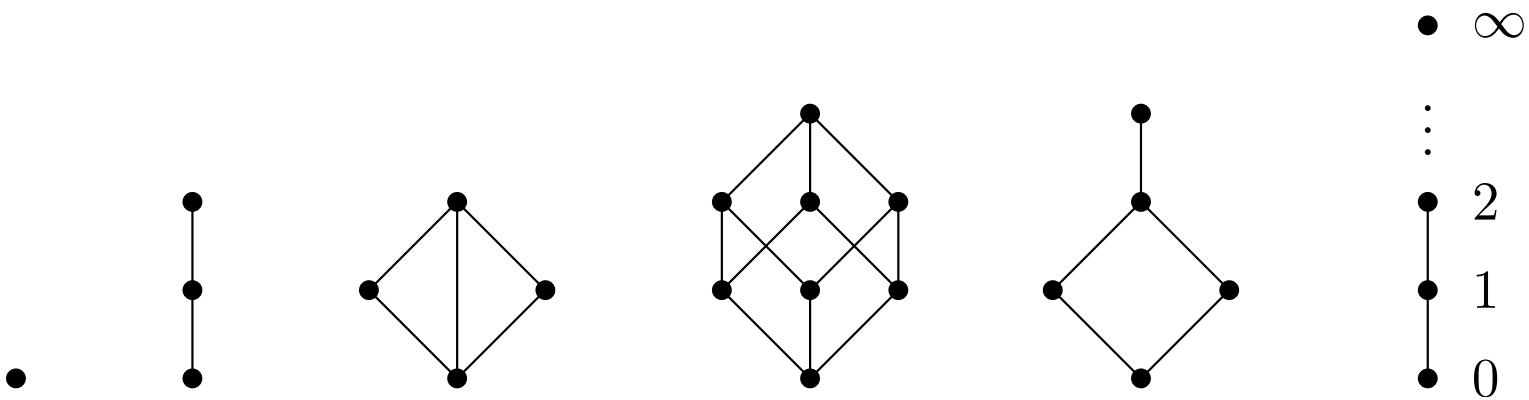
$$x \wedge y := \bigsqcap \{x, y\} \qquad x \vee y := \bigsqcup \{x, y\}$$

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

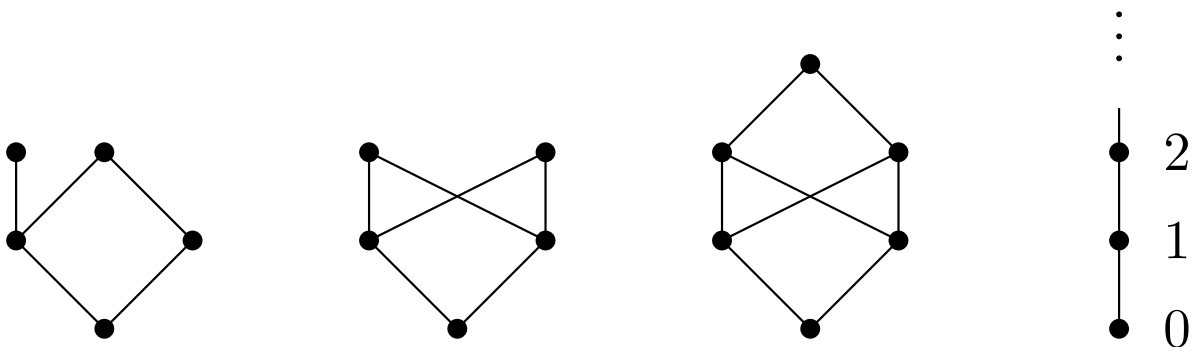
Die *Höhe* eines Verbandes ist die Länge des längsten Pfades von $\bot$ nach $\top$ im Hasse-Diagramm.

Beispiele

Vollständige Verbände:



Keine vollständigen Verbände:

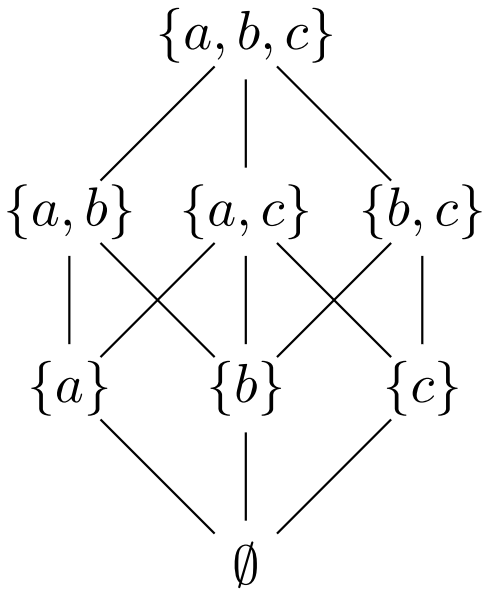


Beispiele

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

In diesem Fall gilt $\bot = \varnothing, \top = U, \bigsqcup X = \bigcup X, \bigsqcap X = \bigcap X$.

Beispiel für $U = \{1, 2, 3\}$:



Monotone Funktionen

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

Definition (Monotonie)

Eine Funktion $f : L \to L$ 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.
- $f(X) = X \backslash \{1\}$ ist monoton.
- Sind f und g monoton, so ist auch $h(X) = g(f(X))$ monoton.
- $f(X) = \mathbb{N} \backslash X$ 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 $f(X) = X \backslash \{1\}$.
- $f(X) = \mathbb{N} \backslash X$ hat keinen Fixpunkt.

Fixpunkte

Theorem (Kleene)

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

$$\mathit{lfp}(f) := \bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \} \quad .$$

Wegen der Monotonie von f gilt:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$$

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

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

Fixpunkte

Theorem (Kleene)

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

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

Wegen der Monotonie von f gilt:

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

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 $f : L \to L$ monoton. Dann bildet die Menge dieser Fixpunkte selbst einen vollständigen Verband.

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

Anwendung: Datenflussanalyse

Die Gleichunssysteme für die verschiedenen Datenflussanalysen haben alle die Form

$$\begin{aligned} X_1 &= F_1(X_1, \ldots, X_n) \\ X_2 &= F_2(X_1, \ldots, X_n) \\ &\ldots \\ X_n &= F_n(X_1, \ldots, X_n) \end{aligned}$$

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:** $U = (\mathcal{P}(\mathit{AExp}), \subseteq)$.

Anwendung: Datenflussanalyse

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

Angenommen $V_1, \ldots, V_n$ sind vollständige Verbände.
Dann ist auch die Menge

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

ein vollständiger Verband, wenn man die Ordnung punktweise definiert:

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

Es gilt $\bot = (\bot, \ldots, \bot)$.

Anwendung: Datenflussanalyse

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

Durch

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

wird eine Funktion $F\colon U \times \cdots \times U \to U \times \cdots \times U$ 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 $\mathit{lfp}(F)$ (bzw. $\mathit{gfp}(F)$).

Fixpunktiteration

Berechnung von *lfp*(F):

$x := (\bot, \ldots, \bot);$
repeat
  old $:= x;$
  $x := F(x);$
until x=old

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

Fixpunktiteration

Die Algorithmen zur Datenflussanalyse entsprachen folgendem etwas besseren Algorithmus.

$x_1 := \bot; \ldots; x_n := \bot;$
repeat
  $\texttt{old}_1 := x_1; \ldots; \texttt{old}_n := x_n;$
  $x_1 := F_1(x_1, \ldots, x_n); \ldots; x_n := F_n(x_1, \ldots, x_n)$
until $x_1 = \texttt{old}_1 \wedge \cdots \wedge x_n = \texttt{old}_n$

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 $(S, \to)$.
Definiere:

$$\mathsf{EX}(B) := \{s \in S \mid \exists s' \in B.\ s \to s'\}$$
$$\mathsf{AX}(B) := \{s \in S \mid \forall s' \in S.\ s \to s' \implies s' \in B\}$$

Im Potenzmengenverband auf S haben wir:

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

$$\varnothing \subseteq B \cup \mathsf{EX}(\varnothing) \subseteq B \cup \mathsf{EX}(B \cup \mathsf{EX}(\varnothing)) \subseteq \ldots$$

Anwendung: CTL, Forts.

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

$$S \supseteq B \cap \mathsf{AX}(S) \supseteq B \cap \mathsf{AX}(B \cap \mathsf{AX}(S)) \supseteq \ldots$$

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

$$\vdash S : D_1 \to D_2$$

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

Die gemeinte Bedeutung von $\vdash S : D_1 \to D_2$ 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."*

Programmanalyse und Typsysteme

Typregeln für das Typurteil

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

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

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

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

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

$$\text{(SUB)}\ \frac{\vdash S : D_1 \to D_2}{\vdash S : D_1' \to D_2'}\ D_1' \subseteq D_1, D_2 \subseteq D_2'$$

Programmanalyse und Typsysteme

Beispiel

Wir geben eine Typherleitung für unser Beispielprogramm an:

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

Schreibe kurz D für $\{1, 5, 2, 4\}$.

Erster Teil der Herleitung:

$$\dfrac{\dfrac{\dfrac{}{\vdash [\texttt{z:=z*y}]^4 : D \to D\backslash\{2\}}(\textsc{Ass}) \quad \dfrac{}{\vdash [\texttt{y:=y-1}]^5 : D\backslash\{2\} \to D\backslash\{2,1\}}(\textsc{Ass})}{\vdash [\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5 : D \to D\backslash\{2,1\}}(\textsc{Seq})}{\vdash [\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5 : D \to D}(\textsc{Sub})}{\vdash \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5) : D \to D}(\textsc{While})$$

Beispielherleitung

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

$$[\mathsf{y}:=\mathsf{x}]^1; [\mathsf{z}:=1]^2; S; [\mathsf{y}:=\mathsf{0}]^6$$

herleiten.

$$\dfrac{\dfrac{}{\vdash [\mathsf{y}:=\mathsf{x}]^1 : \varnothing \to \{1\}} \qquad \dfrac{\dfrac{\dfrac{}{\vdash [\mathsf{z}:=1]^2 : \{1\} \to \{1,2\}}}{\vdash [\mathsf{z}:=1]^2 : \{1\} \to D} \qquad \dfrac{\vdots}{\vdash S : D \to D} \qquad \dfrac{}{\vdash [\mathsf{y}:=\mathsf{0}]^6 : D \to D \cup \{6\}}}{\vdash S; [\mathsf{y}:=\mathsf{0}]^6 : D \to D \cup \{6\}}}{\vdash [\mathsf{z}:=1]^2; S; [\mathsf{y}:=\mathsf{0}]^6 : \{1\} \to D \cup \{6\}}$$

$$\vdash [\mathsf{y}:=\mathsf{x}]^1; [\mathsf{z}:=1]^2; S; [\mathsf{y}:=\mathsf{0}]^6 : \varnothing \to D \cup \{6\}$$

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 $\vdash S : D_1 \to D_2$ herleitbar ist, heißt *Typprüfung* (type checking).

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

Man kann direkt einen Algorithmus für Typinferenz konstruieren.

Programmanalyse und Typsysteme

Ansatz zur Typinferenz

Beispiel: Typinferenz für $\texttt{while}\ [\texttt{y>1}]^3\ \texttt{do}\ ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5)$.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash [\texttt{z:=z*y}]^4 : D_7 \to D_7\backslash\{2\} \cup \{4\}}(\textsc{ass})}{\vdash [\texttt{z:=z*y}]^4 : D_4 \to D_6}(\textsc{sub}) \quad \dfrac{\dfrac{}{\vdash [\texttt{y:=y-1}]^5 : D_8 \to D_8\backslash\{1,6\} \cup \{5\}}(\textsc{ass})}{\vdash [\texttt{y:=y-1}]^5 : D_6 \to D_5}(\textsc{sub})}{\vdash [\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5 : D_4 \to D_5}(\textsc{seq})}{\vdash [\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5 : D_3 \to D_3}(\textsc{sub})}{\vdash \texttt{while}\ [\texttt{y>1}]^3\ \texttt{do}\ ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5) : D_3 \to D_3}(\textsc{while})}{\vdash \texttt{while}\ [\texttt{y>1}]^3\ \texttt{do}\ ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5) : D_1 \to D_2}(\textsc{sub})$$

(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.)

Programmanalyse und Typsysteme

Ansatz zur Typinferenz

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

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

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

$$D_4 \subseteq D_7 \qquad (D_7 \backslash \{2\}) \cup \{4\} \subseteq D_6$$
$$D_6 \subseteq D_8 \qquad (D_9 \backslash \{1,6\}) \cup \{5\} \subseteq D_5$$
$$D_3 \subseteq D_4 \qquad D_5 \subseteq D_3$$
$$D_1 \subseteq D_3 \qquad D_3 \subseteq D_2$$

Gesucht ist eine möglichst kleine Lösung dieser Mengeninklusionen.

Ansatz zur Typinferenz

Nach dem Satz von Knaster-Tarski ist die kleineste 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 $F(D_1, \ldots, D_8)$ als

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

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 σ 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:

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

  wobei D' für $\sigma.D \backslash \{\ell' \mid \ell' \text{ ist Zuweisung and } x\} \cup \{\ell\}$ steht.
- Alle anderen Regeln bleiben unverändert.

Korrektheit des Typsystems für *RD*

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

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

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

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

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 Typinferenz
- Erweiterungen um Exceptions, Referenzen

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

- Kontrollflussanalyse
- Exceptions
- Speicherzugriffe

Funktionale Sprache

Terme:

$$e, e_1, e_2 ::= \texttt{x} \mid n \mid \texttt{true} \mid \texttt{false} \mid e_1\ \mathit{op}\ e_2$$
$$\mid \texttt{fun x -> } e \mid e_1\ e_2 \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2$$
$$\mid \texttt{let x = } e_1 \texttt{ in } e_2 \mid \texttt{recfun f x -> } e$$
$$\mathit{op} ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{=} \mid \texttt{\&} \mid \texttt{|} \mid \ldots$$

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

$$\texttt{fun x -> } e \qquad \texttt{let x = } e_1 \texttt{ in } e \qquad \texttt{recfun f x -> } e$$

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 $e[\mathsf{x} \mapsto e']$ 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:

- $(\texttt{fun x -> y x})[\mathsf{y} \mapsto (\texttt{fun z -> z} * 2)] = (\texttt{fun x -> } (\texttt{fun z -> z} * 2)\ \mathsf{x})$
- $(\texttt{fun x -> y x})[\mathsf{y} \mapsto (\texttt{fun z -> x} * 2)] = (\texttt{fun u -> } (\texttt{fun z -> x} * 2)\ \mathsf{u})$

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: $0, 1, \ldots$
- Funktionswerte, d.h. Terme der Form: `fun` x `->` e

Operationelle Semantik: Regeln

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

$$\text{(FN)}\ \frac{}{\texttt{fun}\ x\ \texttt{->}\ e \Downarrow \texttt{fun}\ x\ \texttt{->}\ e}$$

$$\text{(APP)}\ \frac{e_1 \Downarrow \texttt{fun}\ x\ \texttt{->}\ e_1' \qquad e_2 \Downarrow v_2 \qquad e_1'[x \mapsto v_2] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

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

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

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

Operationelle Semantik: Regeln, Forts.

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

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

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

In (OP) bezeichnet v_1 **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.

$$\tau, \tau_1, \tau_2 ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2$$

Hierbei läuft α über Typvariablen. Wir benutzen die Buchstaben α, β, γ für Typvariablen.

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

Das *Typurteil*

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

sagt aus: Wenn man für Variablen die von Γ zugewiesenen Typen annimmt, dann hat der Term e den Typ τ.

Einfaches Typsystem, Forts.

Beispiele für (korrekte) Typurteile:

$$\vdash \texttt{fun x -> x} + 3 : \texttt{int} \to \texttt{int}$$

$$\vdash (\texttt{fun x -> x} + 3)\, 5 : \texttt{int}$$

$$\vdash \texttt{fun x -> if } x = 0 \texttt{ then true else false} : \texttt{int} \to \texttt{bool}$$

$$[\texttt{x} \mapsto (\texttt{int} \to \texttt{int})] \vdash (\texttt{x}\, 3) + (\texttt{x}\, 2) : \texttt{int}$$

Notation:

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

Einfaches Typsystem, Forts.

$$(\textsf{TT})\ \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad (\textsf{FF})\ \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$(\textsf{CON})\ \frac{}{\Gamma \vdash n : \texttt{int}} \qquad (\textsf{VAR})\ \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau$$

$$(\textsf{FN})\ \frac{\Gamma[\mathsf{x} \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \textsf{fun x -> } e : \tau_1 \to \tau_2}$$

$$(\textsf{APP})\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(\textsf{IF})\ \frac{\Gamma \vdash e : \textsf{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \textsf{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Beispiel: $\vdash (\texttt{fun x -> x})\ (\texttt{fun y -> y}) : \tau \to \tau$ ist herleitbar für alle τ.

Einfaches Typsystem, Forts.

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

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

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

(*) In Regel (OP) müssen τ_1, τ_2 und τ_3 der Operation *op* entsprechen:

op	+	-	=	<	&	...
τ_1	int	int	int	int	bool	...
τ_2	int	int	int	int	bool	...
τ_3	int	int	bool	bool	bool	...

Einfaches Typsystem, Forts.

Satz (Subject Reduction):

Wenn $\vdash e : \tau$ und $e \Downarrow v$ herleitbar sind, dann auch $\vdash v : \tau$.

Ein Term vom Typ $\tau_1 \to \tau_2$ kann also nur zu einem Wert der Form `fun x ->` e auswerten, nicht zum Beispiel zu `true`.

Man kann weiterhin zeigen: Wenn $\vdash e : \tau$ herleitbar ist, dann gilt entweder $e \Downarrow v$, oder e repräsentiert eine nichtterminierende Berechnung.

Typkorrekte Programme können also keine “Laufzeitfehler” haben.

Typinferenz

Gegeben seien Term e und Typkontext Γ.
Welchen Typ e unter den Annahmen in Γ haben kann?

Für eine Typisierung kann unter Umständen eine Einschränkung der Typvariablen nötig sein.
Beispiel: Das Typinferenzproblem $[f \mapsto (\alpha \to \texttt{bool})] \vdash f\ \texttt{3} : ?$ ist lösbar mit Antwort `bool`, wenn man die Typvariable α auf Typ `int` einschränkt.

Typinferenzproblem: Gegeben sind Term e und Typkontext Γ.
Gesucht ist ein Typ τ und eine Typsubstitution θ, so dass

$$\Gamma[\theta] \vdash e : \tau$$

herleitbar ist.

Substitutionen

Eine *Typsubstitution* θ ist eine Abbildung, die endlich viele Typvariablen auf Typen abbildet.
Notation: $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$

Anwendung einer Substitution:

Schreibe $\tau[\theta]$ für die Ersetzung de Typvariablen in τ entsprechend θ.

$$\begin{aligned}
\texttt{int}[\theta] &= \texttt{int} \\
\texttt{bool}[\theta] &= \texttt{bool} \\
(\tau_1 \to \tau_2)[\theta] &= \tau_1[\theta] \to \tau_2[\theta] \\
\alpha[\theta] &= \begin{cases} \tau & \text{wenn } \theta \text{ die Variable } \alpha \text{ auf } \tau \text{ abbildet,} \\ \alpha & \text{wenn } \theta \text{ für die Variable } \alpha \text{ nicht definiert ist.} \end{cases}
\end{aligned}$$

Analog für Typkontexte: Wenn $\Gamma = [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n]$, dann $\Gamma[\theta] = [x_1 \mapsto \tau_1[\theta], \ldots, x_n \mapsto \tau_n[\theta]]$.

Substitutionen, Forts.

Komposition von Substitutionen

Seien θ_1 und θ_2 zwei Substitutionen. Dann gibt es eine Substitution, die wir mit $\theta_1;\theta_2$ bezeichnen, so dass $\tau[\theta_1;\theta_2] = \tau[\theta_1][\theta_2]$ für alle Typen τ gilt.

Beispiel:

$$\theta_1 = [\alpha \mapsto (\beta \to \texttt{bool})] \qquad \theta_2 = [\beta \mapsto \texttt{int}]$$

Dann ist $\theta_1;\theta_2$ die Substitution $[\alpha \mapsto (\texttt{int} \to \texttt{bool}), \beta \mapsto \texttt{int}]$.

Allgemein:

Die Abbildung $\theta_1;\theta_2$ ist für jede Typvariable definiert, für die θ_1 oder θ_2 definiert ist. Jede solche Typvariable α wird von $\theta_1;\theta_2$ auf $\alpha[\theta_1][\theta_2]$ abgebildet.

Spezifikation von $\mathcal{W}$

Der *Typinferenzalgorithmus $\mathcal{W}$* [Damas, Milner 1982] ist wie folgt spezifiziert.

$\mathcal{W}$ ist eine Funktion, die als Eingabe einen Typkontext Γ und einen Term e nimmt. Ausgabe ist entweder "fail" oder ein Paar (θ, τ) bestehend aus einer Typsubstitution θ und einem Typ τ.

Falls $\mathcal{W}(\Gamma, e) = (\theta, \tau)$, so gilt:

- $\Gamma[\theta] \vdash e : \tau$ ist herleitbar.
  „Das Ergebnis ist eine Lösung des Typinferenzproblems."
- Ist $\Gamma[\theta'] \vdash e : \tau'$ herleitbar, so gibt es eine Substitution θ'' mit $\Gamma[\theta'] = \Gamma[\theta][\theta'']$ und $\tau' = \tau[\theta'']$.
  „Alle anderen Lösungen sind Spezialfälle von $\Gamma[\theta] \vdash e : \tau$."

($\mathcal{W}$ liefert also die allgemeinste Lösung.)

Falls $\mathcal{W}(\Gamma, e) =$ "fail", dann ist kein Typurteil der Form $\Gamma[\theta] \vdash e : \tau$ herleitbar.

Beispiele

- $\mathcal{W}(\varnothing, \texttt{fun x -> x}) = (\varnothing, \alpha \to \alpha)$
- $\mathcal{W}([\texttt{x} \mapsto \texttt{bool}], \texttt{x} - 1) =$ “fail”
- $\mathcal{W}([x \mapsto \tau], x) = (\varnothing, \tau)$
- $\mathcal{W}([x \mapsto \alpha], x - 1) = ([\alpha \mapsto \texttt{int}], \texttt{int})$
- $\mathcal{W}([x \mapsto \alpha, y \mapsto \beta], x\ y) = ([\alpha \mapsto (\beta \to \gamma)], \gamma)$
- $\mathcal{W}([x \mapsto \alpha, y \mapsto \alpha], x\ y) =$ “fail”

Algorithmus $\mathcal{W}$

Idee:

- Beginne mit $\Gamma \vdash e : \ldots$ und versuche rückwärts eine Herleitung aufzubauen. Kommt dabei ein unbekannter Typ vor (z.B. τ_1 in (FUN)), so verwende eine neue Typvariable.
- Einige Typregeln verlangen Gleichheit von Typen in verschiedenen Prämissen (z.B. τ_1 in (APP)). Löse solche Gleichungsbedingungen allgemeinstmöglich.
- Für die allgemeinstmögliche Lösung von Gleichungen zwischen Typen wird Unifikation verwendet. Das Ergebnis ist eine Substitution.

Man kann wie auf Folien 287–289 erst alle Gleichungen erzeugen und diese dann lösen. Oft werden die Gleichungen gleich bei der Konstruktion der Herleitung gelöst.

Spezifikation der Unifikation $\mathcal{U}$

Die Unifikationsfunktion $\mathcal{U}$ nimmt als Eingabe zwei Typen τ_1 und τ_2 und gibt als Ausgabe eine Typsubstitution oder “fail“.

Wenn $\mathcal{U}(\tau_1, \tau_2) = \theta$, so muss gelten:

- $\tau_1[\theta] = \tau_2[\theta]$.
- Für jede Substitution θ' mit $\tau_1[\theta'] = \tau_2[\theta']$ existiert θ'' mit $\theta' = \theta; \theta''$. Jede Substitution θ', die die gegebenen Typen gleich macht, ist also ein Spezialfall von θ.

Wenn $\mathcal{U}(\tau_1, \tau_2) =$ “fail”, dann gibt es keine Substitution θ mit $\tau_1[\theta] = \tau_2[\theta]$.

Unifikation: Beispiele

Nach Spezifikation muss gelten:

- $\mathcal{U}(\alpha, \tau) = [\alpha \mapsto \tau]$ wenn α nicht in τ vorkommt.
- $\mathcal{U}(\texttt{int}, \texttt{bool}) =$ “fail”
- $\mathcal{U}(\alpha \to \beta, (\texttt{bool} \to \beta) \to (\texttt{int} \to \gamma)) =$
  $[\alpha \mapsto (\texttt{bool} \to (\texttt{int} \to \gamma)), \beta \mapsto (\texttt{int} \to \gamma)]$

Erinnerung: $\to$ ist rechtsassoziativ, d.h. $\tau_1 \to \tau_2 \to \tau_3$ steht für $\tau_1 \to (\tau_2 \to \tau_3)$.

Ein Algorithmus für $\mathcal{U}$ kann durch Rekursion über die Typsyntax definiert werden (siehe Folie 318).

Definition von $\mathcal{W}$

Die Berechnung von $\mathcal{W}(\Gamma, e)$ erfolgt durch Fallunterscheidung über e. Wenn irgendeine Teilberechnung "fail" liefert, so wird abgebrochen und das Gesamtergebnis ist ebenfalls "fail".

- $\mathcal{W}(\Gamma, \texttt{true}) = (\varnothing, \texttt{bool})$
- $\mathcal{W}(\Gamma, \texttt{false}) = (\varnothing, \texttt{bool})$
- $\mathcal{W}(\Gamma, n) = (\varnothing, \texttt{int})$
- $\mathcal{W}(\Gamma, x) = (\varnothing, \Gamma(x))$
- $\mathcal{W}(\Gamma, e_1 + e_2)$ wird wie folgt berechnet:
  - Berechne rekursiv $(\theta_1, \tau_1) := \mathcal{W}(\Gamma, e_1)$.
    Wenn τ nicht `int` ist, dann "fail".
  - Berechne rekursiv $(\theta_2, \tau_2) := \mathcal{W}(\Gamma[\theta_1], e_2)$.
    Wenn τ nicht `int` ist, dann "fail".
  - Gib $(\theta_1; \theta_2, \texttt{int})$ als Ergebnis zurück.

Definition von $\mathcal{W}$, Forts.

- $\mathcal{W}(\Gamma, \texttt{fun x -> } e)$ wird wie folgt berechnet:
  - Erzeuge neue Typvariable α.
  - Berechne rekursiv $(\theta, \tau) := \mathcal{W}(\Gamma[x \mapsto \alpha], e)$.
  - Gib $(\theta, \alpha[\theta] \to \tau)$ als Ergebnis zurück.
- $\mathcal{W}(\Gamma, e_1\ e_2)$ wird wie folgt berechnet:
  - Berechne rekursiv $(\theta_1, \tau_1) := \mathcal{W}(\Gamma, e_1)$.
  - Berechne rekursiv $(\theta_2, \tau_2) := \mathcal{W}(\Gamma[\theta_1], e_2)$.
  - Erzeuge neue Typvariable β und berechne $\theta_3 := \mathcal{U}(\tau_1[\theta_2], (\tau_2 \to \beta))$.
  - Gib $(\theta_1; \theta_2; \theta_3, \beta[\theta_3])$ als Ergebnis zurück.
- $\mathcal{W}(\Gamma, \texttt{recfun f x -> } e)$ wird wie folgt berechnet:
  - Erzeuge neue Typvariablen α, β.
  - Berechne rekursiv $(\theta, \tau) := \mathcal{W}(\Gamma[f \mapsto (\alpha \to \beta)][x \mapsto \alpha], e)$.
  - Berechne $\theta_1 := \mathcal{U}(\beta[\theta], \tau)$.
  - Gib $(\theta; \theta_1, \alpha[\theta][\theta_1] \to \tau[\theta_1])$ als Ergebnis zurück.

Definition von $\mathcal{W}$, Forts.

- $\mathcal{W}(\Gamma, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$ wird wie folgt berechnet:
  - Berechne rekursiv $(\theta, \tau) := \mathcal{W}(\Gamma, e)$. Wenn τ nicht `bool` ist, dann "fail".
  - Berechne rekursiv $(\theta_1, \tau_1) := \mathcal{W}(\Gamma[\theta], e_1)$.
  - Berechne rekursiv $(\theta_2, \tau_2) := \mathcal{W}(\Gamma[\theta][\theta_1], e_2)$.
  - Berechne $\theta_3 := \mathcal{U}(\tau_1[\theta_2], \tau_2)$.
  - Gib $(\theta; \theta_1; \theta_2; \theta_3, \tau_2[\theta_3])$ als Ergebnis zurück.
- $\mathcal{W}(\Gamma, \texttt{let x} = e_1\ \texttt{in}\ e_2)$ wird wie folgt berechnet:
  - Berechne rekursiv $(\theta_1, \tau_1) := \mathcal{W}(\Gamma, e_1)$.
  - Berechne rekursiv $(\theta_2, \tau_2) := \mathcal{W}(\Gamma[\theta_1][\texttt{x} \mapsto \tau_1], e_2)$.
  - Gib $(\theta_1; \theta_2, \tau_2)$ als Ergebnis zurück.

Definition von $\mathcal{U}$

Die Unifikationsfunktion $\mathcal{U}$ kann rekursiv definiert werden.

$$\begin{aligned}
\mathcal{U}(\texttt{int}, \texttt{int}) &= \varnothing \\
\mathcal{U}(\texttt{bool}, \texttt{bool}) &= \varnothing \\
\mathcal{U}(\alpha, \alpha) &= \varnothing \\
\mathcal{U}(\alpha, \tau) &= [\alpha \mapsto \tau] \text{ wenn } \alpha \text{ nicht freie Variable in } \tau \text{ ist} \\
\mathcal{U}(\tau, \alpha) &= [\alpha \mapsto \tau] \text{ wenn } \alpha \text{ nicht freie Variable in } \tau \text{ ist} \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') &= \theta_1; \theta_2 \text{ wenn } \mathcal{U}(\tau_1, \tau_1') = \theta_1 \neq \text{“fail” und} \\
&\qquad \mathcal{U}(\tau_2[\theta_1], \tau_2'[\theta_1]) = \theta_2 \neq \text{“fail”} \\
\mathcal{U}(\tau_1, \tau_2) &= \text{“fail” sonst}
\end{aligned}$$

Polymorphie

Das einfache Typsystem ist für praktische Anwendungen zu eingeschränkt.

Zum Beispiel ist der Term

$$\texttt{let i} = \texttt{fun}\ x \texttt{ -> } x\ \texttt{in}$$
$$\texttt{if i true then i } 1\ \texttt{else i } 2$$

nicht typisierbar, obwohl $\texttt{fun}\ x \texttt{ -> } x$ den Typ $\alpha \to \alpha$ hat.

Man müsste schreiben:

$$\texttt{let i\_int} = \texttt{fun}\ x \texttt{ -> } x\ \texttt{in}$$
$$\texttt{let i\_bool} = \texttt{fun}\ x \texttt{ -> } x\ \texttt{in}$$
$$\texttt{if i\_bool true then i\_int } 1\ \texttt{else i\_int } 2$$

Typvariablen sind nur Platzhalter für konkrete Typen.

Typschemata

Wir erweitern das Typsystem selbst um Quantifikation, so dass z.B.:
`fun x -> x` : $\forall\alpha.\, \alpha \to \alpha$.

Grammatik für Typen (τ) und Typschemata (σ)

$$\tau_1, \tau_2 ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2$$
$$\sigma ::= \forall(\alpha_1, \ldots, \alpha_n).\tau$$

Das Typschema $\forall(\alpha_1, \ldots, \alpha_n).\tau$ bindet die Variablen $\alpha_1, \ldots, \alpha_n$ in τ.

- α kommt frei vor in $(\alpha \to \tau)$.
- α kommt nicht frei (sondern nur gebunden) vor im Typschema $\forall(\alpha).\, \alpha \to \alpha$. Man könnte ja stattdessen auch $\forall(\beta).\, \beta \to \beta$

Abkürzungen und Beispiele

Abkürzungen:

- Ein Typschema der Form $\forall(\alpha).\tau$ (also $n = 1$) schreiben wir kurz $\forall\alpha.\tau$.
- Ein Typschema der Form $\forall().\tau$ (also $n = 0$) schreiben wir kurz τ. Auf diese Weise werden die *Typen zu einer Teilmenge der Typschemata*.
- Jeder Typ ist ein Typschema aber nicht umgekehrt.

Beispiele:

- $\forall(\alpha, \beta, \gamma).\, (\alpha \to \beta \to \gamma) \to (\beta \to \alpha \to \gamma)$.
- In der Praxis nimmt man auch rekursive Typen, wie z.B. Listen hinzu: $\forall(\alpha, \beta).\, (\alpha \to \beta) \to \texttt{list}(\alpha) \to \texttt{list}(\beta)$.

Unterschied zwischen “let” und Applikation

Man kann schreiben:

$$\texttt{let f = fun x-> x in if f true then f } 0 \texttt{ else f } 1$$

Man kann *nicht* schreiben:

$$(\texttt{fun f -> if f true then } f\ 0 \texttt{ else f } 1)\ (\texttt{fun x-> x})$$

denn Funktionsparameter (hier f) können nicht ein Typschema als “Typ” haben. Let-gebundene Variablen hingegen schon.

System F

Es gibt Typsysteme, die keinen Unterschied zwischen Typen und Typschemata machen und Typen wie $(\forall\alpha.\alpha \to \alpha) \to \texttt{int}$ erlauben. Z.B. System F. Dann wird aber Typinferenz unmöglich (unentscheidbar) und man muss Typannotate zumindest an bestimmten Schlüsselstellen angeben.

Polymorphes Typsystem

Typkontexte bilden nunmehr Variablen auf Typschemata ab.
Das Typurteil hat das Format

$$\Gamma \vdash e : \sigma \quad .$$

Die Typregeln (TT), (FF), (CON), (VAR), (FN), (APP), (RECFUN), (OP), (IF) werden unverändert übernommen mit der gemachten Übereinkunft, dass Typen spezielle Typschemata sind. So ist zum Beispiel die Regel

$$(\text{FN})\ \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathsf{fun}\ x \texttt{->} e : \tau_1 \to \tau_2}$$

nur anwendbar, wenn τ_1 ein Typ und kein echtes Typschema ist.

Es kommen neue Regeln (GEN) und (INST) hinzu; die Regel für (LET) wird verallgemeinert.

Polymorphes Typsystem, Generalisierungsregel

$$(\textsc{gen})\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall(\alpha_1, \ldots, \alpha_n).\tau}\ (*)$$

Seitenbedingung (*): Keine der Variablen $\alpha_1, \ldots, \alpha_n$ kommt frei im Kontext Γ vor.

Beispiel:

$$(\textsc{gen})\ \frac{(\textsc{fun})\ \dfrac{(\textsc{var})\ \dfrac{}{[\mathsf{x} \mapsto \alpha] \vdash \mathsf{x} : \alpha}}{\varnothing \vdash \mathsf{fun\ x\ ->\ x} : \alpha \to \alpha}}{\varnothing \vdash \mathsf{fun\ x\ ->\ x} : \forall\alpha.\alpha \to \alpha}$$

Kein Beispiel:

$$\frac{(\textsc{var})\ \dfrac{}{[\mathsf{x} \mapsto \alpha] \vdash \mathsf{x} : \alpha}}{[\mathsf{x} \mapsto \alpha] \vdash \mathsf{x} : \forall\alpha.\alpha}$$ falsch, da (*) nicht erfüllt

Polymorphes Typsystem, Instanziierungsregel

$$(\textsc{Inst})\ \frac{\Gamma \vdash e : \forall(\alpha_1, \ldots, \alpha_n).\tau}{\Gamma \vdash e : \tau[\theta]}$$

Hier ist $\theta = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ wie vorher eine Einsetzung von Typen (nicht Typschemata) für die Variablen $\alpha_1, \ldots, \alpha_n$.

Beispiel:

$$\dfrac{\dfrac{\dfrac{(\textsc{Var})\ \dfrac{}{\Gamma \vdash \mathtt{f} : \forall\alpha.\,\alpha \to \alpha}}{(\textsc{Inst})\ \dfrac{\Gamma \vdash \mathtt{f} : \mathtt{bool} \to \mathtt{bool}}{}} \quad \vdots}{(\textsc{App})\ \dfrac{\Gamma \vdash \mathtt{f\ true} : \mathtt{bool}}{}} \qquad \dfrac{\dfrac{\dfrac{(\textsc{Var})\ \dfrac{}{\Gamma \vdash \mathtt{f} : \forall\alpha.\,\alpha \to \alpha}}{(\textsc{Inst})\ \dfrac{\Gamma \vdash \mathtt{f} : \mathtt{int} \to \mathtt{int}}{}} \quad \vdots}{(\textsc{App})\ \dfrac{\Gamma \vdash \mathtt{f}\ 0 : \mathtt{int}}{}} \quad \vdots}{\Gamma \vdash \mathtt{if\ f\ true\ then\ f}\ 0\ \mathtt{else}\ 1 : \mathtt{int}}$$

Γ ist $[\mathtt{f} \mapsto (\forall\alpha.\,\alpha \to \alpha)]$.

Polymorphes Typsystem, Let-regel

$$(\textsc{Let})\ \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma[x \mapsto \sigma_1] \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma_2}$$

Beispiel:

$$\dfrac{(\textsc{Gen})\ \dfrac{\vdots}{\varnothing \vdash \texttt{fun x -> x} : \forall\alpha.\,\alpha \to \alpha} \qquad \dfrac{\vdots}{[\texttt{f} \mapsto (\forall\alpha.\,\alpha \to \alpha)] \vdash e : \texttt{int}}\ (\textsc{Let})}{\varnothing \vdash \texttt{let f = fun x -> x in}\ e : \texttt{int}}$$

e ist Abkürzung für $\texttt{if f true then f}\ 0\ \texttt{else}\ 1$.

Typinferenz mit Polymorphie

Die Typinferenzfunktion $\mathcal{W}$ kann auf das polymorphe Typsystem erweitert werden. [Damas-Milner Typinferenz]

Die Funktion $\mathcal{W}(\Gamma, e)$ liefert wie vorher einen Typ (kein Typschema) und eine Substitution (für freie Typvariablen in Γ) zurück (oder “fail”!).

Die Funktion $\mathcal{W}$ berechnet den allgemeinsten Typ. Das kann man ähnlich wie auf Folie 310 spezifizieren.

Typinferenz mit Polymorphie

Idee: Versuche den Variablen im Kontext ein möglichst allgemeines Typschema zuzuweisen.

Dazu werden alle mit der `let`-Regel eingeführten Variablen soweit möglich wie möglich generalisiert.

$$(\textsc{gen})\ \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_1 : \forall(\alpha_1, \ldots, \alpha_n).\, \tau_1}$$

$$(\textsc{let})\ \frac{\Gamma \vdash e_1 : \forall(\alpha_1, \ldots, \alpha_n).\, \tau_1 \qquad \Gamma[\mathsf{x} \mapsto \forall(\alpha_1, \ldots, \alpha_n).\, \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ \mathsf{x} = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Die Instanziierungsregel ist für Variablen nötig.

$$(\textsc{var})\ \frac{}{\Gamma \vdash x : \forall(\alpha_1, \ldots, \alpha_n).\tau}$$

$$(\textsc{inst})\ \frac{\Gamma \vdash x : \forall(\alpha_1, \ldots, \alpha_n).\tau}{\Gamma \vdash x : \tau[\theta]}$$

Man überlegt sich: Alle anderen Verwendungen von (GEN) und (INST) sind überflüssig.

Typinferenz mit Polymorphie

Die Definition von $\mathcal{W}$ wird angepasst, um diese beiden Verwendungen von (INST) und (GEN) zu erfassen.

- $\mathcal{W}(\Gamma, x) = (\varnothing, \tau[\theta])$, wobei τ und θ wie folgt bestimmt werden: Der Typ τ wird so gewählt, dass das Typschema $\Gamma(x)$ die Form $\forall(\alpha_1, \ldots, \alpha_n).\, \tau$ hat. Für θ wählt man eine Substitution, die jedes α_i auf eine neue Typvariable abbildet.
- $\mathcal{W}(\Gamma, \texttt{let x} = e_1 \texttt{ in } e_2)$ wird wie folgt berechnet:
  - Berechne rekursiv $(\theta_1, \tau_1) := \mathcal{W}(\Gamma, e_1)$.
  - Setze $\sigma := \forall(\alpha_1, \ldots, \alpha_n).\, \tau_1$, wobei $\alpha_1, \ldots, \alpha_n$ alle Typvariablen sind, die frei in τ_1 vorkommen, die aber nicht frei in $\Gamma[\theta_1]$ vorkommen.
  - Berechne rekursiv $(\theta_2, \tau_2) := \mathcal{W}(\Gamma[\theta_1][\texttt{x} \mapsto \sigma], e_2)$.
  - Gib $(\theta_1; \theta_2, \tau_2)$ als Ergebnis zurück.

Kontrollflussanalyse

Bei der Analyse von While-Programmen konnten wir einen Kontrollflussgraphen verwenden, der sich direkt vom Programm ablesen lässt.

In der funktionalen Sprache ist der Kontrollfluss schwieriger zu erkennen und benötigt eine eigene Analyse.

Beispiel:

$$
\begin{aligned}
&\texttt{let f} = \texttt{fun x -> x } 1 \texttt{ in}\\
&\texttt{let g} = \texttt{fun y -> y} + 2 \texttt{ in}\\
&\texttt{let h} = \texttt{fun z -> z} + 3 \texttt{ in}\\
&(\texttt{f g}) + (\texttt{f h})
\end{aligned}
$$

Sowohl g als auch h können zur Variable x in der Applikation x 1 “fließen”.

Kontrollflussanalyse

Wir möchten verstehen, welche Funktionen an welchen Programmpunkten auftreten können.

- Anwendung als Grundlage für andere Analysen
- Verallgemeinerte Anwendung: Herkunft von Datenstrukturen. Z.B.: Strings nur als Ergebnis von Sanitizing Funktionen (Abhilfe gegen XSS Attacken).

Abstraktionsterme werden mit eindeutigen Labels markiert:

$$e ::= \cdots \mid \texttt{fun}_\ell\ \texttt{x -> } e \mid \texttt{recfun}_\ell\ \texttt{f x -> } e$$

Die operationelle Semantik wird so angepasst, dass die Labels bei der Reduktion mitgeführt werden:

$$(\textsc{Fn})\ \frac{}{\texttt{fun}_\ell\ x\ \texttt{->}\ e \Downarrow \texttt{fun}_\ell\ x\ \texttt{->}\ e}$$

$$(\textsc{RecFun})\ \frac{}{\texttt{recfun}_\ell\ f\ x\ \texttt{->}\ e \Downarrow \texttt{fun}_\ell\ x\ \texttt{->}\ e[f \mapsto (\texttt{recfun}_\ell\ f\ x\ \texttt{->}\ e)]}$$

Typsystem für Kontrollflussanalyse

Die Typen werden mit Kontrollflussinformationen annotiert.

$$\tau, \tau_1, \tau_2 ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \xrightarrow{\Pi} \tau_2$$

Die Funktionstypen haben die Form $\tau_1 \xrightarrow{\Pi} \tau_2$, wobei Π eine *Menge* von Labels ist.

Intendierte Bedeutung: Das Ergebnis der Auswertung eines Terms e vom Typ $\tau_1 \xrightarrow{\Pi} \tau_2$ kann nur eine Funktion mit einem Label aus der Menge Π sein.

Approximation

Die Kontrollflussinformation ist approximativ in dem Sinne, dass jede mögliche Funktionsherkunft in den Typen aufgeführt ist, aber nicht jede aufgeführte Herkunft tatsächlich möglich sein muss.

Beispiele

- $\vdash \texttt{fun}_1\ \texttt{x} \texttt{->} \texttt{x} : \texttt{int} \xrightarrow{\{1\}} \texttt{int}$
- $\vdash \texttt{if}\ b\ \texttt{then}\ \texttt{fun}_1\ \texttt{x} \texttt{->} \texttt{x}\ \texttt{else}\ \texttt{fun}_2\ \texttt{y} \texttt{->} \texttt{y} + 1 : \texttt{int} \xrightarrow{\{1,2\}} \texttt{int}$
- $[\texttt{f} \mapsto (\texttt{int} \xrightarrow{\{2,3\}} \texttt{int}) \xrightarrow{\{1\}} \texttt{int}] \vdash e : \texttt{int}$
  wobei e den folgenden Term abkürzt:

$$\begin{aligned}&\texttt{let g} = \texttt{fun}_2\ \texttt{y} \texttt{->} \texttt{y} + 2\ \texttt{in}\\ &\texttt{let h} = \texttt{fun}_3\ \texttt{z} \texttt{->} \texttt{z} + 3\ \texttt{in}\\ &(\texttt{f g}) + (\texttt{f h})\end{aligned}$$

Typregeln für Kontrollflussanalyse

Das Typurteil hat wieder die Form $\Gamma \vdash e : \tau$. Sowohl τ als auch alle Typen in Γ sind mit Kontrollflussinformationen annotiert.

$$(\textsf{FN})\ \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathsf{fun}_\ell\ x \texttt{->}\ e : \tau_1 \xrightarrow{\Pi} \tau_2}\ \ell \in \Pi$$

$$(\textsf{RECFUN})\ \frac{\Gamma[f \mapsto (\tau_1 \xrightarrow{\Pi} \tau_2)][x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathsf{recfun}_\ell\ f\ x\ \texttt{->}\ e : \tau_1 \xrightarrow{\Pi} \tau_2}\ \ell \in \Pi$$

$$(\textsf{APP})\ \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\Pi} \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Die anderen Regeln betreffen keine Funktionstypen und werden unverändert übernommen.

Typkorrektheit

Satz (Subject Reduction):

Wenn $\vdash e : \tau$ und $e \Downarrow v$ herleitbar sind, dann auch $\vdash v : \tau$.

Man beweist diesen Korrektheitssatz durch Induktion über $e \Downarrow v$.

Der Beweis verwendet folgendes Substitutionslemma.
Lemma: Wenn $\vdash e : \tau$ und $\Gamma[x \mapsto \tau] \vdash e' : \tau'$ herleitbar sind, dann auch $\Gamma \vdash e'[x \mapsto e] : \tau'$.

Korollar (Korrektheit der Kontrollflussinformation)

Sind $\vdash e : \tau_1 \xrightarrow{\Pi} \tau_2$ und $e \Downarrow \mathsf{fun}_\ell\, x \texttt{ -> } e_1$ herleitbar, dann $\ell \in \Pi$.

Kontrollflussanalyse: Typinferenz

- Man verwendet *Mengenvariablen* als Annotierungen der Funktionstypen.
- Die Funktionen $\mathcal{W}$ und $\mathcal{U}$ liefern jeweils zusätzlich eine Liste von Bedingungen an die Mengenvariablen, so dass jede Lösung der Bedingungn eine Typisierung liefert und umgekehrt.

Beispiel: Typinferenz angewandt auf

$$(\texttt{fun}_1\ \texttt{x->x})\ (\texttt{fun}_2\ \texttt{y->y})$$

liefert den Typ $\alpha \xrightarrow{X} \alpha$, wobei X eine Mengenvariable ist, sowie die zu lösende Bedingung $2 \in X$.

Seiteneffekte

Viele funktionale Sprachen (z.B. OCaml, SML, Lisp, …) erlauben Seiteneffekte.

- mutierbarer Speicher
- Exceptions
- I/O
- …

Für Verifikation und Spezifikation ist ein gutes Verständnis von Seiteneffekten wichtig.
Beispiel: Unveränderliche Daten können ohne Synchronisation von nebenläufigen Threads geteilt werden. Die Korrektheit hängt also von der Abwesenheit bestimmter Effekte ab.

Typsysteme können zur Kontrolle von Seiteneffekten verwendet werden.

Referenztypen

Erweitere die funktionale Sprache um *Referenzen* (auf mutierbare Werte).

$$e, e_1, e_2 ::= \ldots \mid \texttt{ref}_\ell\, e \mid \texttt{!}e_1 \mid e_1 := e_2$$

- Der Term $\texttt{ref}_\ell\, e$ erzeugt eine neue Referenz mit Inhalt e.
- Der Term $\texttt{!}e_1$ gibt den Inhalt der Referenz e_1 zurück.
- Der Term $e_1 := e_2$ setzt den Inhalt der Referenz e_1 auf e_2 und gibt den alten Wert zurück.

Beispiel:

```
let x = ref 3 in
let u = !x in                (* u hat Wert 3 *)
let v = (x := !x + 1) in (* v hat Wert 3 *)
let w = !x in                (* w hat Wert 4 *)
...
```

(Label an `ref` im Beispiel weggelassen)

Java-Analogie

Die Terme entsprechen folgenden Java-Ausdrücken.

- $\texttt{ref}_\ell\ e$ entspricht `new Ref<>(e)`.
- $!e_1$ entspricht `e1.get()`.
- $e_1 := e_2$ entspricht `e1.set(e2)`.

```
class Ref<A> {
  private A value;
  public Ref(A value) { this.value = value; }
  public A set(A x) {
    A old = value; value = x; return old;
  }
  public A get() { return value; }
}
```

Referenztypen

Das einfache Typsystem wird um einen Typ $\tau\ \texttt{ref}$ für Referenzen auf mutierbare Werte des Typs τ erweitert.

$$\tau, \tau_1, \tau_2 ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \tau\ \texttt{ref}$$

Die Typregeln sind die des einfachen Typsystems. Es kommen folgende Regeln für Referenzen hinzu.

$$(\textsc{ref})\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref}_\ell\ e : \tau\ \texttt{ref}}$$

$$(\textsc{deref})\ \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash\ !e : \tau}$$

$$(\textsc{ass})\ \frac{\Gamma \vdash e_1 : \tau\ \texttt{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash\ e_1 := e_2\ : \tau}$$

Beispiel: Fakultät

Abkürzung: Schreibe $e_1; e_2$ als Abkürzung für `let _ =` e_1 `in` e_2, wobei _ eine frische Variable ist.

Fakultätsfunktion:

```
fun x ->
  let n = ref x in
  let r = ref 1 in
  let loop = recfun f u ->
      (r := !r * !n;
       n := !n - 1;
       if !n = 0 then !r else f u) in
  loop 0
```

(Labels an `ref` hier weggelassen)

Beispiel: Zähler

Der folgende Term definiert eine Zählerfunktion.

$$\texttt{let}\ n = \texttt{ref}_\ell\ 0\ \texttt{in}$$
$$\texttt{fun}\ x\ \texttt{->}\ (n\ \texttt{:=}\ \texttt{!}n + 1;\ \texttt{!}n)$$

Diese gibt beim ersten Aufruf 1 zurück, dann 2, dann 3, usw.

Typen mit Effektannotationen

Effektannotationen

Eine *Effektannotation* (kurz *Effekt*) ist eine Menge von *elementaren Effektannotationen*; diese sind:

- $\texttt{read}(\ell)$ (Lesen einer mit Label ℓ erzeugten Referenz)
- $\texttt{write}(\ell)$ (Zuweisen zu einer mit Label ℓ erzeugten Referenz)
- $\texttt{new}(\ell)$ (Erzeugen einer Referenz mit Label ℓ)

Zum Beispiel ist $\{\texttt{write}(o), \texttt{read}(o), \texttt{new}(n)\}$ ein Effekt.

Verfeinerte Typen:

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \mid \tau\ \texttt{ref}_\Pi$$

Hier bezeichnet φ eine Effektannotation und Π wie bisher eine Menge von Labels.

Bedeutung der Typen

- $\tau_1 \xrightarrow{\varphi} \tau_2$ ist der Typ aller Funktionen $\mathtt{f}$, die bei Anwendung höchstens die Effekte in φ haben.
- $\tau\ \mathtt{ref}_\Pi$ ist der Typ von Referenzen, die mit einem Label aus Π erzeugt wurden.

Beispiel: Dem Term

$$\mathtt{fun}\ x_0\ \mathtt{->}\ \mathtt{let}\ n = \mathtt{ref}_\ell\ x_0\ \mathtt{in}$$
$$\mathtt{fun}\ x\ \mathtt{->}\ (n\ \mathtt{:=}\ \mathtt{!}n + 1;\ \mathtt{!}n)$$

kann man folgenden Typ geben:

$$\mathtt{int} \xrightarrow{\{\mathtt{new}(\ell)\}} (\mathtt{int} \xrightarrow{\{\mathtt{write}(\ell), \mathtt{read}(\ell)\}} \mathtt{int})$$

Seiteneffektanalyse als Typsystem

Das Typurteil hat nunmehr die Form:

$$\Gamma \vdash e : \tau \ \& \ \varphi$$

Bedeutung:

- Unter den Typannahmen für Variablen in Γ hat e den Typ τ.
- Bei Auswertung von e treten höchstens die Effekte in φ auf.

Die Typregeln erhält man durch geeignete Annotation der einfachen Typregeln mit Effektinformation.

Seiteneffektanalyse als Typsystem

$$(\textsf{TT})\ \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool} \ \& \ \varnothing} \qquad (\textsf{FF})\ \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool} \ \& \ \varnothing}$$

$$(\textsf{CON})\ \frac{}{\Gamma \vdash n : \texttt{int} \ \& \ \varnothing} \qquad (\textsf{VAR})\ \frac{}{\Gamma \vdash x : \tau \ \& \ \varnothing}\ \Gamma(x) = \tau$$

$$(\textsf{FN})\ \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \ \& \ \varphi}{\Gamma \vdash \textsf{fun}\ x \texttt{->} e : \tau_1 \xrightarrow{\varphi} \tau_2 \ \& \ \varnothing}$$

$$(\textsf{APP})\ \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2 \ \& \ \varphi_2 \qquad \Gamma \vdash e_2 : \tau_1 \ \& \ \varphi_3}{\Gamma \vdash e_1\ e_2 : \tau_2 \ \& \ \varphi_1 \cup \varphi_2 \cup \varphi_3}$$

Fortsetzung

$$\text{(RECFUN)}\ \frac{\Gamma[f \mapsto (\tau_1 \xrightarrow{\varphi} \tau_2)][x \mapsto \tau_1] \vdash e : \tau_2 \ \& \ \varphi}{\Gamma \vdash \texttt{recfun}\ f\ x\ \texttt{->}\ e : \tau_1 \xrightarrow{\varphi} \tau_2 \ \& \ \varnothing}$$

$$\text{(IF)}\ \frac{\Gamma \vdash e_0 : \texttt{bool} \ \& \ \varphi_0 \qquad \Gamma \vdash e_1 : \tau \ \& \ \varphi_1 \qquad \Gamma \vdash e_2 : \tau \ \& \ \varphi_2}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \ \& \ \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

$$\text{(LET)}\ \frac{\Gamma \vdash e_1 : \tau_1 \ \& \ \varphi_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2 \ \& \ \varphi_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \ \& \ \varphi_1 \cup \varphi_2}$$

$$\text{(OP)}\ \frac{\Gamma \vdash e_1 : \tau_1 \ \& \ \varphi_1 \qquad \Gamma \vdash e_2 : \tau_2 \ \& \ \varphi_2}{\Gamma \vdash e_1\ \mathit{op}\ e_2 : \tau_3 \ \& \ \varphi_1 \cup \varphi_2}$$

In Regel (OP) müssen τ_1, τ_2 und τ_3 dem Operator *op* entsprechen, siehe Folie 305.

Fortsetzung

$$(\textsc{deref})\ \frac{\Gamma \vdash e : \tau\ \mathtt{ref}_{\Pi}\ \&\ \varphi}{\Gamma \vdash\ !e : \tau\ \&\ \varphi \cup \{\mathtt{read}(\ell) \mid \ell \in \Pi\}}$$

$$(\textsc{ass})\ \frac{\Gamma \vdash e_1 : \tau\ \mathtt{ref}_{\Pi}\ \&\ \varphi \qquad \Gamma \vdash e_2 : \tau\ \&\ \varphi}{\Gamma \vdash e_1 := e_2 : \tau\ \&\ \varphi \cup \{\mathtt{write}(\ell) \mid \ell \in \Pi\}}$$

$$(\textsc{ref})\ \frac{\Gamma \vdash e : \tau\ \&\ \varphi}{\Gamma \vdash \mathtt{ref}_{\ell}\, e : \tau\ \mathtt{ref}_{\{\ell\}}\ \&\ \varphi \cup \{\mathtt{new}(\ell)\}}$$

$$(\textsc{Sub})\ \frac{\Gamma \vdash e : \tau\ \&\ \varphi \qquad \tau \leqslant \tau'}{\Gamma \vdash e : \tau'\ \&\ \varphi'}\ \varphi \subseteq \varphi'$$

Das Urteil $\tau \leqslant \tau'$ ist auf Folie 351 definiert.

Beispiel

Typherleitung für folgendes Beispiel.

$$e := \texttt{fun}\, x_0 \texttt{->} \underbrace{\texttt{let}\, n = \texttt{ref}_\ell\, x_0\, \texttt{in}\, \texttt{fun}\, x \texttt{->} (n := 5;\ !n)}_{e_1}$$

Beginn der Herleitung:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[x_0 \mapsto \texttt{int}] \vdash x_0 : \texttt{int} \;\&\; \varnothing}\,(\textsf{VAR})}{[x_0 \mapsto \texttt{int}] \vdash \texttt{ref}_\ell\, x_0 : \texttt{int}\, \texttt{ref}_{\{\ell\}} \;\&\; \{\textsf{new}(\ell)\}}\,(\textsf{REF}) \qquad (1)}{[x_0 \mapsto \texttt{int}] \vdash e_1 : \texttt{int} \xrightarrow{\{\textsf{write}(\ell), \textsf{read}(\ell)\}} \texttt{int} \;\&\; \{\textsf{new}(\ell)\}}\,(\textsf{LET})}{\vdash e : \texttt{int} \xrightarrow{\{\textsf{new}(\ell)\}} (\texttt{int} \xrightarrow{\{\textsf{write}(\ell), \textsf{read}(\ell)\}} \texttt{int}) \;\&\; \varnothing}\,(\textsf{FN})$$

Teilherleitung (1) folgt auf der nächsten Folie.

Beispiel, Forts.

Teilherleitung (1):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash n : \mathtt{int\ ref}_{\{\ell\}} \ \&\ \varnothing}\,(\textsc{var})}{\Gamma_2 \vdash\ !n : \mathtt{int} \ \&\ \{\mathtt{read}(\ell)\}}\,(\textsc{deref}) \quad (2)}{\Gamma_1 \vdash n := 5;\ !n\ :\ \mathtt{int} \ \&\ \{\mathtt{write}(\ell), \mathtt{read}(\ell)\}}\,(\textsc{let})}{\Gamma \vdash \mathtt{fun}\, x \texttt{->} n := 5;\ !n\ :\ \mathtt{int} \xrightarrow{\{\mathtt{write}(\ell), \mathtt{read}(\ell)\}} \mathtt{int} \ \&\ \varnothing}\,(\textsc{fn})$$

Teilherleitung (2):

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash n : \mathtt{int\ ref}_{\{\ell\}} \ \&\ \varnothing}\,(\textsc{var}) \qquad \dfrac{}{\Gamma_1 \vdash 5 : \mathtt{int} \ \&\ \varnothing}\,(\textsc{con})}{\Gamma_1 \vdash n := 5 : \mathtt{int} \ \&\ \{\mathtt{write}(\ell)\}}\,(\textsc{ass})$$

Abkürzungen:

$$\Gamma := [x_0 \mapsto \mathtt{int}, n \mapsto \mathtt{int\ ref}_{\{\ell\}}]$$
$$\Gamma_1 := \Gamma[x \mapsto \mathtt{int}]$$
$$\Gamma_2 := \Gamma_1[\_ \mapsto \mathtt{int}]$$

Subtyping

Die Regel (Sub) erlaubt die Anpassung von Effektannotationen, so dass die Typaussage schwächer wird.

- Beispiel: Eine Funktion, die Argumente vom Typ $(\texttt{int} \xrightarrow{\{\texttt{read}(\ell),\texttt{write}(\ell)\}} \texttt{int})$ nimmt, sollte man auch auf Argumente vom Typ $(\texttt{int} \xrightarrow{\{\texttt{read}(\ell)\}} \texttt{int})$ oder $(\texttt{int} \xrightarrow{\{\texttt{write}(\ell)\}} \texttt{int})$ anwenden können.
- Das Subtyping-Urteil $\tau_1 \leqslant \tau_2$ ist definiert durch:

$$\frac{}{\tau \leqslant \tau} \qquad \frac{}{\tau\ \texttt{ref}_{\Pi} \leqslant \tau\ \texttt{ref}_{\Pi'}}\ \Pi \subseteq \Pi'$$

$$\frac{\tau_1' \leqslant \tau_1 \qquad \tau_2 \leqslant \tau_2}{\tau_1 \xrightarrow{\varphi} \tau_1' \leqslant \tau_2 \xrightarrow{\varphi'} \tau_2'}\ \varphi \subseteq \varphi'$$

Beispiel

$$\begin{array}{l}\texttt{let x = ref}_\ell\ 1\ \texttt{in}\\ \quad (\texttt{fun f -> f (fun y -> !x)} + \texttt{f (fun z -> (x := z; z))})\\ \quad (\texttt{fun g -> g}\ 1)\end{array}$$

Die beiden Argumente für f können wie folgt analysiert werden:

$$\ldots \vdash (\texttt{fun y -> !x}) : \texttt{int} \xrightarrow{\{\texttt{read}(\ell)\}} \texttt{int} \ \& \ \varnothing$$

$$\ldots \vdash (\texttt{fun z -> (x := z; z)}) : \texttt{int} \xrightarrow{\{\texttt{write}(\ell)\}} \texttt{int} \ \& \ \varnothing$$

Man gibt f den Typ

$$(\texttt{int} \xrightarrow{\{\texttt{read}(\ell),\texttt{write}(\ell)\}} \texttt{int}) \xrightarrow{\{\texttt{read}(\ell),\texttt{write}(\ell)\}} \texttt{int}\ .$$

In den Applikationen f (fun y -> !x) und f (fun z -> (x := z; z))) können die Typen der Argumente dann durch Subtyping angepasst werden.

Exceptions als Beispiel für Effektpolymorphie

Wir erweitern die funktionale Sprache um Exceptions (die Referenzen lassen wir wieder weg).

$$e ::= \cdots \mid \texttt{raise}\ s \mid \texttt{try}\ e_1\ \texttt{with}\ s\ \texttt{->}\ e_2$$

Hier läuft s über eine festgewählte Menge von Exceptions.

Beispiel: Funktion zur Berechnung von $\binom{x}{y}$.

```
fun x -> fun y ->
  let choose = recfun choose x -> fun y ->
    if x < 0 then raise x_out_of_range
    else if y < 0 | y > x then raise y_out_of_range
    else choose (x-1) y + choose (x-1) (y-1) in
  try
    choose x y
  with x_out_of_range -> 1
```

Operationelle Semantik

Die operationelle Semantik besteht aus allen bisherigen Regeln mit der zusätzlichen Seitenbedingung, dass kein Wert die Form `raise` s hat. Neue Regeln erfassen diese Möglichkeit:

Beispiel für Applikation:

$$\text{(APP)}\ \frac{e_1 \Downarrow \texttt{fun}\ x\ \texttt{->}\ e_1' \qquad e_2 \Downarrow v_2 \qquad e_1'[x \mapsto v_2] \Downarrow v}{e_1\ e_2 \Downarrow v} \quad \begin{array}{l} v_2 \neq \texttt{raise}\ s, \\ v \neq \texttt{raise}\ s' \end{array}$$

$$\text{(APPEX1)}\ \frac{e_1 \Downarrow \texttt{raise}\ s}{e_1\ e_2 \Downarrow \texttt{raise}\ s}$$

$$\text{(APPEX2)}\ \frac{e_1 \Downarrow \texttt{fun x ->}\ e_1' \qquad e_2 \Downarrow \texttt{raise}\ s}{e_1\ e_2 \Downarrow \texttt{raise}\ s}$$

$$\text{(APPEX3)}\ \frac{e_1 \Downarrow \texttt{fun}\ x\ \texttt{->}\ e_1' \qquad e_2 \Downarrow v_2 \qquad e_1'[x \mapsto v_2] \Downarrow \texttt{raise}\ s}{e_1\ e_2 \Downarrow \texttt{raise}\ s}$$

Operationelle Semantik, Forts.

$$(\textsc{raise})\ \frac{}{\texttt{raise}\ s \Downarrow \texttt{raise}\ s}$$

$$(\textsc{try1})\ \frac{e_1 \Downarrow v_1 \qquad v_1 \neq \texttt{raise}\ s}{\texttt{try}\ e_1\ \texttt{with}\ s\ \texttt{->}\ e_2 \Downarrow v_1}$$

$$(\textsc{try2})\ \frac{e_1 \Downarrow \texttt{raise}\ s \qquad e_2 \Downarrow v_2}{\texttt{try}\ e_1\ \texttt{with}\ s\ \texttt{->}\ e_2 \Downarrow v_2}$$

Annotierte Typen

Annotierte Typen:

$$\tau ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \xrightarrow{\varphi} \tau_2$$

Die Annotation φ eines Funktionstyps $\tau_1 \xrightarrow{\varphi} \tau_2$ gibt an, welche Exceptions bei der Funktionsapplikation ausgelöst werden können.

Man könnte für φ einfach Mengen von Exceptions erlauben. Dann wäre das Typsystem aber recht eingeschränkt.

Beispiel: Die Funktionskomposition

```
fun f -> fun g -> fun x -> g (f x)
```

hätte dann für beliebige Mengen E_1 und E_2 den Typ

$$(\alpha \xrightarrow{E_1} \beta) \xrightarrow{\varnothing} (\beta \xrightarrow{E_2} \gamma) \xrightarrow{\varnothing} (\alpha \xrightarrow{E_1 \cup E_2} \gamma) \quad .$$

Problem: Man kann keine Mengen Mengen E_1 und E_2 finden, die in allen Situationen passend sind.

Annotierte Typen mit Effektpolymorphie

Die Funktionskomposition

```
fun f -> fun g -> fun x -> g (f x)
```

sollte den polymorphen Typ

$$\forall \alpha, \beta, \gamma, \varepsilon_1, \varepsilon_2.\ (\alpha \xrightarrow{\varepsilon_1} \beta) \xrightarrow{\varnothing} (\beta \xrightarrow{\varepsilon_2} \gamma) \xrightarrow{\varnothing} (\alpha \xrightarrow{\varepsilon_1 \cup \varepsilon_2} \gamma)$$

haben.

Die quantifizierten Variablen können bei jeder Verwendung verschieden instantiiert werden.

Annotierte Typen mit Effektpolymorphie

Effektausdrücke:

$$\varphi \ ::= \ \varepsilon \mid \varnothing \mid \{s\} \mid \varphi_1 \cup \varphi_2$$

Hierbei ist s eine beliebige Exception und ε steht für eine *Effektvariable*.

Annotierte Typen:

$$\tau ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \xrightarrow{\varphi} \tau_2$$

Typschemata:

$$\sigma ::= \forall(\zeta_1, \ldots, \zeta_n).\tau$$

Die ζ_i bezeichnen Typ- *oder* Effektvariablen.

Typregeln

Das Typurteil hat nunmehr die Form $\Gamma \vdash e : \sigma \ \& \ \varphi$, wobei Γ Variablen auf *Typschemata* abbildet.

$$(\textsf{TT})\ \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool} \ \& \ \varnothing} \qquad (\textsf{FF})\ \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool} \ \& \ \varnothing}$$

$$(\textsf{CON})\ \frac{}{\Gamma \vdash n : \texttt{int} \ \& \ \varnothing} \qquad (\textsf{VAR})\ \frac{}{\Gamma \vdash x : \tau \ \& \ \varnothing}\ \Gamma(x) = \tau$$

$$(\textsf{FN})\ \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \ \& \ \varphi}{\Gamma \vdash \texttt{fun}\ x \texttt{->}\ e : \tau_1 \xrightarrow{\varphi} \tau_2 \ \& \ \varnothing}$$

$$(\textsf{APP})\ \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2 \ \& \ \varphi_2 \qquad \Gamma \vdash e_2 : \tau_1 \ \& \ \varphi_3}{\Gamma \vdash e_1\ e_2 : \tau_2 \ \& \ \varphi_1 \cup \varphi_2 \cup \varphi_3}$$

In den Typregeln werden Typen wieder als spezielle Typschemata aufgefasst (τ steht für $\forall().\,\tau$).

Typregeln, Fortsetzung

$$\text{(RECFUN)}\ \frac{\Gamma[f \mapsto (\tau_1 \xrightarrow{\varphi} \tau_2)][x \mapsto \tau_1] \vdash e : \tau_2 \ \& \ \varphi}{\Gamma \vdash \texttt{recfun}\ f\ x\ \texttt{->}\ e : \tau_1 \xrightarrow{\varphi} \tau_2 \ \& \ \varnothing}$$

$$\text{(IF)}\ \frac{\Gamma \vdash e_0 : \texttt{bool} \ \& \ \varphi_0 \qquad \Gamma \vdash e_1 : \tau \ \& \ \varphi_1 \qquad \Gamma \vdash e_2 : \tau \ \& \ \varphi_2}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \ \& \ \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

$$\text{(LET)}\ \frac{\Gamma \vdash e_1 : \sigma_1 \ \& \ \varphi_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2 \ \& \ \varphi_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2 \ \& \ \varphi_1 \cup \varphi_2}$$

$$\text{(OP)}\ \frac{\Gamma \vdash e_1 : \tau_1 \ \& \ \varphi_1 \qquad \Gamma \vdash e_2 : \tau_2 \ \& \ \varphi_2}{\Gamma \vdash e_1\ \mathit{op}\ e_2 : \tau_3 \ \& \ \varphi_1 \cup \varphi_2}$$

In Regel (OP) müssen τ_1, τ_2 und τ_3 dem Operator *op* entsprechen, siehe Folie 305.

Typregeln, Fortsetzung

$$(\textsc{gen})\ \frac{\Gamma \vdash e : \tau \ \& \ \varphi}{\Gamma \vdash e : \forall(\zeta_1, \ldots, \zeta_n).\tau \ \& \ \varphi}$$

$$(\textsc{inst})\ \frac{\Gamma \vdash e : \forall(\zeta_1, \ldots, \zeta_n).\tau \ \& \ \varphi}{\Gamma \vdash e : \tau[\theta] \ \& \ \varphi}$$

In der Instanziierungsregel ist θ eine Substitution von Typen für Typvariablen und Effektausdrücken für Effektvariablen.

Typregeln, Fortsetzung

$$\text{(RAISE)}\ \frac{}{\Gamma \vdash \texttt{raise}\, s : (\forall \alpha.\, \alpha)\ \&\ \{s\}}$$

$$\text{(HANDLE)}\ \frac{\Gamma \vdash e_1 : \tau\ \&\ \varphi \cup \{s\} \qquad \Gamma \vdash e_2 : \tau\ \&\ \varphi}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{with}\, s\, \texttt{->}\, e_2 : \tau\ \&\ \varphi}$$

$$\text{(SUB)}\ \frac{\Gamma \vdash e : \tau\ \&\ \varphi \qquad \tau \leqslant \tau'}{\Gamma \vdash e : \tau'\ \&\ \varphi'}\ \varphi \leqslant \varphi'$$

Das *Subeffecting*-Urteil $\varphi_1 \leqslant \varphi_2$ soll so definiert sein, dass jede Exception in φ_1 auch in φ_2 erfasst ist. Beispiele: $\{s\} \leqslant \{s\} \cup \{s'\}$ und $\varepsilon \leqslant \{s\} \cup \varepsilon$

Subtyping $\tau \leqslant \tau'$ ist wie bisher (Folie 351) definiert.

Beispiel

```
let app = fun f -> fun x -> f x in
let f = fun y -> if y < 0 then raise neg else y in
let g = fun z -> if z > 0 then raise pos else 0 - z in
app f 1 + app g (-1)
```

Dieses Programm wertet zu 2 aus.

Der Variablen app kann das folgende Typschema zugewiesen werden:

$$\forall \alpha, \beta, \varepsilon. (\alpha \xrightarrow{\varepsilon} \beta) \xrightarrow{\varnothing} (\alpha \xrightarrow{\varepsilon} \beta)$$

In den beiden Applikationen app f und app g wird dieses Typschema zu $(\texttt{int} \xrightarrow{\{\texttt{neg}\}} \texttt{int})$ und $(\texttt{int} \xrightarrow{\{\texttt{pos}\}} \texttt{int})$ instanziiert.

Zusammenfassung Typsysteme für funktionale Programme

- Typsysteme zur kompositionalen Spezifikation von Programmeigenschaften
- Das polymorphe Typsystem („let-Polymorphie“) ist Kern vieler funktionaler Sprachen (SML, Haskell, OCaml,…)
- Hindley-Milner Typinferenz
- Effekttypsysteme verallgemeinern das Typsystem durch Effektannotationen für Ausdrücke und Funktionen.
- Effekttypsysteme erlauben die strukturierte Analyse von Programmen mit Funktions- (Prozedur-, Methoden-)Variablen.
- Zur Lösung von Constraintsystemen, die bei der Typinferenz anfallen, werden verschiedene Lösungsalgorithmen herangezogen.

Wiederholung: Hoare Logik

Ein *Hoare-Tripel* ist ein Ausdruck der Form $\{P\}S\{Q\}$, wobei S Programmstück ist und P, Q Zusicherungen.

- Bedeutung: Wenn vor der Ausführung von S die Zusicherung P gilt, dann gilt danach Q.
- Zusicherungen sind logische Formeln, die Aussage über Werte von Programmvariablen (allg. Speicherzustände) ausdrücken.
- Beispiele:

$$\{x = 5\}\ [x{:=}7]^{\ell}\ \{x = 7\}$$

$$\{i = 0 \wedge n \geqslant 0 \wedge r = 1\}$$
$$\texttt{while}\ [i < n]^1\ \texttt{do}\ \left([r{:=}r * x]^2;\ [i{:=}i + 1]^3\right)$$
$$\{r = x^n\}$$

Bedeutung

Bedeutung von $\{P\}S\{Q\}$ genau definiert:

Gilt $\sigma \in [\![P]\!]$ und $\langle S, \sigma\rangle \Downarrow \sigma'$ so auch $\sigma' \in [\![Q]\!]$.

Für eine Zusicherung P schreiben wir hier $[\![P]\!]$ für die Menge aller Programmzustände σ, für welche die Zusicherung gilt.

$$
\begin{aligned}
[\![a_1 = a_2]\!] &= \{\sigma \mid [\![a_1]\!]\sigma = [\![a_2]\!]\sigma\} \\
[\![\neg P]\!] &= \{\sigma \mid \sigma \notin [\![P]\!]\} \\
[\![P \wedge Q]\!] &= [\![P]\!] \cap [\![Q]\!] \\
[\![P \vee Q]\!] &= [\![P]\!] \cup [\![Q]\!] \\
&\ldots
\end{aligned}
$$

Beispiel: $[\![\mathsf{x} = 3]\!] = \{\sigma \mid \sigma(\mathsf{x}) = 3\}$

Hoare-Regeln

$$\text{(H-While)}\ \frac{\{I \wedge b\}\, S\, \{I\}}{\{I\}\, \texttt{while}\, [b]^\ell\, \texttt{do}\, S\, \{I \wedge \neg b\}}$$

$$\text{(H-Skip)}\ \frac{}{\{P\}\, [\texttt{skip}]^\ell\, \{P\}}$$

$$\text{(H-Ass)}\ \frac{}{\{Q[x \mapsto e]\}\, [x := e]^\ell\, \{Q\}}$$

$$\text{(H-Seq)}\ \frac{\{P\}\, S_1\, \{R\} \qquad \{R\}\, S_2\, \{Q\}}{\{P\}\, S_1\, ;\, S_2\, \{Q\}}$$

$$\text{(H-If)}\ \frac{\{P \wedge b\}\, S_1\, \{Q\} \qquad \{P \wedge \neg b\}\, S_2\, \{Q\}}{\{P\}\, \texttt{if}\, [b]^\ell\, \texttt{then}\, S_1\, \texttt{else}\, S_2\, \{Q\}}$$

$$\text{(H-Weak)}\ \frac{\{P\}\, S\, \{Q\}}{\{P'\}\, S\, \{Q'\}}\ P' \Rightarrow P \text{ und } Q \Rightarrow Q'$$

Die Ausdrücke b und e müssen seiteneffektfrei sein!

Abschwächungsregel

Für die Rückwärtsanwendung der Regeln ist es sinnvoll, sie mit der Regel (Weak) zu kombinieren.

Beispiele:

$$\text{(H-Ass+Weak)}\ \frac{P \Rightarrow Q[x \mapsto e]}{\{P\}\ x := e\ \{Q\}}$$

$$\text{(H-While+Weak)}\ \frac{\{I \wedge b\}\ S\ \{I\}}{\{P\}\ \texttt{while}\ [b]^{\ell}\ \texttt{do}\ S\ \{Q\}}\ I \wedge \neg b \Rightarrow Q \text{ und } P \Rightarrow I$$

Korrektheit der Hoare Logik

Theorem

Seien $\langle S, \sigma \rangle \Downarrow \sigma'$ und $\{P\}S\{Q\}$ herleitbar. Falls $\sigma \in [\![P]\!]$, so auch $\sigma' \in [\![Q]\!]$.

Dies kann man durch Induktion über die Herleitung von $\{P\}S\{Q\}$ beweisen. Im Fall der While-Schleife benötigt man noch eine innere Induktion über die Anzahl der Reduktionsschritte.

Vollständigkeit der Hoare-Logik

Lemma

Für jedes Programmstück S und jede Nachbedingung Q ist das Hoare-Tripel mit den Hoare'schen Regeln herleitbar.

$$\{WP(S, Q)\}\ S\ \{Q\}$$

wobei

$$[\![WP(S, Q)]\!] = \{\sigma \mid \forall \sigma'. \langle S, \sigma\rangle \Downarrow \sigma' \Rightarrow \sigma' \in [\![Q]\!]\}.$$

Theorem

Jedes gültige Hoare-Tripel ist auch herleitbar.

NB: Werden die Zusicherungen in einer hinreichend starken Logik (z.B. Prädikatenlogik) ausgedrückt, so ist $WP(S, Q)$ definierbar.

Hoare-Logik als Datenflussanalyse

Die Hoare'schen Regeln ähneln in ihrer Struktur sehr stark den Typregeln für Reaching Definitions (Folie 282).

Ähnlich wie zu letzteren, gibt es auch für die Hoare-Logik eine äquivalente Formulierung als Datenflussproblem.

Der zugrundeliegende Verband ist der Potzenmengenverband über der Menge aller Programmzustände σ. Für jede Zusicherung P ist $[\![P]\!]$ ein Element dieses Verbandes.

Dieser Verband ist vollständig, hat aber keine endliche Höhe (Gegenbeispiel: $P_i = \{\sigma \mid \sigma(x) \leqslant i\}$ und $\bigcup_i P_i = \{\sigma \mid \sigma(x) \in \mathbb{N}\}$).

Datenflussinklusionen

$$HL_{entry}(\ell) \subseteq \left\{\sigma \mid \forall \sigma'.\, \sigma \to^{\ell} \sigma' \Rightarrow \sigma' \in HL_{exit}(\ell)\right\}$$

$$HL_{exit}(\ell) \subseteq \bigcap_{\ell':\ell\to\ell'} [\![\neg cond(\ell,\ell')]\!] \cup HL_{entry}(\ell')$$

Hierbei bezeichnet $cond(\ell,\ell')$ die wie folgt definierte Bedingung zum Beschreiten der Kante von ℓ nach ℓ'.

- Für jede Ja-Kante $[b]^{\ell} \xrightarrow{\text{ja}} [\ldots]^{\ell'}$: $cond(\ell,\ell') := b$.
- Für jede Nein-Kante $[b]^{\ell} \xrightarrow{\text{nein}} [\ldots]^{\ell'}$: $cond(\ell,\ell') := \neg b$.
- Für alle anderen Kanten: $cond(\ell,\ell') := \texttt{true}$.

Die Notation $\sigma \to^{\ell} \sigma'$ bedeutet, dass σ' Folgezustand von σ beim Abarbeiten von ℓ ist. NB $\sigma' = \sigma$, falls ℓ eine Abfrage ist.

Datenflussinklusionen

$$HL_{entry}(\ell) \subseteq \left\{\sigma \mid \forall \sigma'.\, \sigma \to^{\ell} \sigma' \Rightarrow \sigma' \in HL_{exit}(\ell)\right\}$$
$$HL_{exit}(\ell) \subseteq \bigcap_{\ell' : \ell \to \ell'} [\![\neg cond(\ell, \ell')]\!] \cup HL_{entry}(\ell')$$

Jede Lösung dieses Datenflussproblems (also ein Prädikat für jedes $HL_{entry/exit}(\ell)$) ist korrekt im folgenden Sinne:

Ist $\sigma_1 \to^{\ell_1} \sigma_2 \to^{\ell_2} \cdots \to^{\ell_{n-1}} \sigma_n$ eine Abarbeitung im Kontrollflussgraphen und ist $\sigma_1 \in HL_{entry}(\ell_1)$, so folgt $\sigma_n \in HL_{exit}(\ell_{n-1})$.

Datenflussgleichungen

Nach dem Satz von Knaster-Tarski kann man die größte Lösung der Datenflussbedingungen durch Bestimmung der größten Lösung folgenden Gleichungssystems bestimmen.

$$HL_{entry}(\ell) = \left\{(\sigma_{init}, \sigma) \mid \forall \sigma'.\, \sigma \to^{\ell} \sigma' \Rightarrow (\sigma_{init}, \sigma') \in HL_{exit}(\ell)\right\}$$

$$HL_{exit}(\ell) = \bigcap_{\ell' : \ell \to \ell'} [\![\neg cond(\ell, \ell')]\!] \cup HL_{entry}(\ell')$$

Da der hier zugrundeliegende Verband keine endliche Höhe hat, kann man die Lösung jedoch im allgemeinen nicht durch endlich viele Iterationen finden.

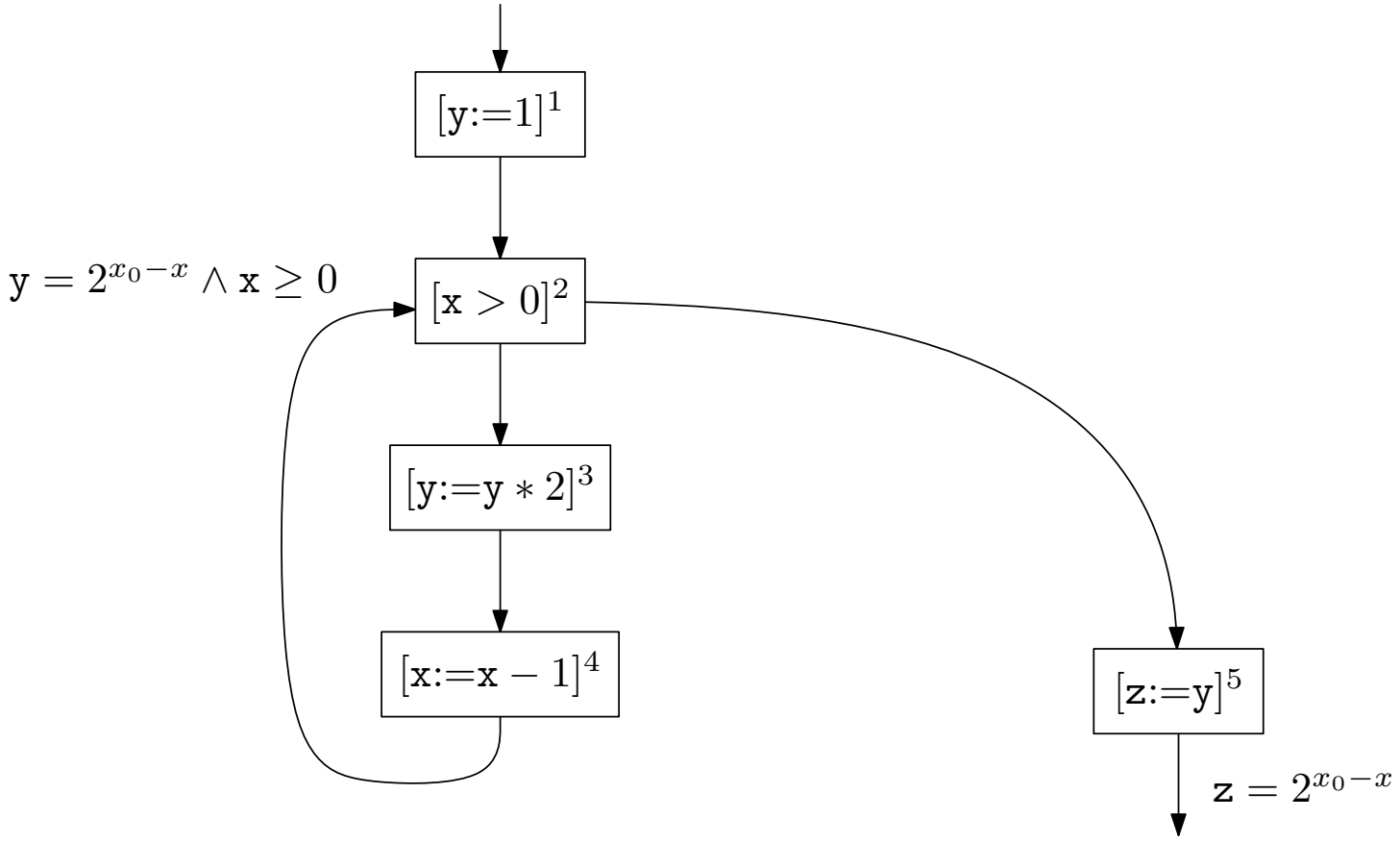
Floyd-Hoare

Um ein durch einen beliebigen Kontrollflussgraphen gegebenes Programm zu verifizieren, kann man von Hand an jede Kante geeignete Zusicherungen schreiben und dann überprüfen, dass die Datenflussbedingungen erfüllt sind.

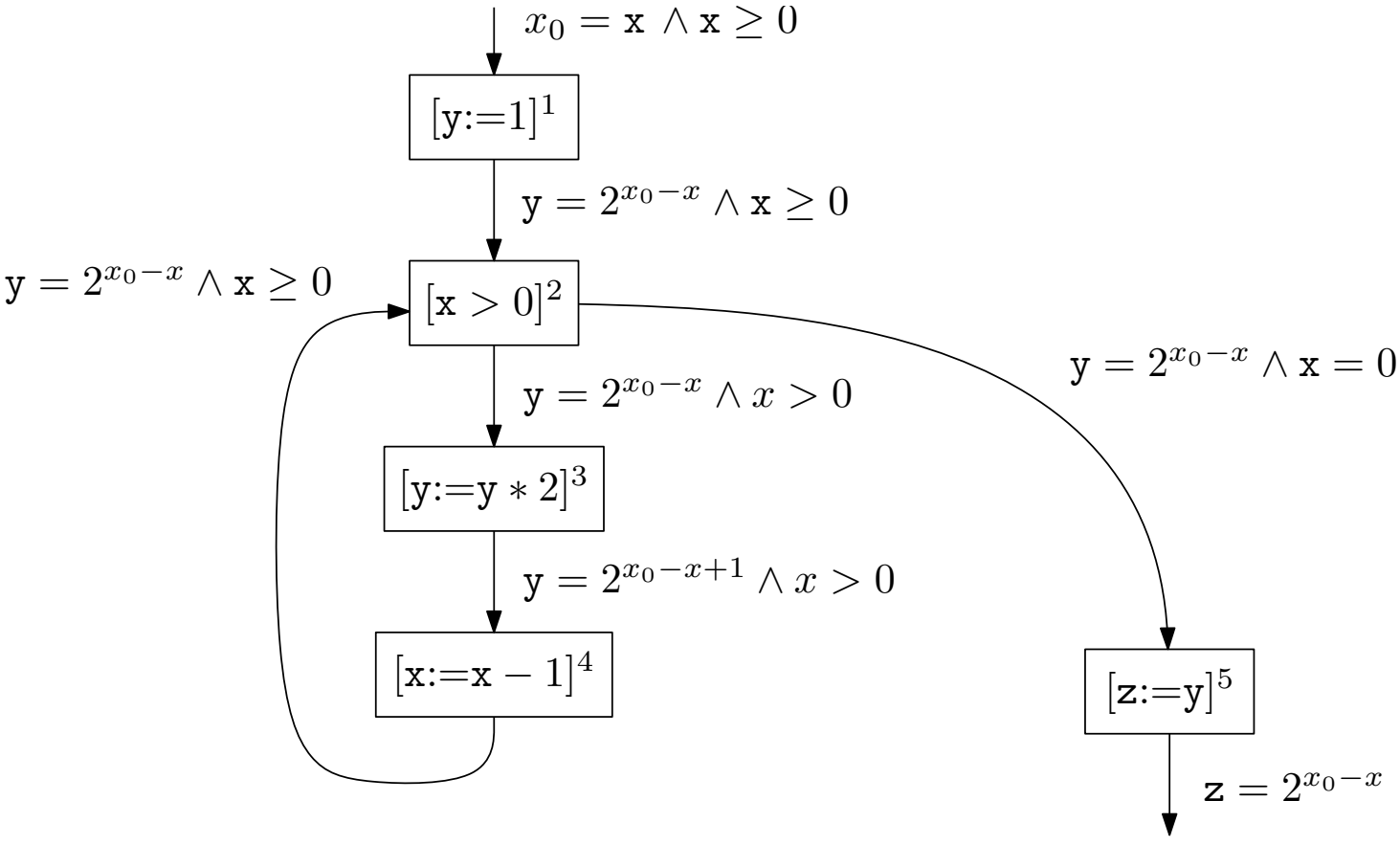
Man kann Zusicherungen auch weglassen und mit den Datenflussinklusionen automatisch rekonstruieren. Es genügt, dass auf jeder Schleife im Kontrollflussgraphen mindestens eine Zusicherung steht.

Diese als “Floyd-Hoare-Formalismus” bezeichnete Vorgehensweise ist günstig für unstrukturierte Programme, z.B. Assembler, oder komplizierte Schleifenkonstrukte (do-while, break, etc).

Floyd-Hoare: Beispiel



Floyd-Hoare: Beispiel



Zusammenfassung: Hoare Logik

- Die Hoare'schen Regeln erlauben die kompositionale Verifikation imperativer Programme.
- Man kann die Hoare'schen Regeln als Typsystem auffassen. Der Korrektheitsbeweis erfolgt wie bei den Typsystemen für Programmanalysen.
- Umgekehrt kann man die Hoare-Logik auch als Datenflussanalyse formulieren. Man kommt so zum Floyd-Hoare Formalismus, der sich besonders für unstrukturierte Programme eignet.