Kapitel III

Programmanalyse und Typsysteme

Inhalt Kapitel 3

Motivation

- "Exakte Methoden" wie in den ersten beiden Kapiteln sind auf Systeme mit nicht zu großem Zustandsraum beschränkt.
- Durch entsprechend abstrakte Modellierung kann man zwar den Zustandsraum hinreichend verkleinern; der Modellierungsschritt bleibt dann aber unverifiziert.
- Dieses und das nächste Kapitel befassen sich mit Analysemethoden für Software realistischer Größe.
- Die zu beweisenden Eigenschaften sind allerdings entweder schwächer, oder erfordern menschliche Hilfe.
- Beispiele "schwächerer Eigenschaften": Abwesenheit von Laufzzeitfehleren wie Division durch Null, falscher Speicherzugriff, ungefangene Exceptions, Verletzung von Ressourcenschranken und -protokollen (z.B. Dateizugriff).
- Beispiele für "menschliche Hilfe": Typannotate, Zusicherungen, Invarianten, formale Beweise.

Die Darstellung dieses Kapitels folgt dem Buch *Principles of*

Einfache While-Sprache

- Arithmetische Ausdrücke:

$$a ::= x \mid n \mid a_1 op_a a_2$$

- Boole'sche Ausdrücke:

$$b ::= \texttt{true} \mid \texttt{false} \mid \texttt{not } b \mid b_1 op_n b_2 \mid a_1 op_r a_2$$

- Programmstücke (statements):

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

Hierbei bezeichnen: op_a arithmetische Operatoren wie $+$; op_b Boole'sche Operatoren wie $\wedge$; op_r Relationsoperatoren wie $\leqslant$.

x rangiert über integer-wertige Programmvariablen, n über Integerkonstanten, ℓ über Labels (konkret: natürliche Zahlen), die elementare Programmteile markieren. Jedes Label soll nur einmal vorkommen (vgl.: Zeilennummern)

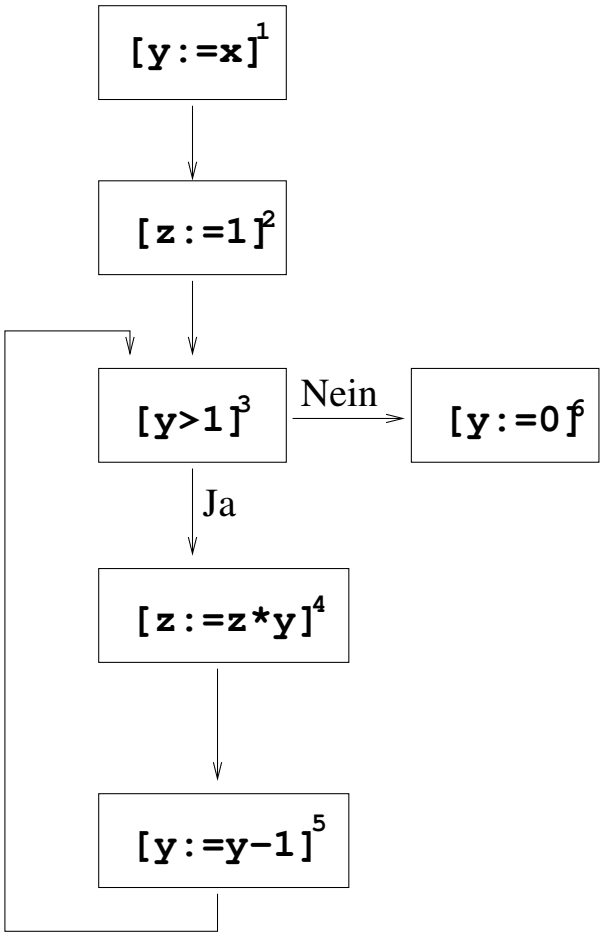
Beispielprogramm

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

Alternative Notation:

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

Kontrollflussgraph



- Pfade im Kontrollflussgraphen entsprechen möglichen Ausführungssequenzen;
- Erinnerung: Kontrollfluss = "der Computer", also was und in welcher Reihenfolge es gemacht wird;
- Knoten des Kontrollflussgraphen sind die elementaren Anweisungen, also die, die ein Label tragen;
- Knoten, die unmittelbar aufeinanderfolgen *können*, werden durch Kanten verbunden.
- Bei Fallunterscheidungen können die Kanten mit Antworten beschriftet werden.

Reichweite von Zuweisungen

Definition

Eine *Wertzuweisung* $x := a$ *erreicht* einen bestimmten Programmpunkt, wenn es eine Programmausführung gibt, die diesen Programmpunkt über die Wertzuweisung erreicht und zwischen der Wertzuweisung und dem Programmpunkt die Variable x nicht verändert wird.

Im Beispiel erreicht die Zuweisung `y:=x` bei Label 1 den Eingang zur Instruktion `z:=1` bei 2.

Für Programmpunkt ℓ bezeichnet $RD_{entry}(\ell)$ die Zuweisungen, die seinen (ℓ's) Eingang erreichen und $RD_{exit}(\ell)$ die, die seinen Ausgang erreichen.

Detailliertes Beispiel

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

Mit $(x, ?)$ bezeichnen wir die Zuweisung eines Startwertes außerhalb des analysierten Programmfragments.

Für Programmpunkte ℓ in denen keine Variable verändert wird, stimmen $RD_{entry}(\ell)$ und $RD_{exit}(\ell)$ überein.

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

Datenflussgleichungen für RD_{exit}

$$
\begin{array}{lcl}
RD_{exit}(1) & = & (RD_{entry}(1) \setminus \{(\mathtt{y}, \ell) \mid \ell = 1..6, ?\}) \cup \{(\mathtt{y}, 1)\} \\
RD_{exit}(2) & = & (RD_{entry}(2) \setminus \{(\mathtt{z}, \ell) \mid \ell = 1..6, ?\}) \cup \{(\mathtt{z}, 2)\} \\
RD_{exit}(3) & = & RD_{entry}(3) \\
RD_{exit}(4) & = & (RD_{entry}(4) \setminus \{(\mathtt{z}, \ell) \mid \ell = 1..6, ?\}) \cup \{(\mathtt{z}, 4)\} \\
RD_{exit}(5) & = & (RD_{entry}(5) \setminus \{(\mathtt{y}, \ell) \mid \ell = 1..6, ?\}) \cup \{(\mathtt{y}, 5)\} \\
RD_{exit}(6) & = & (RD_{entry}(6) \setminus \{(\mathtt{y}, \ell) \mid \ell = 1..6, ?\}) \cup \{(\mathtt{y}, 6)\}
\end{array}
$$

Allgemeines Schema:

$$
\begin{array}{lcl}
RD_{exit}(\ell) & = & (RD_{entry}(\ell) \setminus \textit{“Zuweisungen, die entwertet werden”}) \cup \\
 & & \textit{“neu gemachte Zuweisungen”}
\end{array}
$$

Datenflussgleichungen für RD_{entry}

$$
\begin{aligned}
RD_{entry}(1) &= \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\} \\
RD_{entry}(2) &= RD_{exit}(1) \\
RD_{entry}(3) &= RD_{exit}(2) \cup RD_{exit}(5) \\
RD_{entry}(4) &= RD_{exit}(3) \\
RD_{entry}(5) &= RD_{exit}(4) \\
RD_{entry}(6) &= RD_{exit}(3)
\end{aligned}
$$

Allgemeines Schema:

$$
\begin{aligned}
RD_{entry}(\ell) &= \{(x, ?) \mid x \text{ Programmvariable}\} \cup \\
&\quad \bigcup_{\ell' : \ell' \to \ell} RD_{exit}(\ell'), \text{ falls } \ell \text{ Einstiegspunkt} \\
RD_{entry}(\ell) &= \bigcup_{\ell' : \ell' \to \ell} RD_{exit}(\ell'), \text{ sonst}
\end{aligned}
$$

Kleinste Lösung

- Wir interessieren uns für die kleinste Lösung dieser Gleichungen. ("Kleinst" im Sinne von komponentenweiser Inklusion)
- Jede Lösung ist sicher in dem Sinne, dass jede Reichweitenbeziehung auch erfasst wird.
- Es gibt Lösungen, die mehr Reichweitenbeziehungen aufführen, als tatsächlich vorhanden sind, z.B. bleiben die Gleichungen gültig, wenn man zu den *RD*-Mengen in der Schleife jeweils den Eintrag $(\mathtt{x}, 5)$ hinzunimmt.
- Auch die kleinste Lösung kann überflüssige Einträge enthalten. Beispiel:

$$\mathtt{while}\ [\mathtt{true}]^1\ \mathtt{do}\ [\mathtt{skip}]^2; [\mathtt{x:=y}]^3$$

Hier gilt selbst für die kleinste Lösung $RD_{entry}(3) = (\mathtt{y}, ?)$.

Es gibt kompliziertere Beispiele und im allgemeinen sind die semantisch definierten Reichweiten unentscheidbar.

Berechnung der kleinsten Lösung

Die kleinste Lösung eines Systems von solchen Mengengleichungen kann man berechnen, indem man für jede der gesuchten Mengen *RD* eine mit $\varnothing$ initialisierte *dynamische Menge* vorsieht und solange Mengen durch die rechten Seiten ihrer definierenden Gleichungen ersetzt, bis nichts mehr passiert.

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	$(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)$	$(\mathtt{y},1)$
2	$\varnothing$	$(\mathtt{z},2)$
3	$\varnothing$	$\varnothing$
4	$\varnothing$	$(\mathtt{z},4)$
5	$\varnothing$	$(\mathtt{y},5)$
6	$\varnothing$	$(\mathtt{y},6)$

$\rightsquigarrow$

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

Fortsetzung 1

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

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

$\rightsquigarrow$

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

$\rightsquigarrow$

Fortsetzung 2

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

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

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

Constraint-basierter Ansatz

Anstatt Gleichungen aufzustellen, kann man auch Bedingungen formulieren, denen die unbekannten Mengen RD_{entry} und RD_{exit} genügen müssen.

Diese Bedingungen ("*constraints*") unterscheiden sich von den Gleichungen nur durch die Verwendung von $\supseteq$ anstelle von $=$. Z.B. hat man

$$RD_{exit}(5) \quad \supseteq \quad (RD_{entry}(1) \setminus \{(y, \ell) \mid \ell = 1..6, ?\}) \cup \{(y, 5)\}$$

oder noch "konkreter" aber äquivalent:

$$\begin{array}{lcl} RD_{exit}(5) & \supseteq & RD_{entry}(5) \setminus \{(y, \ell) \mid \ell = 1..6, ?\} \\ RD_{exit}(5) & \supseteq & \{(y, 5)\} \end{array}$$

Die erste Bedingung kommt daher, dass alle eingangs vorhandenen Reichweiten, die nicht überschrieben werden, fortgelten. Die zweite Bedingung kommt von der explizit hinzugekommenen Zuweisung und deren Reichweite.

Constraint-basierter Ansatz, Forts.

Die Bedeutung des Constraint-basierten Ansatzes liegt darin, dass man sie inkrementell erzeugen kann und nicht die gesamte Gleichung für eine linke Seite (hier RD_{exit}) auf einmal aufstellen muss.
Die "Gleichung" $RD_{exit}(5) = \{(\mathtt{y}, 5)\}$ wäre ja nicht richtig.

Man löst ein System von Constraints ganz ähnlich wie ein Gleichungssystem: Für jede linke Seite eine mit $\varnothing$ initialisierte dynamische Menge vorhalten; solange die rechten Seiten auswerten und dazugeben, bis sich nichts mehr ändert.

Anwendung: Constant folding

Mit den folgenden Regeln kann man ein gegebenes Programm u.U. verbessern:

- Werte rechte Seiten ohne Variablen zu Konstante aus. Z.B.: $\texttt{x:=5*9-1} \leadsto \texttt{x:=44}$.
- Propagiere Konstantendefinitionen: Falls $[x\texttt{:=}n]^\ell$ und (x, ℓ) der einzige Eintrag zu x in $RD_{entry}(\ell')$ ist, so ersetze $[y\texttt{:=}a]^{\ell'}$ durch $[y\texttt{:=}a[n/x]]^{\ell'}$. Analog ersetze $[b]^{\ell'}$ durch $[b[n/x]]^{\ell'}$
  Z.B.: Falls $[\texttt{x:=44}]^5$ und $RD_{entry}(7) = \{(\texttt{x}, 5), (\texttt{y}, ?)\}$, dann $[\texttt{y:=2*x-3}]^7 \leadsto [\texttt{y:=85}]^7$

Verfügbare Ausdrücke

Wir interessieren uns jetzt dafür, welche arithmetischen Ausdrücke an einem bestimmten Programmpunkt verfügbar 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:

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

Datenflussgleichungen

Wir bezeichnen mit $AE_{entry}(\ell)$ bzw. $AE_{exit}(\ell)$ die bei Ein- und Ausgang eines Labels verfügbaren Ausdrücke. Es gelten die folgenden Datenflussgleichungen:

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

wobei B^ℓ das durch ℓ beschriftete Programmstück ("Block") ist und

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

Datenflussgleichungen im Beispiel

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

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

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

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

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	$\varnothing$	{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:

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

Hier gelten die Gleichungen

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

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

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

Vollständiger Verband

Definition (Vollständiger Verband)

Eine partielle Ordnung $(L, \sqsubseteq)$ heißt vollständiger Verband, wenn jede Teilmenge $U \subseteq L$ ein Supremum $\bigsqcup U$ in L hat.

- Beispiele: $(\mathcal{P}(X), \subseteq)$ (Potenzmengenverband einer Menge); $\mathcal{P}(X) \times \mathcal{P}(Y)$ mit komponentenweiser Inklusion; $\mathcal{P}(X)^Y$ mit punktweiser Inklusion, ...
- Kleinstes Element $\bot = \bigsqcup \varnothing$, größtes Element $\top = \bigsqcup L$;
- Infima, z.B.: $x \sqcap y = \bigsqcup \{z \mid z \sqsubseteq x, z \sqsubseteq y\}$;

Fixpunkte

Theorem (Knaster-Tarski)

Es sei $(L, \sqsubseteq)$ ein vollständiger Verband und $F : L \to L$ monoton, d.h., $x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)$. Dann hat F einen Fixpunkt, d.h. es existiert $x \in L$ mit $F(x) = x$. Darüberhinaus 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).
- Das Supremum von $P \subseteq L$, wobei P nur Fixpunkte enthält, ist der kleinste Fixpunkt von $F|_U$ wobei $U = \{x \mid P \sqsubseteq x\}$ (Beweis an der Tafel).

Endliche Höhe

Ein Verband hat endliche Höhe h, wenn aus

$$x_0 \sqsubsetneq x_1 \sqsubsetneq x_2 \sqsubsetneq x_3 \sqsubsetneq \cdots \sqsubsetneq x_n$$

folgt $n \leqslant h$. Die Höhe ist also die maximale Länge einer echt aufsteigenden Kette.
Beispiele: Potenzmengenverband über endlicher Menge, Untervektorraumverband eines endlich dimensionalen Vektorraums.
Wenn L endliche Höhe h hat, so gilt für den kleinsten Fixpunkt $\mathrm{lfp}(F)$ einer monotonen Funktion:

$$\mathrm{lfp}(F) = F^h(\bot)$$

und für den größten Fixpunkt $\mathrm{gfp}(F)$:

$$\mathrm{gfp}(F) = F^h(\top)$$

Anwendung: Systeme von Mengengleichungen

Sei $\mathcal{G}$ eine Menge und $X_1, \ldots X_n$ Variablen mit Werten aus $\mathcal{P}(\mathcal{G})$.

Eine Belegung der Variablen ist eine Funktion $\eta : \{X_1, \ldots, X_n\} \to \mathcal{P}(\mathcal{G})$, die jeder Variablen eine Teilmenge von $\mathcal{G}$ zuweist.

Diese Belegungen bilden mit punktweiser Inklusion einen vollständigen Verband.

Sei nun für jede Variable X_i eine monotone "rechte Seite" F_i gegeben, die einer Belegung η einen Wert $F_i(\eta) \in \mathcal{P}(\mathcal{G})$ zuweist. (Monotonie: $\eta \sqsubseteq \eta' \Rightarrow F_i(\eta) \sqsubseteq F_i(\eta')$).

Fortsetzung

Wir interessieren uns für Belegungen η für die gilt: $\eta(X_i) = F_i(\eta)$ für $i = 1, \ldots, n$, also für "Lösungen" des Gleichungssystems

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

Diese entsprechen gerade Fixpunkten von F gegeben durch $F(\eta)(X_i) = F_i(\eta)$, wobei F nunmehr eine monotone Funktion auf dem Verband der Belegungen ist.

Sie existieren also und können, falls $\mathcal{G}$ endlich ist, durch Iteration berechnet werden.

Alle bisherigen Beispiele von Datenflussgleichungen hatten dieses Format.

Liveness von Variablen

Eine Programmvariable ist an einem bestimmten Programmpunkt *live*, wenn ihr vor diesem Programmpunkt ein Wert zugewiesen wurde und dieser Wert später noch gebraucht 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.

Beispiel:

$$[\texttt{x:=2}]^1; [\texttt{y:=4}]^2; [\texttt{x:=1}]^3; (\texttt{if } [\texttt{y>x}]^4 \texttt{ then } [\texttt{z:=y}]^5 \texttt{ else } [\texttt{z:=y*y}]^6); [\texttt{x:=z}]^7$$

Live-Variablen Gleichungen

Wir bezeichnen mit $LV_{entry}(\ell)$ bzw. $LV_{exit}(\ell)$ die Mengen der Variablen, die bei Ein- und Ausgang eines Labels live sind. Es gelten die folgenden Datenflussgleichungen:

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

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

Es handelt sich um eine Rückwärtsanalyse, da Information entgegen der Kanten im Kontrollflussgraphen propagiert wird. Wir gehen davon aus, das am Programmende keine Variable live ist. Alternativ können wir bestimmte Variablen als “Output” deklarieren und dann zur entsprechenden LV_{exit} hinzugeben.

Datenflussgleichungen im Beispiel

$$[\texttt{x:=2}]^1; [\texttt{y:=4}]^2; [\texttt{x:=1}]^3; (\texttt{if } [\texttt{y>x}]^4 \texttt{ then } [\texttt{z:=y}]^5 \texttt{ else } [\texttt{z:=y*y}]^6); [\texttt{x:=z}]^7$$

$$\begin{array}{lcl}
LV_{entry}(1) & = & LV_{exit}(1) \setminus \{\texttt{x}\} \\
LV_{entry}(2) & = & LV_{exit}(2) \setminus \{\texttt{y}\} \\
LV_{entry}(3) & = & LV_{exit}(3) \setminus \{\texttt{x}\} \\
LV_{entry}(4) & = & LV_{exit}(4) \cup \{\texttt{x}, \texttt{y}\} \\
 & & \\
LV_{entry}(5) & = & (LV_{exit}(5) \setminus \{\texttt{z}\}) \cup \{\texttt{y}\} \\
LV_{entry}(6) & = & (LV_{exit}(6) \setminus \{\texttt{z}\}) \cup \{\texttt{y}\} \\
LV_{entry}(7) & = & \{\texttt{z}\}
\end{array}$$

$$\begin{array}{lcl}
LV_{exit}(1) & = & LV_{entry}(2) \\
LV_{exit}(2) & = & LV_{entry}(3) \\
LV_{exit}(3) & = & LV_{entry}(4) \\
LV_{exit}(4) & = & LV_{entry}(5) \cup \\
 & & LV_{entry}(6) \\
LV_{exit}(5) & = & LV_{entry}(7) \\
LV_{exit}(6) & = & LV_{entry}(7) \\
LV_{exit}(7) & = & \emptyset
\end{array}$$

Kleinste Lösung

$$[\texttt{x:=2}]^1; [\texttt{y:=4}]^2; [\texttt{x:=1}]^3; (\texttt{if } [\texttt{y>x}]^4 \texttt{ then } [\texttt{z:=y}]^5 \texttt{ else } [\texttt{z:=y*y}]^6); [\texttt{x:=z}]^7$$

ℓ	LV_{entry}	LV_{exit}
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{\texttt{y}\}$
3	$\{\texttt{y}\}$	$\{\texttt{x}, \texttt{y}\}$
4	$\{\texttt{x}, \texttt{y}\}$	$\{\texttt{y}\}$
5	$\{\texttt{y}\}$	$\{\texttt{z}\}$
6	$\{\texttt{y}\}$	$\{\texttt{z}\}$
7	$\{\texttt{z}\}$	$\emptyset$

Wir interessieren uns für die kleinste Lösung, da die Gleichungen eine Approximation von oben darstellen.

Typsystem für Reaching Definitions

$$\frac{}{[x:=a]^{\ell'} : D \to D \backslash \{(x,\ell) \mid \ell \in \mathbf{Lab} \cup \{?\}\} \cup \{(x,\ell')\}} \quad (\text{ASS})$$

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

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

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

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

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

Bedeutung des Typsystems

- Mit dem Typsystem lassen sich *Typurteile* der Form $S : D_1 \to D_2$ herleiten, wobei D_1, D_2 Mengen von Zuweisungen, also (x, ℓ) und $(x, ?)$ sind.
-
- Ist $RD_{entry/exit}(-)$ Lösung des Constraintsystems, so lässt sich für jedes Programmstück S das Typurteil

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

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.

Beispiel

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

Es sei $D := \{(\texttt{x},?),(\texttt{y},1),(\texttt{y},5),(\texttt{z},2),(\texttt{z},4)\}$.

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

Operationelle Semantik

Programmzustände (σ) sind Abbildungen von Variablen auf Werte. Die Urteile

- $\langle S, \sigma \rangle \rightarrow \sigma'$ (Programmstück S liefert bei Startzustand σ den Endzustand σ')
- $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$ (Die Auswertung von S im Zustand σ führt zum Zwischenzustand σ'; es bleibt noch S' abzuarbeiten.)

werden durch folgende Regeln definiert:

Regeln für die operationelle Semantik

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

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \to \sigma} \quad (\text{SKIP})$$

$$\frac{\langle S_1, \sigma \rangle \to \langle S_1', \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \to \langle S_1'; S_2, \sigma' \rangle} \quad (\text{SEQ1})$$

$$\frac{\langle S_1, \sigma \rangle \to \sigma' \qquad \langle S_2, \sigma' \rangle \to \langle S_2', \sigma'' \rangle}{\langle S_1; S_2, \sigma \rangle \to \langle S_2', \sigma'' \rangle} \quad (\text{SEQ2})$$

$$\frac{\langle S_1, \sigma \rangle \to \sigma' \qquad \langle S_2, \sigma' \rangle \to \sigma''}{\langle S_1; S_2, \sigma \rangle \to \sigma''} \quad (\text{SEQ3})$$

Regeln für die operationelle Semantik, Forts.

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

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

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

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

Instrumentierung der operationellen Semantik

- Für den Korrektheitsbeweis der *RD*-Analyse reichern wir die Zustände um eine Komponente an, die die Menge der aktiven Zuweisungen mitführt.
- Mit $\sigma.D$ bezeichnen wir diese Komponente.
- Der Anfangszustand σ_I setzt alle Variablen auf 0 und es gilt $\sigma_I.D = \{(x, ?) \mid x \text{ Programmvariable}\}$.
- Die Regel [ass] ist wie folgt anzupassen:

$$\frac{}{\langle [x := a]^\ell, \sigma \rangle \to \sigma[x \mapsto [\![a]\!]\sigma][D \mapsto \sigma.D \setminus \{(x, \ell) \mid \ell \text{ Label oder } ?\} \cup \{(x, \ell)\}]} \text{(ASS)}$$

Korrektheit des Typsystems für *RD*

Korrektheit des *RD*-Typsystems

Es sei $S : D_1 \to D_2$ und $\sigma.D \subseteq D_1$.

- 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 $S' : D_m \to D_2$.

Der Beweis erfolgt durch simultane Induktion über die Herleitung von $\langle S, \sigma \rangle \to X$ (erste Priorität) und über die Herleitung von $S : D_1 \to D_2$ (zweite Priorität).

Alternative: Big-Step Semantik. Hier gibt es nur Urteile der Form $\langle S, \sigma \rangle \to \sigma'$. Die Korrektheitsaussage vereinfacht sich etwas.

Typinferenz

Man kann das Typsystem als Zwischenschritt zum Beweis der Korrektheit der Programmanalyse einsetzen: Jede Lösung der Datenflussgleichungen liefert Typherleitung; jede Typherleitung liefert semantisch korrekte Aussage.

Man kann alternativ auch gleich einen Algorithmus zur Findung einer Typherleitung konstruieren: Hierzu baut man eine "Skelett-Typherleitung", die Mengenvariablen anstelle von konkreten Mengen enthält. Die Nebenbedingungen in den Regeln wie *sub* werden dann zu Constraints deren Lösungen den Typherleitungen entsprechen.

Typsystem für Live-Variablen

Für live Variablen verwenden wir ein ähnliches Format für Typurteile: $S : L_1 \to L_2$, wobei L_1, L_2 Mengen von Variablen sind. Die Bedeutung kann hier deklarativer gefasst werden und bedarf auch keiner Instrumentierung:
Wir schreiben $\sigma \restriction_L$ für die Einschränkung der Abbildung σ auf L. Also bedeutet $\sigma \restriction_L = \sigma' \restriction_L$, dass σ und σ' auf den Variablen aus L übereinstimmen.

Korrektheitseigenschaft des Live-Variablen Typsystems

Es gelte $S : L_1 \to L_2$ und $\sigma \restriction_{L_1} = \sigma' \restriction_{L_1}$.

- Falls $\langle S, \sigma \rangle \to \sigma_1$, so existiert σ_1' mit $\langle S, \sigma' \rangle \to \sigma_1'$ und $\sigma_1 \restriction_{L_2} = \sigma_1' \restriction_{L_2}$.
- Falls $\langle S, \sigma \rangle \to \langle S', \sigma_1 \rangle$, so existieren L_m und σ_1' mit $S' : L_m \to L_2$ und $\langle S, \sigma' \rangle \to \langle S', \sigma_1' \rangle$ und $\sigma_1 \restriction_{L_m} = \sigma_1' \restriction_{L_m}$

Typsystem für Live Variablen

$$\frac{}{[x:=a]^{\ell'} : L\backslash\{x\} \cup FV(a) \to L} \quad (\text{ASS})$$

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

$$\frac{S_1 : L_1 \to L_2 \qquad S_2 : L_2 \to L_3}{S_1; S_2 : L_1 \to L_3} \quad (\text{SEQ})$$

$$\frac{S_1 : L_1 \to L_2 \qquad S_2 : L_1 \to L_2}{\texttt{if } [b]^{\ell} \texttt{ then } S_1 \texttt{ else } S_2 : L_1 \to L_2} \quad (\text{IF})$$

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

$$\frac{S : L_1 \to L_2 \qquad L_1 \subseteq L_1' \qquad L_2' \subseteq L_2}{S : L_1' \to L_2'} \quad (\text{SUB})$$

Zusammenfassung Typsysteme

- 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 Typsysteme, Forts.

- Typkorrektheit wird in der Regel durch doppelte Induktion über operationelle Semantik und Typherleitungen bewiesen. Die Möglichkeit über die operationelle Semantik zu induzieren ergibt sich, da Typkorrektheit meist die Form "Wenn $\langle S, \sigma \rangle \rightarrow \ldots$, dann ... hat.
- Durch Rückwärtsanwendung der Typregeln ("Skelett-Herleitung") können 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

- Analysen für funktionale Sprachen und Sprachen mit Prozeduren und Methoden lassen sich am besten als Typsystem formulieren.
- Diese Typsysteme verallgemeinern das von SML bekannte Typsystem um Effektannotationen ("Typ- und Effektsysteme")
- Wir betrachten hier Typsysteme für die Erkennung von Speicherzugriffen und Exceptions in (unreinen) funktionalen Sprachen.
- Die Begriffe Polymorphie und polymorphe Rekursion werden dabei erläutert.

Funktionale Sprache

Terme:

$$\begin{array}{rcl} e & ::= & c \mid x \mid \texttt{fn}_\pi x \texttt{=>} e_0 \mid \texttt{fun}_\pi\ f\ x \texttt{=>} e_0 \mid e_1\ e_2 \\ & \mid & \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x \texttt{=} e_1 \texttt{in}\ e_2 \mid e_1\ op\ e_2 \end{array}$$

π rangiert über "Labels" und dient der Markierung von Programmpunkten.
Beispiele:

- $(\texttt{fn}_X\ \texttt{x => x})(\texttt{fn}_Y\ \texttt{y => y})$.
- $\texttt{fun}_F\ \texttt{f x => if x} = 0\ \texttt{then 1 else x} * \texttt{f}(\texttt{x} - 1)$

Einfaches Typsystem

Typen: $\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2$
Typkontexte Γ: Endliche Abbildungen von Variablen auf Typen.

$$\frac{}{\Gamma \vdash c : \tau^c} \quad (\textsc{Con})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (\textsc{Var})$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e_0 : \tau_2}{\Gamma \vdash \texttt{fn}_\pi\ x \texttt{=>}\ e_0 : \tau_1 \to \tau_2} \quad (\textsc{Fn})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad (\textsc{App})$$

Beispiel: $\vdash (\texttt{fn}_X\ \texttt{x => x}))(\texttt{fn}_Y\ \texttt{y => y}) : \tau \to \tau$ für jedes τ.

Einfaches Typsystem, Forts.

$$\frac{\Gamma[f \mapsto \tau_1 \to \tau_2][x \mapsto \tau_1] \vdash e_0 : \tau_2}{\Gamma \vdash \texttt{fun}_\pi\ f\ x \texttt{=>}\ e_0 : \tau_1 \to \tau_2} \quad (\textsc{Fun})$$

$$\frac{\Gamma \vdash e_0 : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e_0 \texttt{then}\ e_1 \texttt{else}\ e_2 : \tau} \quad (\textsc{If})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2} \quad (\textsc{Let})$$

$$\frac{\Gamma \vdash e_1 : \tau_1^{op} \qquad \Gamma \vdash e_2 : \tau_2^{op}}{\Gamma \vdash e_1\ op\ e_2 : \tau_3^{op}} \quad (\textsc{Op})$$

Beispiel:

$\texttt{fun}_F\ \texttt{f}\ \texttt{x}\ \texttt{=>}\ \texttt{if}\ \texttt{x} = 0\ \texttt{then}\ 1\ \texttt{else}\ \texttt{x} * \texttt{f}(\texttt{x} - 1) : \texttt{int} \to \texttt{int}$, falls

$\tau_1^* = \tau_2^* = \tau_3^* = \tau_1^- = \tau_2^- = \tau_3^- = \tau^0 = \tau^1 = \texttt{int}$.

Typinferenz

Wir erweitern die Typen um Typvariablen:

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

Die Funktion $\mathcal{W}$ nimmt einen erweiterten Typkontext Γ und einen Term e und liefert entweder "fail" zurück, oder ein Paar (θ, τ) bestehend aus einer Einsetzung (Substitution) θ von erweiterten Typen für die Typvariablen in Γ und einem erweiterten Typ τ, sodass gilt:

- Falls $\mathcal{W}(\Gamma, e) = (\theta, \tau)$, so gilt
  - $\Gamma_0 \vdash e : \tau_0$ für alle durch Einsetzung von konkreten Typen für Variablen in $\Gamma[\theta]$ und τ entstehende Γ_0 und τ_0.
  - Ist $\Gamma_0 \vdash e : \tau_0$ und entsteht Γ_0 durch Einsetzung aus Γ, so entstehen Γ_0 und τ_0 durch Einsetzung von konkreten Typen für Variablen in $\Gamma[\theta]$ und τ (also wie im ersten Spiegelstrich).
- Falls $\mathcal{W}(\Gamma, e) =$ "fail", dann gibt es keine Typisierung $\Gamma_0 \vdash e : \tau_0$, sodass Γ_0 durch Einsetzung aus Γ hervorgeht.

Beispiele

- $\mathcal{W}(\varnothing, \texttt{fn}_X\ \texttt{x => x}) = (\varnothing, \alpha \to \alpha)$
- $\mathcal{W}([\texttt{x}\mapsto\texttt{bool}], \texttt{x} - 1) =$ “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”

Definition von $\mathcal{W}$

$$\frac{}{\mathcal{W}(\Gamma, c) = (\varnothing, \tau^c)} \quad (\textsc{Con})$$

$$\frac{}{\mathcal{W}(\Gamma, x) = (\varnothing, \Gamma(x))} \quad (\textsc{Var})$$

$$\frac{\mathcal{W}(\Gamma[x \mapsto \alpha], e_0) = (\theta, \tau) \qquad \alpha \text{ frisch}}{\mathcal{W}(\Gamma, \texttt{fn}_\pi\ x \texttt{=>}\ e_0) = (\theta, \alpha[\theta] \to \tau)} \quad (\textsc{Fn})$$

$$\frac{\mathcal{W}(\Gamma[f \mapsto \alpha_1 \to \alpha_2][x \mapsto \alpha_1], e_0) = (\theta, \tau_2) \qquad \theta_1 \text{ sodass } \alpha_2[\theta][\theta_1] = \tau_2[\theta_1]}{\mathcal{W}(\Gamma, \texttt{fun}\ f\ x \texttt{=>}\ e_0) = (\theta;\theta_1, \alpha_1[\theta][\theta_1] \to \tau_2[\theta_1])} \quad (\textsc{Fun})$$

In Regel $\textsc{Fun}$ ist θ_1 so allgemein wie möglich zu wählen ("allgemeinster Unifikator").

Wir definieren $\mathcal{W}(\Gamma, e) = (\theta, \tau)$, wenn das mit mit diesen Regeln (und den auf der nächsten Folie) herleitbar ist.

Typinferenz, Forts.

$$\frac{\begin{array}{c}\mathcal{W}(\Gamma, e_0) = (\theta_0, \tau_0) \\ \mathcal{W}(\Gamma[\theta_0], e_1) = (\theta_1, \tau_1) \\ \mathcal{W}(\Gamma[\theta_0][\theta_1], e_2) = (\theta_2, \tau_2)\end{array}}{\mathcal{W}(\Gamma, \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2) = (\theta_0;\theta_1;\theta_2;\theta, \tau)} \quad (\textsc{If})$$

wobei θ allgemeinstmöglich so zu wählen ist, dass $\tau_0[\theta_1][\theta_2]\theta = \texttt{bool}$ und $\tau_1[\theta_2][\theta] = \tau_2[\theta]$.

$\theta_1;\theta_2$ bezeichnet die Hintereinanderausführung von θ_1 und θ_2.

Die Regeln für $\textsc{App}$, $\textsc{Let}$ und $\textsc{Op}$ verbleiben als Übung.

Allgemeinstmögliche Wahl

Für erweiterte Typen (mit Variablen) τ_1, τ_2 bezeichne $\mathcal{U}(\tau_1, \tau_2)$ die allgemeinstmögliche Einsetzung θ von erweiterten Typen für die Variablen, so dass $\tau_1[\theta] = \tau_2[\theta]$. Wenn es keine solche gibt, dann ist $\mathcal{U}(\tau_1, \tau_2) =$ "fail".

Beispiele

- $\mathcal{U}(\alpha, \tau) = [\alpha \mapsto \tau]$
- $\mathcal{U}(\texttt{int}, \texttt{bool}) =$ "fail"
- $\mathcal{U}(\alpha \to \beta, (\texttt{bool} \to \beta) \to \texttt{int} \to \gamma) =$
  $[\alpha \mapsto \texttt{bool} \to \texttt{int} \to \gamma, \beta \mapsto \texttt{int} \to \gamma]$

$\theta_1;\theta_2$ bezeichnet die Hintereinanderausführung von θ_1 und θ_2:
$[\alpha \mapsto \beta \to \texttt{bool}]; [\beta \mapsto \texttt{int}] = [\alpha \mapsto \texttt{int} \to \texttt{bool}, \beta \mapsto \texttt{int}]$.
Formal kann $\mathcal{U}$ durch Rekursion über die Typsyntax definiert werden.(wird hier nicht behandelt)

Formale Spezifikation des allgemeinsten Unifikators

Formal spezifiziert man $\mathcal{U}$ wie folgt:

- $\mathcal{U}(\tau_1, \tau_2) =$ "fail" genau dann, wenn keine Substitution θ existiert mit $\tau_1[\theta] = \tau_2[\theta]$.
- Falls aber $\mathcal{U}(\tau_1, \tau_2) = \theta_0$, so muss gelten:
  - $\tau_1[\theta_0] = \tau_2[\theta_0]$
  - Wenn immer $\tau_1[\theta] = \tau_2[\theta]$, so existiert θ_1 mit $\theta = \theta_0; \theta_1$. Man erhält also θ durch Spezialisierung von θ_0.

Typschemata

Bisher konnten Funktionen nur konkrete Typen haben, wie z.B. $\texttt{fn}_X\ x \texttt{ => } x : \texttt{int} \to \texttt{int}$.
Wir erweitern jetzt das Typsystem selbst um Typvariablen und Quantifikation, sodass z.B.: $\texttt{fn}_X\ \texttt{x => x} : \forall\alpha.\alpha \to \alpha$.

SML Notation

In SML *schreibt man* $\alpha \to \alpha$ oder $\texttt{'a} \to \texttt{'a}$, *meint aber* $\forall\alpha.\alpha \to \alpha$.

Grammatik für Typen (τ) und Typschemata (σ)

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

$$\sigma ::= \forall(\alpha_1, \ldots, \alpha_n).\tau$$

Abkürzungen und Beispiele

- 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.
- Beispiel: $\forall(\alpha, \beta, \gamma).(\alpha \to \beta \to \gamma) \to (\beta \to \alpha \to \gamma)$.
- 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:

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

Man kann *nicht* schreiben:

$$(\texttt{fn}_F\ \texttt{f => if f true then}\ f\ \texttt{0 else f 1})(\texttt{fn}_X\ \texttt{x=> x})$$

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 binden nunmehr Variablen auf Typschemata ab.
Das Typurteil hat das Format $\Gamma \vdash e : \sigma$.
Die Typregeln Con, Var, Fn, Fun, App, Fun, Op, If, App werden unverändert übernommen mit der gemachten Übereinkunft, dass Typen spezielle Typschemata sind. So ist zum Beispiel die Regel

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e_0 : \tau_2}{\Gamma \vdash \texttt{fn}_\pi\ x \texttt{ => } e_0 : \tau_1 \to \tau_2} \quad (\text{Fn})$$

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

Polymorphes Typsystem, Generalisierungsregel

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha_1, \ldots, \alpha_n \text{ kommen nicht frei in } \Gamma \text{ vor}}{\Gamma \vdash e : \forall(\alpha_1, \ldots, \alpha_n).\sigma}\ (\textsc{Gen})$$

α kommt frei vor in $\alpha \to \alpha$ oder in $[x \mapsto \alpha]$, o.ä.

α kommt nicht frei (sondern nur gebunden) vor in $\forall\alpha.\alpha \to \alpha$ oder $[x \mapsto \forall\alpha.\alpha \to \alpha]$. Man könnte ja stattdessen auch $\forall\beta.\beta \to \beta$ oder $[x \mapsto \forall\beta.\beta \to \beta]$ schreiben.

Beispiel:

$$\dfrac{\dfrac{\texttt{x} \mapsto \alpha \vdash \texttt{x} : \alpha}{\varnothing \vdash \texttt{fn}_X\ \texttt{x => x} : \alpha \to \alpha}\ \textsc{Fn}}{\varnothing \vdash \texttt{fn}_X\ \texttt{x => x} : \forall\alpha.\alpha \to \alpha}\ \textsc{Gen}$$

Polymorphes Typsystem, Instanzierungsregel

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

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

Beispiel:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{f} \mapsto \forall\alpha.\alpha\to\alpha \vdash \mathtt{f} : \forall\alpha.\alpha \to \alpha}\,\textsc{Var}}{\mathtt{f} \mapsto \forall\alpha.\alpha\to\alpha \vdash \mathtt{f} : \mathtt{bool} \to \mathtt{bool}}\,\textsc{Inst} \quad \dfrac{\dfrac{}{\mathtt{f} \mapsto \forall\alpha.\alpha\to\alpha \vdash \mathtt{f} : \forall\alpha.\alpha \to \alpha}\,\textsc{Var}}{\mathtt{f} \mapsto \forall\alpha.\alpha\to\alpha \vdash \mathtt{f} : \mathtt{int} \to \mathtt{int}}\,\textsc{Inst}}{\mathtt{f} \mapsto \forall\alpha.\alpha\to\alpha \vdash \mathtt{f\ true} : \mathtt{bool}}\,\textsc{App} \quad \dfrac{}{\mathtt{f} \mapsto \forall\alpha.\alpha\to\alpha \vdash \mathtt{f\ 0} : \mathtt{int}}\,\textsc{App}$$

$$\mathtt{f} \mapsto \forall\alpha.\alpha\to\alpha \vdash \mathtt{if\ f\ true\ then\ f\ 0\ else\ 1} : \mathtt{int}$$

If und Co

Polymorphes Typsystem, Let-regel

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

Beispiel:

$$\frac{\texttt{f} \mapsto \forall\alpha.\alpha \to \alpha \vdash \texttt{if f true then f 0 else 1} : \texttt{int} \qquad \emptyset \vdash \texttt{fn}_X \texttt{ x=> x} : \forall\alpha.\alpha \to \alpha}{\emptyset \vdash \texttt{let f = fn}_X \texttt{ x=> x in if f true then f 0 else 1} : \texttt{int}}$$

Typinferenz mit Polymorphie

- Die Funktion $\mathcal{W}(\Gamma, e)$ liefert wie vorher einen Typ (kein Typschema) und eine Substitution (für freie Typvariablen in Γ) zurück (oder "fail"!).
- Bei $\mathcal{W}(\Gamma, x)$ muss das Typschema $\Gamma(x)$ mit frischen Variablen instanziert werden.
- Ist also $\Gamma = [x \mapsto \forall\alpha.\alpha \to \alpha, y \mapsto \alpha]$, so ist $\mathcal{W}(\Gamma, x) = \beta \to \beta$.
- Bei $\mathcal{W}(\Gamma, \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2)$ muss der rekursiv ermittelte Typ von e_1 *geschlossen* werden.
- Ist also $\mathcal{W}(\Gamma, e_1) = (\theta, \tau)$, so tätigt man den Aufruf $\mathcal{W}(\Gamma[\theta][x \mapsto \forall\vec{\alpha}.\tau], e_2)$, wobei $\vec{\alpha}$ alle Typvariablen umfasst, die frei in τ vorkommen, aber gleichzeitig nicht frei in $\Gamma[\theta]$ vorkommen.

Kontrollflussanalyse

- Wir möchten für jede Funktion wissen, mit welchen `fn`- oder `fun`-Kontrukten sie erzeugt wurde.
- Dazu benutzen wir verfeinerte Funktionstypen der Form $\tau_1 \xrightarrow{\Pi} \tau_2$, wobei Π eine *Menge* von Labels ist.
- Beispiele:

$$\texttt{fn}_X\texttt{x=>x} : \texttt{int} \xrightarrow{\{X\}} \texttt{int}$$

$$\texttt{if } b \texttt{ then fn}_X\texttt{x=>x else fn}_Y\texttt{y=>y} + 1 : \texttt{int} \xrightarrow{\{X,Y\}} \texttt{int}$$

- (Polymorphie ist hier zunächst wieder "abgeschaltet".)
- Verallgemeinerte Anwendung: Herkunft von Datenstrukturen (hier Funktionen). Z.B.: Strings nur als Ergebnis von Sanitizing Funktionen (Abhilfe gegen XSS Attacken).

Typregeln für Kontrollflussanalyse

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

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e_0 : \tau_2}{\Gamma \vdash \texttt{fn}_\pi\ x \texttt{=>}\ e_0 : \tau_1 \xrightarrow{\{\pi\} \cup \Pi} \tau_2} \quad (\textsc{Fn})$$

$$\frac{\Gamma[f \mapsto \tau_1 \xrightarrow{\{\pi\} \cup \Pi} \tau_2][x \mapsto \tau_1] \vdash e_0 : \tau_2}{\Gamma \vdash \texttt{fun}_\pi\ f\ x \texttt{=>}\ e_0 : \tau_1 \xrightarrow{\{\pi\} \cup \Pi} \tau_2} \quad (\textsc{Fun})$$

Die anderen Regeln werden sinngemäß übernommen.

Approximation

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

Kontrollflussanalyse: Typinferenz

- Man verwendet *Mengenvariablen* als Annotierungen der Funktionstypen.
- Die Funktionen $\mathcal{W}$ und $\mathcal{U}$ liefern jeweils zusätzlich eine Liste von Constraints über den Mengenvariablen, sodass jede Lösung der Constraints eine Typisierung liefert und umgekehrt.

Beispiel: Typinferenz angewandt auf

$$(\texttt{fn}_X\ \texttt{x=>x})(\texttt{fn}_Y\ \texttt{y=>y})$$

liefert den verallgemeinerten Typ $\alpha \xrightarrow{\zeta} \alpha$ und die Constraintmenge $\{Y\} \subseteq \zeta$.

Operationelle Semantik

Wir definieren ein Auswerteurteil $e \longrightarrow v$, gelesen e wertet zu v aus, wobei e ein geschlossener Ausdruck ist und e ein *Wert* ist, also eine Konstante, oder ein `fn`-Ausdruck.

$$\frac{}{c \longrightarrow c} \quad (\textsc{Con})$$

$$\frac{}{\mathtt{fn}_\pi\ x\mathtt{=>}e \longrightarrow \mathtt{fn}_\pi\ x\mathtt{=>}e} \quad (\textsc{Fn})$$

$$\frac{}{\mathtt{fun}_\pi\ f\ x\mathtt{=>}e \longrightarrow \mathtt{fn}_\pi\ x\mathtt{=>}e\left[f \mapsto \mathtt{fun}_\pi\ f\ x\mathtt{=>}e\right]} \quad (\textsc{Fun})$$

$$\frac{e_1 \longrightarrow \mathtt{fn}_\pi\ x\mathtt{=>}\ e_1' \qquad e_2 \longrightarrow v_2 \qquad e_1'[x \mapsto v_2] \longrightarrow v}{e_1\ e_2 \longrightarrow v} \quad (\textsc{App})$$

$$\frac{e_1 \longrightarrow \mathtt{true} \qquad e_2 \longrightarrow v}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \longrightarrow v} \quad (\textsc{If1})$$

Operationelle Semantik, Forts.

$$\frac{e_1 \longrightarrow \texttt{false} \qquad e_3 \longrightarrow v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow v} \quad (\textsc{If2})$$

$$\frac{e_1 \longrightarrow v_1 \qquad e_2[x \mapsto v_1] \longrightarrow v}{\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2 \longrightarrow v} \quad (\textsc{Let})$$

$$\frac{e_1 \longrightarrow v_1 \qquad e_2 \longrightarrow v_2 \qquad v = v_1 \ \mathbf{op} \ v_2}{e_1 \ op \ e_2 \longrightarrow v} \quad (\textsc{Op})$$

- $e[x \mapsto v]$ bezeichnet die Einsetzung (Substitution) von v für x in e.
- v_1 **op** v_2 ist das entsprechende Ergebnis des Operators op. Z.B. 3+4 = 7. Wir verlangen, dass es den Typ τ_3^{op} besitzt.

Semantische Typkorrektheit

Korrektheit des einfachen Typsystems

Wenn $\varnothing \vdash e : \tau$ und $e \longrightarrow v$, dann auch $\varnothing \vdash v : \tau$. Außerdem bleibt die Auswertung von e nicht stecken, in dem Sinne, dass keine Auswertungsregel anwendbar wäre. Z.B. bleibt 0 1 (0 angewendet auf 1) stecken.

Man kann das "Steckenbleiben" sauber definieren, indem man einen Sonderwert *wrong* hinzugibt und Auswerteregeln wie etwa $e_1\ e_2 \longrightarrow \mathit{wrong}$, falls $e \longrightarrow v$ und v kein `fn`-Ausdruck ist.

Korrektheit der Kontrollflussanalyse

Wenn $\varnothing \vdash e : \tau$ und $e \longrightarrow v$, dann auch $\varnothing \vdash v : \tau$. Insbesondere gilt:

Wenn $\varnothing \vdash e : \tau_1 \xrightarrow{\Pi} \tau_2$ und $e \longrightarrow \mathtt{fn}_\pi x \mathtt{=>} e_1$, dann $\pi \in \Pi$.

Man beweist die Korrektheit jeweils durch Induktion über $e \longrightarrow v$.

Erweiterung der Sprache um Referenzen

$$e ::= \cdots \mid \texttt{ref}_\pi\ e \mid \texttt{!}e \mid e_1 := e_2$$

Abkürzung:
$e_1; e_2$ steht für $\texttt{let}\ \_ = e_1 \texttt{in}\ e_2$ wobei $\_$ eine frische Variable ist.

Programm für Fibonaccizahlen:

$$\begin{aligned}&\texttt{fn}_F\ \texttt{m=>let o = ref}_O\ 0\ \texttt{in let n = ref}_N\ 1\ \texttt{in}\\ &\quad(\texttt{fun}_G\ \texttt{f m=>if m} \leqslant 1\ \texttt{then 44 else}\\ &\qquad \texttt{f}(\texttt{m} - 1); \texttt{let x=!o in o:=!n; n:=!n} + \texttt{x})\ \texttt{m};\\ &\quad \texttt{if m} \leqslant 1\ \texttt{then m else !o}\end{aligned}$$

Seiteneffektanalyse

Effektannotationen

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

- $!\pi$ (Lesen einer bei π erzeugten Referenz)
- $\pi :=$ (Zuweisen zu einer bei π erzeugten Referenz)
- $\texttt{ref}\pi$ (Erzeugen einer Referenz bei π)

Zum Beispiel ist $\{O :=, !O, !N\}$ ein Effekt.

Verfeinerte Typen:

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

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

Seiteneffektanalyse als Typsystem

Das Typurteil hat nunmehr die Form $\Gamma \vdash e : \tau \,\&\, \varpi$.

$$\frac{}{\Gamma \vdash c : \tau^c \,\&\, \varnothing} \quad (\textsc{Con})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \,\&\, \varnothing} \quad (\textsc{Var})$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e_0 : \tau_2 \,\&\, \varpi}{\Gamma \vdash \mathtt{fn}_\pi\ x \texttt{=>}\ e_0 : \tau_1 \xrightarrow{\varpi} \tau_2 \,\&\, \varnothing} \quad (\textsc{Fn})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varpi_1} \tau_2 \,\&\, \varpi_2 \qquad \Gamma \vdash e_2 : \tau_1 \,\&\, \varpi_3}{\Gamma \vdash e_1\ e_2 : \tau_2 \,\&\, \varpi_1 \cup \varpi_2 \cup \varpi_3} \quad (\textsc{App})$$

Fortsetzung

$$\frac{\Gamma[f \mapsto \tau_1 \xrightarrow{\varpi} \tau_2][x \mapsto \tau_1] \vdash e_0 : \tau_2 \,\&\, \varpi}{\Gamma \vdash \texttt{fun}_\pi\ f\ x \texttt{=>}\ e_0 : \tau_1 \xrightarrow{\varpi} \tau_2 \,\&\, \varpi} \quad (\textsc{Fun})$$

$$\frac{\Gamma \vdash e_0 : \texttt{bool} \,\&\, \varpi \qquad \Gamma \vdash e_1 : \tau \,\&\, \varpi \qquad \Gamma \vdash e_2 : \tau \,\&\, \varpi}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \,\&\, \varpi} \quad (\textsc{If})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \,\&\, \varpi_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2 \,\&\, \varpi}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \,\&\, \varpi_1 \cup \varpi} \quad (\textsc{Let})$$

$$\frac{\Gamma \vdash e_1 : \tau_1^{op} \,\&\, \varpi \qquad \Gamma \vdash e_2 : \tau_2^{op} \,\&\, \varpi}{\Gamma \vdash e_1\ op\ e_2 : \tau_3^{op} \,\&\, \varpi} \quad (\textsc{Op})$$

Fortsetzung

$$\frac{\Gamma \vdash e : \mathtt{ref}_\Pi \tau \;\&\; \varpi}{\Gamma \vdash\; !e : \tau \;\&\; \varpi \cup\; !\Pi} \quad (\textsc{Deref})$$

$$\frac{\Gamma \vdash e_1 : \mathtt{ref}_\Pi \tau \;\&\; \varpi \qquad \Gamma \vdash e_2 : \tau \;\&\; \varpi}{\Gamma \vdash e_1 := e_2 : \tau \;\&\; \varpi \cup \Pi :=} \quad (\textsc{Ass})$$

$$\frac{\Gamma \vdash e : \tau \;\&\; \varpi}{\Gamma \vdash \mathtt{ref}_\pi e : \mathtt{ref}_{\{\pi\}} \tau \;\&\; \varpi \cup \{\mathtt{ref}\pi\}} \quad (\textsc{Ref})$$

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

Hier bedeuten $!\Pi = \{!\pi \mid \pi \in \Pi\}$ und $\Pi := = \{\pi := \mid \pi \in \Pi\}$.

Subtyping

- Die Regel SUB erlaubt die Anpassung von Typen und Effekten, sodass z.B. beide Zweige einer Fallunterscheidung denselben Typ und Effekt haben.
- In der Kontrollflussanalyse wurde diese Anpassung in die Regel VAR eingearbeitet. Das würde hier zu aufwendig.
- Das Subtyping-Urteil $\tau_1 \leqslant \tau_2$ ist definiert durch:

$$\frac{\Pi \subseteq \Pi'}{\mathtt{ref}_{\Pi} \leqslant \mathtt{ref}_{\Pi'}} \qquad \frac{\tau_2 \leqslant \tau_1 \quad \tau_1' \leqslant \tau_2' \quad \varpi \subseteq \varpi'}{\tau_1 \xrightarrow{\varpi} \tau_1' \leqslant \tau_2 \xrightarrow{\varpi'} \tau_2'}$$

Beispiele

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

Die beiden Argumente für `f` können wie folgt analysiert werden:

$$\texttt{int} \xrightarrow{\{!A\}} \texttt{int} \,\&\, \varnothing$$
$$\texttt{int} \xrightarrow{\{A:=\}} \texttt{int} \,\&\, \varnothing$$

Daher gibt man `f` den Typ

$$(\texttt{int} \xrightarrow{\{!A,A:=\}} \texttt{int}) \xrightarrow{\{!A,A:=\}} \texttt{int}$$

Die Typen der Argumente werden dann durch Subtyping angepasst.

Exceptions

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

$$e ::= \cdots \mid \texttt{raise}\ s \mid \texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2$$

Hier rangiert s über eine festgewählte Menge von Exceptions.

Beispiel:

```
let comb = fun f x => fn y =>
  if x < 0 then raise y-out-of-range
  else if y < 0 or y > x then raise y-out-of-range
    else if y = 0 or y = x then 1
      else f (x − 1) y + f (x − 1) (y − 1)
in handle x-out-of-range as 0 in comb x y
```

Operationelle Semantik

Die bisherigen Auswerteregeln bleiben in Kraft, gelten aber nicht für den Fall, dass ein Teilausdruck zu `raise` s für ein s auswertet. Hinzu kommen:

$$\frac{e_1 \longrightarrow \texttt{raise}\ s}{e_1\ e_2 \longrightarrow \texttt{raise}\ s}$$

$$\frac{e_1 \longrightarrow \texttt{fn x=>}e_1' \qquad e_2 \longrightarrow \texttt{raise}\ s}{e_1\ e_2 \longrightarrow \texttt{raise}\ s}$$

$$\frac{e_1 \longrightarrow \texttt{fn}\ x\texttt{=>}\ e_1' \qquad e_2 \longrightarrow v_2 \qquad e_1'[x \mapsto v_2] \longrightarrow \texttt{raise}\ s}{e_1\ e_2 \longrightarrow \texttt{raise}\ s}$$

Analoge Regeln für `fun`, `let`, op.

Operationelle Semantik, Forts.

$$\frac{}{\texttt{raise}\ s \longrightarrow \texttt{raise}\ s}$$

$$\frac{e_2 \longrightarrow v_2 \qquad v_2 \neq \texttt{raise}\ s}{\texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2 \longrightarrow v_2}$$

$$\frac{e_2 \longrightarrow \texttt{raise}\ s \qquad e_1 \longrightarrow v_1}{\texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2 \longrightarrow v_1}$$

Annotierte Typen mit Effektpolymorphie

Effektausdrücke:

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

Hierbei bezeichnet s den Effekt des möglichen Auslösens der Exception s und ϵ rangiert über *Effektvariablen*.

Annotierte Typen:

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

α rangiert wie bisher über Typvariablen.

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 an *Typschemata* bindet.

$$\frac{}{\Gamma \vdash c : \tau^c \,\&\, \varnothing} \quad (\textsc{Con})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \,\&\, \varnothing} \quad (\textsc{Var})$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e_0 : \tau_2 \,\&\, \varphi}{\Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e_0 : \tau_1 \xrightarrow{\varphi} \tau_2 \,\&\, \varnothing} \quad (\textsc{Fn})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2 \,\&\, \varphi_2 \qquad \Gamma \vdash e_2 : \tau_1 \,\&\, \varphi_3}{\Gamma \vdash e_1\ e_2 : \tau_2 \,\&\, \varphi_1 \cup \varphi_2 \cup \varphi_3} \quad (\textsc{App})$$

Fortsetzung

$$\frac{\Gamma[f \mapsto \tau_1 \xrightarrow{\varphi} \tau_2][x \mapsto \tau_1] \vdash e_0 : \tau_2 \,\&\, \varphi}{\Gamma \vdash \texttt{fun}\ f\ x\texttt{=>}\ e_0 : \tau_1 \xrightarrow{\varphi} \tau_2 \,\&\, \varnothing} \quad (\textsc{Fun})$$

$$\frac{\Gamma \vdash e_0 : \texttt{bool} \,\&\, \varphi \qquad \Gamma \vdash e_1 : \tau \,\&\, \varphi \qquad \Gamma \vdash e_2 : \tau \,\&\, \varphi}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \,\&\, \varphi} \quad (\textsc{If})$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \,\&\, \varphi_1 \qquad \Gamma[x \mapsto \sigma_1] \vdash e_2 : \tau_2 \,\&\, \varphi_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2 \,\&\, \varphi_1 \cup \varphi_2} \quad (\textsc{Let})$$

$$\frac{\Gamma \vdash e_1 : \tau_1^{op} \,\&\, \varphi \qquad \Gamma \vdash e_2 : \tau_2^{op} \,\&\, \varphi}{\Gamma \vdash e_1\ op\ e_2 : \tau_3^{op} \,\&\, \varphi} \quad (\textsc{Op})$$

Fortsetzung

$$\frac{\Gamma \vdash e : \tau \,\&\, \varphi \qquad \zeta_1, \ldots, \zeta_n \text{ kommen nicht frei in } \Gamma \text{ und } \varphi \text{ vor}}{\Gamma \vdash e : \forall(\zeta_1, \ldots, \zeta_n).\tau \,\&\, \varphi} \quad (\textsc{Gen})$$

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

Hier ist θ eine Einetzung von Typen für Typvariablen und Effektausdrücken für Effektvariablen.

Fortsetzung

$$\frac{}{\Gamma \vdash \texttt{raise}\ s : \forall\alpha.\alpha \,\&\, \{s\}} \quad (\textsc{Raise})$$

$$\frac{\Gamma \vdash e_1 : \tau \,\&\, \varphi \qquad \Gamma \vdash e_2 : \tau \,\&\, \varphi \cup \{s\}}{\Gamma \vdash \texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2 : \tau \,\&\, \varphi} \quad (\textsc{Handle})$$

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

Das *Subeffecting*-Urteil $\varphi_1 \leqslant \varphi_2$ versteht sich als punktweise Teilmengenbeziehung.
Subtyping ist wie bisher (Folie 232) definiert.

Beispiel

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

Dieses Programm wertet zu 2 aus.
Der Funktion `f` kann das folgende Typschema zugewiesen werden:

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

Die Argumentfunktionen für `f` erhalten die Typen $(\texttt{int} \xrightarrow{\{\texttt{neg}\}} \texttt{int})$ und $(\texttt{int} \xrightarrow{\{\texttt{pos}\}} \texttt{int})$.

Typinferenz

Um Typen im Exception-Typsystem automatisch zu inferieren geht man zunächst so vor wie bei der polymorphen Typinferenz für normale Typen.

Die $\mathcal{U}$-Funktion erzeugt dann ein System von Constraints, welche auch universell quantifizierte Effektvariablen enthalten. Diese Constraints lassen sich mit verschiedenen Methoden, wie Iteration oder auch SAT-Solver lösen. Die Details werden hier nicht präsentiert.

Polymorphe Rekursion

Erlaubt man Typschemata (statt nur einfacher Typen) für die mit `fun` gebundene Funktion so spricht man von *polymorpher Rekursion*. Für Typschemata mit Typvariablen ist dann die Typinferenz unentscheidbar. Für Typschemata, die nur Effektvariablen quantifizieren lässt sich die Inferenz aber durchführen.

Das ist sinnvoll, wenn zur Rechtfertigung einer Instanz des Typschemas eine andere Instanz im rekursiven Aufruf benötigt wird.

Weiterhin kann man auch über Constraints quantifizieren, also z.B. $\forall \epsilon \geqslant \{s\} \ldots$ und außerdem Kontrollflussinformation zur Findung passender Instanzierungen bei $\textsc{Inst}$ heranziehen.

Zusammenfassung Typsysteme für funktionale Programme

- Typ- und Effektsysteme verallgemeinern das SML Typsystem durch Effektannotationen für Ausdrücke und Funktionen.
- Sie erlauben die strukturierte Analyse von Programmen mit Funktions- (Prozedur-, Methoden-)Variablen.
- Zur Lösung von Constraintsystemen, die bei der Typinferenz anfallen, werden verschiedene Lösungsalgorithmen herangezogen.
- Man unterscheidet einfache und polymorphe Typsysteme, sowie Typsysteme mit polymorpher Rekursion.

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:

$$\text{Gilt } \sigma \in [\![P]\!] \text{ und } \langle S, \sigma \rangle \to^* \sigma' \text{ so auch } \sigma' \in [\![Q]\!].$$

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

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

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

Hoare-Regeln

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

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

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

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

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

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

Abschwächungsregel

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

Beispiele:

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

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

Bezugnahme auf Startwerte

Um auch ausdrücken zu können, dass bestimmte Programmvariablen nicht verändert werden, erlauben wir die Bezugnahme auf Startwerte der Variablen *vor Ausführung des Gesamtprogramms*, also nicht nur vor Ausführung von S.

Diese werden vom Programmstück nicht verändert.

Allgemeiner kann man beliebige Hilfsvariablen zulassen, die vom Programmstück ebenfalls nicht verändert werden.

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

Für eine Zusicherung P besteht $[\![P]\!]$ dann aus allen Paaren (σ_{init}, σ), wobei σ_{init} die Anfangswerte der Variablen enthält und σ wie vorher die aktuellen Werte der Variablen.

Korrektheit der Hoare Logik

Theorem

Sei $\langle S, \sigma \rangle \to^* \sigma'$ und σ_{init} ein beliebger Zustand und $\{P\}S\{Q\}$ herleitbar. Falls $(\sigma_{init}, \sigma) \in [\![P]\!]$, so auch $(\sigma_{init}, \sigma') \in [\![Q]\!]$.

Dies beweist man durch simultane Induktion über die Anzahl der Reduktionsschritte (erste Priorität) sowie die Herleitung von $\langle S, \sigma \rangle \to X$ (zweite Priorität).

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

$$[\![WP(S, Q)]\!] = \{(\sigma_{init}, \sigma) \mid \forall \sigma'. \langle S, \sigma\rangle \rightarrow^* \sigma' \Rightarrow (\sigma_{init}, \sigma') \in [\![Q]\!]\}.$$

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 z.B. auf Folie 196.

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

Der zugrundeliegende Verband besteht aus allen Prädikaten auf Paaren von Programmzuständen, also $\mathcal{P}(states^2)$.
Dieser Verband ist vollständig, hat aber keine endliche Höhe (Gegenbeispiel: $P_i = \{(\sigma_{init}, \sigma) \mid \sigma(x) \in \mathbb{N}, \sigma(x) \leqslant i\}$ und $\bigcup_i P_i = \{(\sigma_{init}, \sigma) \mid \sigma(x) \in \mathbb{N}\}$).

Datenfluss

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

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

Ausnahme: Für das Ausgangslabel ℓ_e darf $HL_{exit}(\ell_e)$ beliebig sein.

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.

Datenfluss

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

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

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

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

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 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).

Datenflussgleichungen

Für jede beliebige Wahl der Menge $HL_{exit}(\ell_e)$ für das Ausgangslabel ℓ_e ist das folgende Gleichungssystem lösbar.

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

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

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

Hoare-Regeln für Prozeduren

- Wir interessieren uns jetzt für eine Erweiterung des Hoare-Kalküls auf (der Einfachheit halber) parameterlose Prozeduren.
- Wir nehmen an, dass es eine Reihe solcher Prozeduren gibt und dass der Rumpf der Prozedur f mit S_f bezeichnet ist.
- Man könnte einfach die Hoare-Regeln darauf erweitern, indem man (wie bei den Typsystemen) einen Typkontext mit Annahmen $\{P\}f\{Q\}$ einführt und dann die folgenden zusätzlichen Regeln einführt:

Einfache Regeln für Prozeduren

$$\frac{}{\Gamma, \{P\}f\{Q\} \vdash \{P\}f()\{Q\}} \quad \text{(H-App')}$$

$$\frac{\Gamma, \{P\}f\{Q\} \vdash \{P\}S_f\{Q\} \text{ und } \Gamma, \{P\}f\{Q\} \vdash \mathcal{J}}{\Gamma \vdash \mathcal{J}} \quad \text{(H-Fun')}$$

In Regel Fun' steht $\mathcal{J}$ für ein beliebiges Hoare-Tripel.

Dies liefert ein korrektes System, allerdings gibt es Probleme mit der Vollständigkeit.

Man kann $\{x = x_{init}\}\ f()\ \{x = x_{init}\}$ nicht herleiten, wenn der Rumpf S_f zum Beispiel $x{:=}x - 1;\ f();\ x{:=}x + 1$ ist.

Spezifikationen als Relationen

Besser ist es Spezifikationen von Prozeduren der Form $f : R$ im Typkontext vorzuhalten, wobei R eine Relation auf Zuständen ist und die Bedeutung von $f : R$ wie folgt ist:

$$\text{Wenn } \langle f(), \sigma \rangle \to^* \sigma', \text{ dann } R(\sigma, \sigma').$$

Die Regeln werden wie folgt angepasst:

$$\frac{\forall \sigma_{init}, \sigma. (\sigma_{init}, \sigma) \in [\![P]\!] \wedge R(\sigma, \sigma') \Rightarrow (\sigma_{init}, \sigma') \in [\![Q]\!]}{\Gamma, f : R \vdash \{P\} f() \{Q\}} \text{ (H-App)}$$

$$\frac{\Gamma, f : R \vdash \{Pre_R\}\, S_f\, \{Post_R\} \text{ und } \Gamma, f : R \vdash \mathcal{J}}{\Gamma \vdash \mathcal{J}} \text{ (H-Fun)}$$

wobei

$$[\![Pre_R]\!] = \{(\sigma_{init}, \sigma) \mid \sigma_{init} = \sigma\}$$
$$[\![Post_R]\!] = \{(\sigma_{init}, \sigma) \mid R(\sigma_{init}, \sigma)\}$$

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.
- Durch die Auffassung als Typsystem ergibt sich eine natürliche Verallgemeinerung der Hoare-Regeln auf rekursive Prozeduren.

Ausblick: Java Modelling Language

Die Java Modelling Language (JML) ist eine Sprache zur Spezifikation von Eigenschaften von Java-Programmen.

Spezifikation im Stil der Hoare Logik:

- Vorbedingungen
- Nachbedingungen
- Invarianten
- Zusicherungen (Assertions)

JML kann als Ausprägung des **Design by Contract** angesehen werden.

Design by Contract

Analogie zu Verträgen im Geschäftsleben:

- Jede Programmkomponente (Methode, Klasse) bietet einen Vertrag an.
- Ein Vertrag beinhaltet das Versprechen der Programmkomponente, eine bestimmte Leistung zu erbringen, wenn bestimmte Voraussetzungen erfüllt sind.
- Zusammensetzung von Komponenten nur nach Vertrag: Das Gesamtprogramm hat gewünschte Eigenschaften, wenn nur alle Komponenten ihre Verträge einhalten.

Verträge

Verträge werden in der Dokumentation angegeben (informell oder formal, z.B. in JML-Syntax)

Vertrag einer Klasse:

- eine Vertragsklausel für jede Methode
- Klasseninvarianten: Die Klasse garantiert, dass ihre Objekte stets eine bestimmte Eigenschaft haben.

Vertrag einer Methode:

- Vorbedingung
- Nachbedingung
- Effektbeschreibung

Beispiele (Methoden)

Informell:

- Javadoc von `equals` in `java.lang.Object`
- Javadoc von `hashCode` in `java.lang.Object`

JML-Syntax:

```
/*@  requires betrag > 0;
  @  ensures (kontoStand == \old(kontoStand) + betrag) &&
  @          \result == kontoStand;
  @*/
 public int einzahlen(int betrag) {
    ...
```

Klasseninvarianten

- Oft sollen Instanzvariablen zu jedem Zeitpunkt bestimmte Eigenschaften haben. Beispiel: Instanzvariable `!= null`
- Anstatt sie in allen Vor- und Nachbedingungen aufzuführen formuliert man sie als **Klasseninvariante**.
- Eine Invariante wird nur einmal für die Klasse formuliert, muss jedoch in allen Methoden und Konstruktoren beachtet werden.
  - Jeder Konstruktor muss die Invariante als Nachbedingung herstellen.
  - Jede Methode darf die Invariante als Vorbedingung annehmen.
  - Jede Methode muss die Invariante als Nachbedingung sicherstellen.

Beispiele (Klasseninvarianten)

```
public class Konto {
  private Person inhaber;
  /*@ invariant inhaber != null @*/
  private int kontoStand;
  /*@ invariant kontoStand > -MAX_DISPO @*/
  ...
}

public class LinkedList {
  private Node head;
  private int length;
  /*@ invariant length == head.length() @*/
  // head.length() berechnet Länge in Zeit O(n).
  ...
}
```

Beispiele (Klasseninvarianten)

```
public class Bank {

  // redundante Datenhaltung aus Effizienzgründen
  private Map<Person, Set<Konto>> konten;
  private Map<Konto, Person> inhaber;

  /* Invariante:
   *  Für alle (Person p) und alle (Konto k) gilt:
   *    konten.get(p).contains(k) <==>
   *                inhaber.get(k).equals(p)
   */
}
```

Verträge und Vererbung

Eine Unterklasse ist an den Vertrag der Oberklasse gebunden.

Jede überschreibende Methode muss den Vertrag der überschriebenen Methode einhalten:

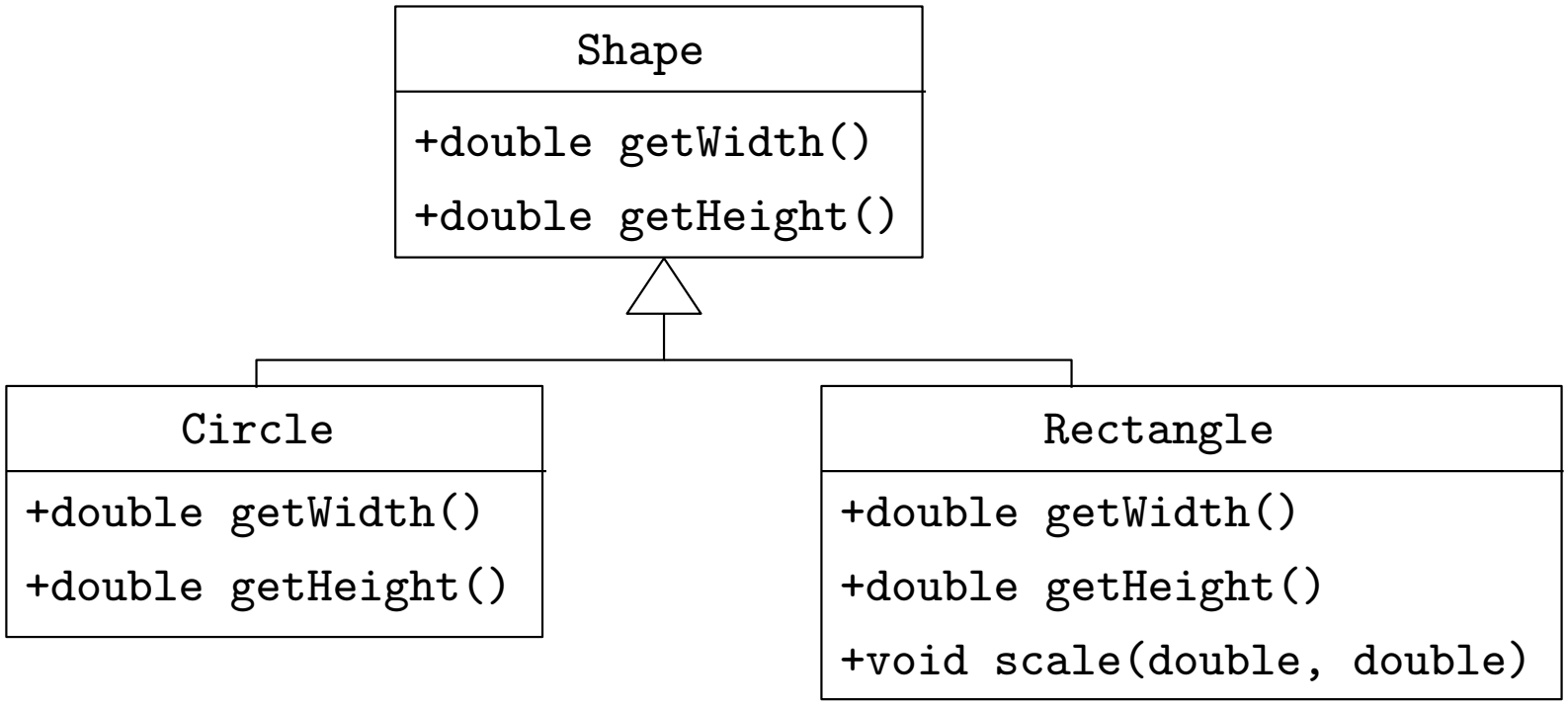
- Vorbedingungen können höchstens abgeschwächt werden.
- Nachbedingungen können höchstens verstärkt werden.

In der Unterklasse müssen alle Invarianten der Oberklasse erhalten bleiben.

- Konstruktoren der Unterklasse müssen die Invarianten herstellen.
- Überschriebene Methoden müssen die Invariante erhalten.

Verträge und Vererbung

Beispiel: Wir wollen folgende Klassenhierarchie um eine Klasse `Square` für Quadrate erweitern.



Sollen `Square` als Unterklasse von `Rectangle` modelliert werden?

Verträge und Vererbung

Vertrag von `void scale(double sx, double sy)`:

- Vorbedingung: `sx` und `sy` beide nichtnegativ.
- Nachbedingung: Es gilt `getWidth() == sx * w`, wobei `w` das Ergebnis von `getWidth()` vor dem Aufruf von `scale` ist. (und analog für die Höhe)

Die Klasse `Square` kann diesen Vertrag nicht erfüllen.
`Square` kann nicht als Unterklasse von `Rectangle` eingeordnet werden, sondern von `Shape`.

Überprüfung von Verträgen

In der objektorientierten Programmierung werden Verträge oft nur informell angegeben und überprüft. DBC wird dann nur als Entwicklungsmethodologie verwendent.

Beispiel: Dokumentation der Methoden `equals` und `hashCode` in `java.lang.Object`.

Es ist wünschenwert die Einhaltung von Verträgen zu überprüfen, um Programmierfehler durch Vertragsverletzungen zu erkennen.

Möglichkeiten:

- stichprobenartige Überprüfung, z.B. durch Unit-Tests
- Überprüfung durch Zusicherungen zur Programmlaufzeit
- allgemeine Überprüfung durch statische Verifikation

Überprüfung durch Zusicherungen

Java stellt eine Zusicherungsanweisung zur Überprüfung von Verträgen bereit:

```
assert test : errorValue;
```

- Verhält sich wie

  ```
  if (!test) { throw new AssertionError(errorValue); }
  ```

- Standardmäßig werden Assertions ignoriert, d.h. sie haben keinen Einfluss auf den Progammablauf oder die Geschwindigkeit.
- Assertion müssen erst aktiviert werden:
  `java -ea Main` (enable assertions)

Überprüfung durch Zusicherungen

Zusicherungen werden zur Überprüfung von Eigenschaften verwendet, die nach Vertrag garantiert gelten müssen.

- Im Idealfall wird der Vertrag eingehalten, die assert-Instruktionen machen gar nichts.
- Ist der Vertrag verletzt, so führen die assert-Instruktionen zum Programmabbruch und zur Meldung des Vertragsbruchs.
- Assertions werden nur zum Testen des Vertrags benutzt. Das Verhalten des Programms soll vom Ein- oder Ausschalten von Assertions unbeeinflusst sein.

Überprüfung durch Zusicherungen

Zusicherungen werden zur Überprüfung von Eigenschaften verwendet, die nach Vertrag garantiert gelten müssen.

- Nach Konvention werden die Vorbedingungen an Argumente in öffentlichen Methoden **nicht** mit `assert` geprüft. Dort soll z.B. `IllegalArgumentException` ausgelöst werden.
- Die Argumente and Vorbedingungen in privaten Methoden können jedoch mit `assert` überprüft werden.
- Invarianten und Nachbedinungen werden mit `assert` überprüft.

Java Modelling Language

Formale Sprache zum Ausdrücken von Verträgen in Java.

- Spezifikationen in speziellen Kommentaren `/*@ ... @*/`
- Aussagen in Java-artiger Syntax

```
/*@ invariant
  @   (next == null && len == 0) ||
  @   (next != null && len == 1 + next.len);
  @*/
```

- ausdrucksstärker als Java-Zusicherungen

```
//@assert (\forall int i;
//@              i <= 0 && i < a.length; a[i] >= 0);
```

`http://jmlspecs.org`

Java Modelling Language

JML dient als gemeinsame Sprache vieler Analysetools.

Anwendungsgebiete:

- Generierung von Laufzeittests, z.B. OpenJML

  Vorbedingungen, Nachbedingungen, Invarianten, etc. können zur Laufzeit des Programms überprüft werden.
- Generierung von Unittests, z.B. `jmlunit`
- statische Programmanalyse, z.B. ESC/Java2, OpenJML, ...

  automatische Erkennung verschiedener möglicher Vertragsbrüche, z.B. Invariante `field != null`
- formale Programmverifikation, z.B. Krakatoa
- ...

JML — Vor- und Nachbedingungen

Beispiel:

```
/*@ requires betrag > 0;
  @ ensures (kontoStand == \old(kontoStand) + betrag);
  @*/
int einzahlen(int betrag) {
   // ...
}
```

JML — Vor- und Nachbedingungen

Beispiel:

```
/*@ requires a != null && (\forall int i;
  @                              0 <= i && i < a.length-1;
  @                              a[i] <= a[i+1]);
  @ ensures
  @  (\result >= 0  && \result < a.length
  @   && a[\result] == x)
  @  ||
  @  (\result == -1 &&
  @    (\forall int k; 0 <= k && k < a.length;
  @       a[k] != x));
  @*/
int binarySearch(int[] a, int x) {
   // ...
}
```

JML — Effektbeschreibungen

Beispiel:

```
/*@ requires betrag > 0;
  @ ensures (kontoStand == \old(kontoStand) - betrag);
  @ signals (KontoException e)
  @           betrag > kontoStand &&
  @           kontoStand == \old(kontoStand);
  @*/
int abheben(int betrag) {
   // ...
}
```

JML — Invarianten

Beispiel:

```
public class IntSet {
    private int[] elems;
    private int size;

    //@ invariant elems != null;

    /*@ invariant (\forall int i; 0 <= i && i < size-1;
      @                  elems[i] < elems[i+1])
      @*/

    ...
}
```

Statische Analyse

Beispiel: ESC/Java2 (Extended Static Checking for Java)

- Tool zum Finden von Fehlern in Java-Programmen
- Implementierung verschiedener Analysen, z.B. zum Erkennen von NullPointerExceptions
- weder korrekt (falsche Programme werden akzeptiert) noch vollständig (richtige Programme werden abgelehnt)
- Funktioniert nur mit älteren JDK-Versionen.
  Nachfolger in Entwicklung: OpenJML
- Beispiel: `Bag.java` (Erik Poll, siehe Vorlesungshomepage)

Beispiel: Krakatoa

- Programmverifikation (Korrektheit)
- Liefert stärkere Garantien, ist aber auch aufwändiger und benötigt mehr Wissen über Theorembeweiser

Zusammenfassung: Design by Contract und JML

Design by Contract

- Methodologische Richtlinien zur komponentenbasierten Softwareentwicklung
- Entwicklungsprinzipen:
  - Benutze nur die im Vertrag einer Klasse zugesichterten Leistung, stelle Vorbedingungen stets sicher.
  - Dokumentiere Klasseninvarianten.
  - Minimiere das öffentliche Interface von Klassen.
- Ursprünglich in der Programmiersprache Eiffel (1985) entwickelt.

Java Modelling Language

- formale Spezifikation von Verträgen
- Basis für Programmanalyse
  (Testen, statische Analyse, Verifikation)