Kapitel 1

Einführung

1.1 Was ist Semantik

Während die *Syntax* einer Sprache die Gestalt korrekt geformter Sätze der Sprache beschreibt, legt die *Semantik* der Sprache die Bedeutung der syntaktisch korrekten Sätze fest.

In Bezug auf Programme spezifiziert die Semantik eines Programms, was bei Ausführung des Programms passieren soll.

1.2 Wozu verwendet man Semantik

Kenntnisse von der Semantik eines Programms sind nützlich für

- Softwareentwicklung
  - Gliederung in Teilaufgaben
  - Schnittstellendefinition
  - Argumentieren über Programme
  - Fehlersuche
  - Korrektheitsbeweise
- Konstruktion von Übersetzern
  - Korrektheit (Übereinstimmung von Quell- und Zielprogramm)
  - Optimierung ("Äquivalenz" von Originalprogramm und verbessertem Programm)
  - Programmanalyse
- Programmiersprachen
  - neue Konzepte
  - Orthogonalität

Ansätze:

- Compiler und Rechner
  Nachteile: Compiler und Rechner sind nur partiell bekannt und können fehlerhaft sein.
- Mathematische Konzepte
  Vorteil: Abstraktion von Nebensächlichkeiten, Möglichkeit formaler Beweise
  Nachteil: der mathematische Formalismus benötigt ebenfalls eine "Semantik".

Kapitel 2

Operationelle Semantik

2.1 Ausdrücke

Beispiele für Ausdrücke:

Basisausdrücke:

1	(das ASCII-Zeichen)
-173	(die Zeichenfolge)
$az1$	(ein Variablenname)

Zusammengesetzte Ausdrücke:

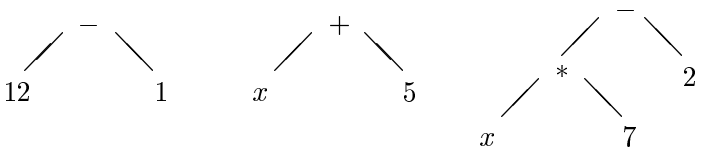
$12 - 1$	(Operatoranwendung)
$x + 5$	
$x * 7 - 2$	

Die Syntax der Ausdrücke ist nur eine textuelle Repräsentation.

2.1.1 Syntax von Ausdrücken

Die *abstrakte Syntax* liefert eine verbesserte Darstellung, indem zusammengehörige Bestandteile zusammengesetzter Ausdrücke hierarchisch gruppiert werden.

Beispiele:



Diese Strukturen repräsentieren vollständig geklammerte Ausdrücke.

Im folgenden verwenden wir *Grammatiken*. Diese bilden eine induktive Vorschrift zum Aufbau komplexer syntaktischer Strukturen aus einfachen syntaktischen Strukturen.

Für Ausdrücke bilden Konstanten c und Variablen x den Induktionsanfang.
Operatoranwendungen bilden den Induktionsschritt.

Aus diesem Definitionsprinzip für Ausdrücke leiten wir ein Beweisprinzip für Behauptungen über Ausdrücke ab.

Definition: Wir spezifizieren die abstrakte Syntax von *Integer-Ausdrücken* durch eine kontextfreie Grammatik.

$$
\begin{array}{rcll}
e & ::= & c & \text{(Integer-Konstante)} \\
 & | & x & \text{(Integer-Variable)} \\
 & | & (-e_1) & \text{(unäre Operatoranwendung)} \\
 & | & (e_1 + e_2) & \text{(binäre Operatoranwendung)} \\
 & | & (e_1 - e_2) & \text{"} \\
 & | & (e_1 * e_2) & \text{"}
\end{array}
$$

Bemerkung: Wir nehmen an, daß die syntaktische Repräsentation von Konstanten, Variablen und Operatoren verschieden ist. Die Indizes in den rechten Seiten dienen der Unterscheidung der Vorkommen von e (e ist das Nichtterminal für Ausdrücke).

Definition: Die *Länge* $|e|$ eines Ausdrucks e ist folgendermaßen induktiv definiert:

$$
\begin{array}{ll}
|c| & = 1 \\
|x| & = 1 \\
|-e_1| & = |e_1| + 1 \\
|e_1 \,\square\, e_2| & = |e_1| + |e_2| + 1 \ , \quad \text{wobei } \square \in \{+, -, *\}.
\end{array}
$$

Definition: Die *Tiefe* $d(e)$ eines Ausdrucks e ist induktiv definiert als:

$$
\begin{array}{ll}
d(c) & = 0 \\
d(x) & = 0 \\
d(-e_1) & = d(e_1) + 1 \\
d(e_1 \,\square\, e_2) & = \max(d(e_1), d(e_2)) + 1 \ , \quad \text{wobei } \square \in \{+, -, *\}.
\end{array}
$$

Als Fingerübung soll nun der folgende Satz induktiv über dem Aufbau eines Ausdrucks bewiesen werden.

Satz: $|e| \ \leq \ 2^{d(e)+1} - 1$.

Beweis:

Induktionsanfang $e \equiv c$ oder $e \equiv x$: $\quad |e| = 1 = 2 - 1 = 2^{d(e)+1} - 1 \quad \surd$

Induktionsschritt:

- $e \equiv -e_1$: $\quad |e| = |e_1| + 1 \ \leq \ (2^{d(e_1)+1} - 1) + 1 = 2^{d(e_1)+1} = 2^{d(e)} \leq 2^{d(e)+1} - 1$

- $e \equiv e_1 \,\square\, e_2$: $\quad |e| = |e_1| + |e_2| + 1 \ \leq \ (2^{d(e_1)+1} - 1) + (2^{d(e_2)+1} - 1) + 1 = 2 \cdot (2^{d(e_1)} + 2^{d(e_2)}) - 1 \leq 2 \cdot 2^{\max(d(e_1), d(e_2))} - 1 \leq 2 \cdot 2^{\max(d(e_1), d(e_2))+1} - 1 = 2 \cdot 2^{d(e)} - 1 = 2^{d(e)+1} - 1$

Definition: Unter Verwendung von Integer-Ausdrücken definieren wir *Boolesche-Ausdrücke* b:

$$\begin{array}{rcll} b & ::= & \mathit{True} \mid \mathit{False} & (\text{BOOLEsche Konstanten}) \\ & \mid & (\neg b_1) & (\text{unäre Operatoranwendung}) \\ & \mid & (b_1 \wedge b_2) & (\text{binäre Operatoranwendung}) \\ & \mid & (b_1 \vee b_2) & \text{"} \\ & \mid & (e_1 = e_2) & (\text{Vergleichsoperator}) \\ & \mid & (e_1 < e_2) & \text{"} \end{array}$$

Der Einfachheit halber verzichten wir auf BOOLEsche Variablen.

Beispiele für BOOLEsche Ausdrücke: $\neg(y = 7)$, True, $(x < 0) \vee (0 < x)$

2.1.2 Semantik von Ausdrücken

Definition: Wir bezeichen eine Abbildung $\sigma : \mathit{Vars} \to D$, die jeder Variablen aus *Vars* einen Wert aus D zuordnet, als *Variablenbelegung.*

Bemerkung: Zugunsten einer übersichtlicheren Notation verzichten wir in Zukunft auf Klammern für Funktionsargumente. Beispielsweise schreiben wir für $f(x)$ einfach $f\,x$.

Definition: Sei *Vars* eine Menge von Variablen und $\mathbb{Z}$ die Menge der ganzen Zahlen (Integer-Werte).
Die *Semantik* eines Integer-Ausdrucks e ist eine Funktion $[\![e]\!] : (\mathit{Vars} \to \mathbb{Z}) \to \mathbb{Z}$, die definiert ist wie folgt:

$$\begin{array}{ll} [\![c]\!]\sigma & = c \\ [\![x]\!]\sigma & = \sigma\, x \\ [\![-e_1]\!]\sigma & = -[\![e_1]\!]\sigma \\ [\![e_1 \,\square\, e_2]\!]\sigma & = [\![e_1]\!]\sigma \,\square\, [\![e_2]\!]\sigma, \quad \square : \mathbb{Z}^2 \to \mathbb{Z},\ \square \in \{+, -, *\} \end{array}$$

Genaugenommen benötigen wir zusätzliche Funktionen, die jeder Repräsentation einer Zahl oder eines Operators die eigentliche Zahl bzw. den eigentlichen Operator zuordnet.
Um diesen "Overhead" zu vermeiden, identifizieren wir einfach Repräsentationen von Zahlen und deren Bedeutung bzw. Repräsentationen von Operatoren und die Operatoren selbst.

Beispiel:

Für eine Variablenbelegung $\sigma_1 = \{x \mapsto 3,\ y \mapsto 7\}$ erhalten wir $[\![x - y]\!]\sigma_1 = -4$.

Beispiel:

Für $\sigma_2 = \{x \mapsto 5,\ y \mapsto -3\}$ erhalten wir $[\![x - y]\!]\sigma_2 = 8$.

Wird definieren die Semantik BOOLEscher Ausdrücke analog.

Definition: Sei *Vars* eine Menge von Variablen und $\mathbb{B} = \{\mathit{True}, \mathit{False}\}$ die Menge der BOOLEschen Konstanten.
Die *Semantik* eines BOOLEschen Ausdrucks b ist eine Funktion $[\![b]\!] : (\mathit{Vars} \to \mathbb{Z}) \to \mathbb{B}$, die definiert ist wie folgt:

$$
\begin{array}{lll}
[\![\mathit{True}]\!]\sigma & = \mathit{True} & \\
[\![\mathit{False}]\!]\sigma & = \mathit{False} & \\
[\![\neg b_1]\!]\sigma & = \neg[\![b_1]\!]\sigma & \\
[\![b_1 \,\square\, b_2]\!]\sigma & = [\![b_1]\!]\sigma \,\square\, [\![b_2]\!]\sigma, & \square : \mathbb{B}^2 \to \mathbb{B},\ \square \in \{\wedge, \vee\} \\
[\![e_1 \,\square\, e_2]\!]\sigma & = [\![e_1]\!]\sigma \,\square\, [\![e_2]\!]\sigma, & \square : \mathbb{Z}^2 \to \mathbb{B},\ \square \in \{=, <\}
\end{array}
$$

Beispiel:

$$
\begin{aligned}
\sigma_1 &= \{x \mapsto 3,\ y \mapsto 7\} \\
[\![(x < y) \vee (y < 0)]\!]\sigma_1 &= [\![x < y]\!]\sigma_1 \vee [\![y < 0]\!]\sigma_1 \\
&= (\sigma_1\, x < \sigma_1\, y) \vee (\sigma_1\, y = 0) \\
&= \mathit{True} \vee \mathit{False} \\
&= \mathit{True}
\end{aligned}
$$

2.2 Eine einfache Programmiersprache

Wir wollen hier eine einfache Programmiersprache betrachten, die aus folgenden Konstrukten besteht:

- Ausdrücke (wie im vorigen Abschnitt definiert)
- Zuweisungen
- bedingte Anweisungen
- Schleifen

Beispiel: Das Programm $\tilde{p}$.

```
// Berechnung von y := ⌈log2 x⌉
if (x < 1) then
    y := −1;
else
    y := 0;
    z := 1;
    while (¬(x < z)) do
      y := y + 1;
      z := 2 ∗ z;
    end;
end;
```

Definition: Die Syntax eines *Programms* (Programmfragments) p ist gegeben als:

$$
\begin{array}{lcll}
p & ::= & s\,;\,p & \text{(Anweisung, gefolgt von Programm)} \\
 & \mid & \varepsilon & \text{(leere Anweisung)}
\end{array}
$$

Anweisungen (Statements) s haben folgenden Aufbau:

$$
\begin{array}{lcll}
s & ::= & x := e & \text{(Zuweisung)} \\
 & \mid & \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end} & \text{(bedingte Verzweigung)} \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{end} & \text{(Schleife)}
\end{array}
$$

2.3 Operationelle Small-Step-Semantik

Idee: Wir wollen die Programmausführung als Folge einfacher Berechnungsschritte darstellen, wobei jeder Berechnungsschritt die Variablenbelegung verändert. Die Semantik eines Programms ist dann der Effekt einer solchen Folge.

Zunächst führen wir einen Operator $\oplus$ zum Ersetzen von Variablenwerten in Variablenbelegungen ein.

Definition: Sei $\sigma : \mathit{Vars} \to D$ eine Variablenbelegung und $\{x_1, \ldots, x_k\} \subseteq \mathit{Vars}$ eine Menge von Variablen.
Dann ist $\sigma \oplus \{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$ die Variablenbelegung, die man erhält, wenn man jede Variable x_i mit dem Wert d_i überschreibt $(1 \leq i \leq k)$.

Definition: Sei p_1 das auszuführende Programm(fragment), σ_1 die Ausgangsbelegung der Variablen für p_1.
Ein *Berechnungsschritt* ist eine Relation

$$p_1 \mid \sigma_1 \;\to\; p_2 \mid \sigma_2,$$

wobei p_2 das Programm nach Ausführung des Berechnungsschritts und σ_2 die Variablenbelegung nach Ausführung des Berechnungsschritts ist.
Wir bezeichnen mit $\to^*$ die *reflexive und transitive Hülle* von $\to$.

Sei $p' \equiv s; p$.

Wir definieren Berechnungsschritte durch Fallunterscheidung möglicher Formen für Anweisungen s:

- $s \equiv x := e$
  Dann gilt $s; p \mid \sigma_1 \;\to\; p \mid \sigma_2$ mit $\sigma_2 = \sigma_1 \oplus \{x \mapsto [\![e]\!]\sigma_1\}$.
  e wird also mittels σ_1 ausgewertet und die Variable x mit dem Ergebnis überschrieben.

- $s \equiv$ `if` b `then` p_1 `else` p_2 `end`
  Dann gilt $s; p \mid \sigma_1 \;\to\; p_i\, p \mid \sigma_1$, wobei $i = 1$, falls $[\![b]\!]\sigma_1 = \mathit{True}$ (`then`-Fall) und
  $i = 2$, falls $[\![b]\!]\sigma_1 = \mathit{False}$ (`else`-Fall).

- $s \equiv$ `while` b `do` p_1 `end`
  Dann gilt $s; p \mid \sigma_1 \;\to\; p_1\, s;\, p \mid \sigma_1$, sofern $[\![b]\!]\sigma_1 = \mathit{True}$ und
  $s; p \mid \sigma_1 \;\to\; p \mid \sigma_1$, sofern $[\![b]\!]\sigma_1 = \mathit{False}$.

  Im ersten Fall wird der Schleifenrumpf betreten und die Schleife muß erneut ausgeführt werden. Im zweiten Fall wird mit der Ausführung des restlichen Programms fortgefahren.

Definition: Eine *Berechnung* für ein Programm(fragment) p auf einer Variablenbelegung σ ist eine (möglicherweise unendliche) Folge von Berechnungsschritten

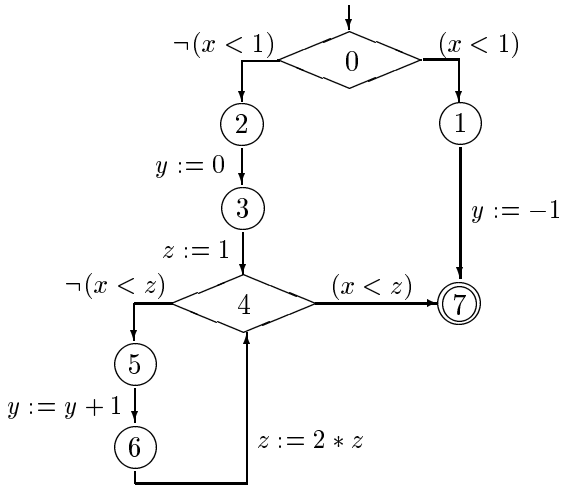
$$p_0 \mid \sigma_0 \;\to\; p_1 \mid \sigma_1 \;\to\; \cdots \;\to\; p_m \mid \sigma_m \;\to\; \cdots,$$

wobei $p_0 \mid \sigma_0 \equiv p \mid \sigma$.
Eine *erfolgreich terminierende* Berechnung für ein Programm(fragment) p auf einer Variablenbelegung σ ist eine Folge der Form $p_0 \mid \sigma_0 \;\to\; p_1 \mid \sigma_1 \;\to\; \cdots \;\to\; \varepsilon \mid \sigma_k$.

Bemerkung: Die bei Auswertung von p möglicherweise erreichbaren Programmfragmente p_i repräsentieren den Kontrollflußgraphen (KFG) von p.

Die Knoten des Kontrollflußgraphen repräsentieren die Programmpunkte, d. h. die noch abzuarbeitenden Programmstücke.
Die Kanten des Kontrollflußgraphen werden mit der auszuführenden Anweisungen (bei Zuweisungen) bzw. mit den Bedingungen des Programmschritts (bei `if`- oder `while`-Statements) beschriftet. Eine Kante stellt immer einen möglichen Berechnungsschritt dar.

Beispiel: Kontrollflußgraph für $\tilde{p}$



Jeder Knoten entspricht also dem noch abzuarbeitenden Rest des Programms.
Für Knoten 0 wäre das beispielsweise gerade $\tilde{p}$, für die anderen Knoten sind die entsprechenden Programmteile in der folgenden Tabelle aufgelistet:

Knoten	Programmteil
1	$y := -1;$
2	$y := 0;\ z := 1;$ `while` $(\cdots)$ `do` $\cdots$ `end`;
3	$z := 1;$ `while` $(\cdots)$ `do` $\cdots$ `end`;
4	`while` $(\neg(x < z))$ `do` $\cdots$ `end`;
5	$y := y + 1;\ z := 2 * z;$ `while` $(\cdots)$ `do` $\cdots$ `end`;
6	$z := 2 * z;$ `while` $(\cdots)$ `do` $\cdots$ `end`;
7	ε

Beispiel: Berechnung von $\tilde{p}$ für $x = 3$

$$
\begin{array}{rlll}
 & 0 \mid \{x \mapsto 3,\ y \mapsto _,\ z \mapsto _\} & \to & 2 \mid \{x \mapsto 3,\ y \mapsto _,\ z \mapsto _\} \\
\to & 3 \mid \{x \mapsto 3,\ y \mapsto 0,\ z \mapsto _\} & \to & 4 \mid \{x \mapsto 3,\ y \mapsto 0,\ z \mapsto 1\} \\
\to & 5 \mid \{x \mapsto 3,\ y \mapsto 0,\ z \mapsto 1\} & \to & 6 \mid \{x \mapsto 3,\ y \mapsto 1,\ z \mapsto 1\} \\
\to & 4 \mid \{x \mapsto 3,\ y \mapsto 1,\ z \mapsto 2\} & \to & 5 \mid \{x \mapsto 3,\ y \mapsto 1,\ z \mapsto 2\} \\
\to & 6 \mid \{x \mapsto 3,\ y \mapsto 2,\ z \mapsto 2\} & \to & 4 \mid \{x \mapsto 3,\ y \mapsto 2,\ z \mapsto 4\} \\
\to & 7 \mid \{x \mapsto 3,\ y \mapsto 2,\ z \mapsto 4\} & &
\end{array}
$$

Im weiteren bezeichnen wir partielle Funktionen mit $\rightsquigarrow$.

Definition: Die partielle Funktion $\mathit{Eff}(p) : (\mathit{Vars} \to \mathbb{Z}) \rightsquigarrow (\mathit{Vars} \to \mathbb{Z})$, die definiert ist durch $\mathit{Eff}(p) = \{\sigma \mapsto \sigma' \mid p \mid \sigma \to^* \varepsilon \mid \sigma'\}$ nennen wir den *Effekt* eines Programms p.

Bemerkung:

- $\mathit{Eff}(p)$ beschreibt nur das Ein-/Ausgabeverhalten des Programms p.

- Programmabbrüche (etwa Division durch Null) oder Nichtterminierung (z. B. Endlosschleifen) werden nicht unterschieden.

Wir zeigen nun folgenden nützlichen Satz.

Satz: $p_1\, p_2 \mid \sigma_1 \;\to^*\; \varepsilon \mid \sigma_2$ gilt genau dann, wenn eine Variablenbelegung σ existiert mit $p_1 \mid \sigma_1 \;\to^*\; \varepsilon \mid \sigma$ und $p_2 \mid \sigma \;\to^*\; \varepsilon \mid \sigma_2$.

Beweis:

"$\Rightarrow$": Sei $q_0 \mid \tau_0 \;\to\; q_1 \mid \tau_1 \;\to\; \cdots \;\to\; q_m \mid \tau_m$ die Berechnungsfolge
mit $q_0 \mid \tau_0 \;\equiv\; p_1\, p_2 \mid \sigma_1$ und $q_m \mid \tau_m \;\equiv\; \varepsilon \mid \sigma$.
Sei j in dem Sinne maximal, daß für alle $i \leq j$ gilt: $q_i \;\equiv\; q_i'\, p_2$.
Insbesondere gilt $q_0' \;\equiv\; p_1\, p_2$ und $q_j' \;\equiv\; \varepsilon$.
Außerdem gilt nach Definition von $\to$: $q_0' \mid \tau_0 \;\to\; \cdots \;\to\; q_j' \mid \tau_j$.
Folglich gilt $p_1 \mid \sigma_1 \;\to^*\; \varepsilon \mid \sigma$ mit $\sigma = \tau_j$ und $p_2 \mid \sigma \;\to^*\; \varepsilon \mid \sigma_2$.

"$\Leftarrow$": Mittels Induktion über die Länge einer Berechnung $p_1 \mid \sigma_1 \;\to\; \cdots \;\to\; \varepsilon \mid \sigma$
zeigt man, daß $p_1\, p_2 \mid \sigma_1 \;\to^*\; p_2 \mid \sigma$ gilt.
Nun kann man die Berechnung $p_2 \mid \sigma \;\to^*\; \varepsilon \mid \sigma_2$ einfach anfügen. □

2.4 Operationelle Big-Step-Semantik

Ist man nur am Effekt *Eff*(p) eines Programms interessiert, möchte man vielleicht ganz auf das Berechnungskonzept des vorigen Abschnitts verzichten.
Stattdessen kann man **direkt** ein *Regelsystem* angeben, daß es erlaubt, Übergänge $(\sigma \mapsto \sigma') \in$ *Eff*(p) zwischen Variablenbelegungen zu *verifizieren*.

Die Regeln, die wir angeben werden, erlauben es aus Aussagen über Komponenten, Aussagen über ein Gesamtprogramm abzuleiten.
Solche Regeln geben wir immer in der Form $\frac{\text{Voraussetzungen}}{\text{Schlußfolgerung}}$ an.

Definition: Die *Aussagen* unseres Regelsystems haben die Gestalt $[\![b]\!]\sigma = t,\ t \in \{\mathit{False}, \mathit{True}\}$ oder $p \mid \sigma_1 \mapsto \sigma_2$.

Axiome des Systems sind

1. $[\![b]\!]\sigma = \mathit{False}$,
2. $[\![b]\!]\sigma = \mathit{True}$,
3. $\varepsilon \mid \sigma \mapsto \sigma$,
4. $x := e; \mid \sigma \mapsto \sigma \oplus \{x \mapsto [\![e]\!]\sigma\}$.

Regeln:

- Conc $\quad \dfrac{s; \mid \sigma_1 \mapsto \sigma \qquad p \mid \sigma \mapsto \sigma_2}{s; p \mid \sigma_1 \mapsto \sigma_2}$
- If1 $\quad \dfrac{[\![b]\!]\sigma_1 = \mathit{True} \qquad p_1 \mid \sigma_1 \mapsto \sigma_2}{\texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end};} \mid \sigma_1 \mapsto \sigma_2$

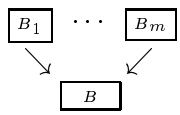
- If2 $$\frac{[\![b]\!]\sigma_1 = \mathit{False} \qquad p_2 \mid \sigma_1 \mapsto \sigma_2}{\texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end;} \mid \sigma_1 \mapsto \sigma_2}$$

- While1 $$\frac{[\![b]\!]\sigma_1 = \mathit{True} \qquad p \mid \sigma_1 \mapsto \sigma \qquad \texttt{while } b \texttt{ do } p \texttt{ end} \mid \sigma \mapsto \sigma_2}{\texttt{while } b \texttt{ do } p \texttt{ end} \mid \sigma_1 \mapsto \sigma_2}$$

- While2 $$\frac{[\![b]\!]\sigma = \mathit{False}}{\texttt{while } b \texttt{ do } p \texttt{ end;} \mid \sigma \mapsto \sigma}$$

In den Voraussetzungen kommen nur Aussagen über Bestandteile des Programms in der Schlußfolgerung vor (structural operational semantics (SOS)).

Ein *Beweis* einer Aussage A ist ein geordneter (einwachsender) Baum, dessen Knoten Aussagen sind. Die Wurzel des Baums ist die zu beweisenden Aussage A. Axiome bilden die Blätter des Baums.

Wann immer ein Baum der Form 

existiert,

muß es auch eine Regel $\frac{B_1 \ \cdots \ B_m}{B}$ geben.

Gibt es einen Beweisbaum für A, dann schreiben wir auch kurz $\vdash A$.

Haben wir eine feste Ordnung $(x_1, \ldots, x_n)$ für die Variablen einer Variablenbelegung $\sigma = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ vereinbart, so schreiben wir im folgenden abkürzend $\sigma = (d_1, \ldots, d_n)$.

Beispiel: Anwendung auf unser Beispielprogramm $\tilde{p}$

Beweis von $\tilde{p} \mid \{x \mapsto 3, y \mapsto _, z \mapsto _\} \mapsto \{x \mapsto 3, y \mapsto 2, z \mapsto 4\}$

$$\frac{\overbrace{y := y + 1; \mid (3,1,2) \mapsto (3,2,2)}^{A_1} \qquad \overbrace{z := 2 * z; \mid (3,2,2) \mapsto (3,2,4)}^{A_2}}{\underbrace{y := y + 1; z := 2 * z; \mid (3,1,2) \mapsto (3,2,4)}_{C_1}}$$

$$\frac{\overbrace{y := y + 1; \mid (3,0,1) \mapsto (3,1,1)}^{A_3} \qquad \overbrace{z := 2 * z; \mid (3,1,1) \mapsto (3,1,2)}^{A_4}}{\underbrace{y := y + 1; z := 2 * z; \mid (3,0,1) \mapsto (3,1,2)}_{C_2}}$$

$$\frac{\overbrace{[\![\neg(x < z)]\!](3,2,4) = \mathit{False}}^{A_5}}{\underbrace{\texttt{while } (\neg(x < z)) \texttt{ do } y := y + 1; z := 2 * z; \texttt{ end;} \mid (3,2,4) \mapsto (3,2,4)}_{C_3}}$$

$$\frac{\overbrace{[\![\neg(x < z)]\!](3,1,2) = \mathit{True}}^{A_6} \qquad C_1 \qquad C_3}{\underbrace{\texttt{while } (\neg(x < z)) \texttt{ do } y := y + 1; z := 2 * z; \texttt{ end} \mid (3,1,2) \mapsto (3,2,4)}_{C_4}}$$

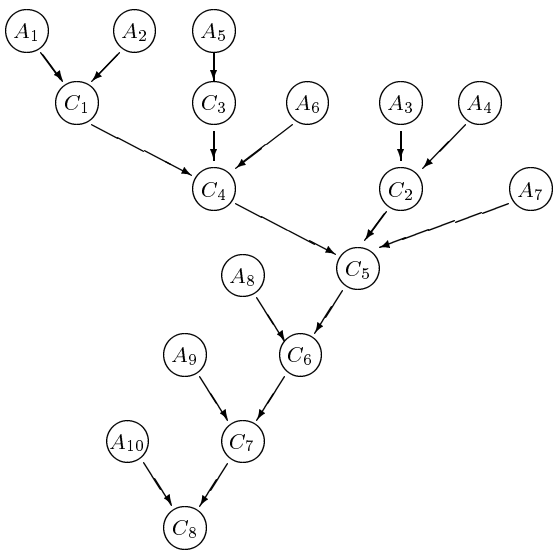
$$\underbrace{\dfrac{\overbrace{[\![\neg(x<z)]\!](3,0,1)=\mathit{True}}^{A_7} \qquad C_2 \qquad C_4}{\texttt{while}\ (\neg(x<z))\ \texttt{do}\ y:=y+1; z:=2*z;\ \texttt{end} \mid (3,0,1)\mapsto(3,2,4)}}_{C_5}$$

$$\underbrace{\dfrac{\overbrace{z:=1; \mid (3,0,_)\mapsto(3,0,1)}^{A_8} \qquad C_5}{z:=1;\ \texttt{while}\ (\neg(x<z))\ \texttt{do}\ \cdots\ \texttt{end} \mid (3,0,_)\mapsto(3,2,4)}}_{C_6}$$

$$\underbrace{\dfrac{\overbrace{y:=0; \mid (3,_,_)\mapsto(3,0,_)}^{A_9} \qquad C_6}{y:=0;\ z:=1;\ \texttt{while}\ (\cdots)\ \texttt{do}\ \cdots\ \texttt{end}; \mid (3,_,_)\mapsto(3,2,4)}}_{C_7}$$

$$\underbrace{\underbrace{\dfrac{\overbrace{[\![x<1]\!](3,_,_)=\mathit{False}}^{A_{10}} \qquad C_7}{\texttt{if}\cdots\texttt{then}\cdots\texttt{else}\ y:=0;\ z:=1;\ \texttt{while}\ (\cdots)\ \texttt{do}\ \cdots;\ \texttt{end};\ \texttt{end}; \mid (3,_,_)\mapsto(3,2,4)}}_{\tilde{p}}}_{C_8}$$

Der entsprechende Beweisbaum sieht dann so aus:



Wir zeigen nun folgenden Zusammenhang zwischen Small-Step- und Big-Step-Semantik.

Satz: $p \mid \sigma_1 \ \to^* \ \varepsilon \mid \sigma_2$ gilt genau dann, wenn $\vdash p \mid \sigma_1 \mapsto \sigma_2$ gilt.

Beweis:

"⇒": Induktion über die Länge n einer Berechnungsfolge $p \mid \sigma_1 \ \to \ \cdots \ \to \ \varepsilon \mid \sigma_2$:

$n = 0:\ p \mid \sigma_1 \ \equiv \ \varepsilon \mid \sigma_2 \ \implies \vdash \varepsilon \mid \sigma_2 \mapsto \sigma_2$ (Axiom 3).

$n > 0$:

- Fall 1: $p \equiv s; p'$ mit $p' \not\equiv \varepsilon$
  Nach dem letzten Satz gibt es eine Variablenbelegung σ mit $s \mid \sigma_1 \rightarrow^* \varepsilon \mid \sigma$ und $p' \mid \sigma \rightarrow^* \varepsilon \mid \sigma_2$, wobei diese Berechnungen jeweils Längen $< n$ haben.
  Nach Induktionsvoraussetzung gilt dann $\vdash s; \mid \sigma_1 \mapsto \sigma$ und $\vdash p' \mid \sigma \mapsto \sigma_2$.
  Aus der Conc-Regel folgt dann $\vdash s; p' \mid \sigma_1 \mapsto \sigma_2$.
- Fall 2: $p \equiv s;$
  - $s \equiv x := e$
    Mit $\sigma_2 = \sigma_1 \oplus \{x \mapsto [\![e]\!]\sigma_1\}$ folgt $\vdash x := e; \mid \sigma_1 \mapsto \sigma_2$ (Axiom 4).
  - $s \equiv$ `if` b `then` p_1 `else` p_2 `end`
    Es sei $[\![b]\!]\sigma_1 = \mathit{True}$. Dann gilt $s; \mid \sigma_1 \rightarrow p_1 \mid \sigma_1 \rightarrow^* \varepsilon \mid \sigma_2$.
    Nach Induktionsvoraussetzung gilt $\vdash p_1 \mid \sigma_1 \mapsto \sigma_2$.
    Aus der If1-Regel folgt dann $\vdash s; \mid \sigma_1 \mapsto \sigma_2$.

    Die Argumentation für $[\![b]\!]\sigma_1 = \mathit{False}$ verläuft völlig analog.
  - $s \equiv$ `while` b `do` p `end`
    Es sei $[\![b]\!]\sigma_1 = \mathit{True}$.
    Dann gilt $s; \mid \sigma_1 \rightarrow p\ s \mid \sigma_1 \rightarrow^* \varepsilon \mid \sigma_2$.
    Nach dem letzten Satz gibt es eine Variablenbelegung σ mit kürzeren Berechnungen $p \mid \sigma_1 \rightarrow^* \varepsilon \mid \sigma$ und $s; \mid \sigma \rightarrow^* \varepsilon \mid \sigma_2$.
    Nach Induktionsvoraussetzung gilt $\vdash p \mid \sigma_1 \mapsto \sigma$ und $\vdash s; \mid \sigma \mapsto \sigma_2$.
    Aus der While1-Regel folgt dann $\vdash s; \mid \sigma_1 \mapsto \sigma_2$.

    Sei nun $[\![b]\!]\sigma_1 = \mathit{False}$.
    Dann gilt $s; \mid \sigma_1 \rightarrow \varepsilon \mid \sigma_1$, d. h. $\sigma_2 \equiv \sigma_1$.
    Aus der While2-Regel folgt dann $\vdash s; \mid \sigma_1 \mapsto \sigma_2$.

"$\Leftarrow$": Übungsaufgabe ? □

Kapitel 3

Denotationelle Semantik

Eff(p) ist eine partielle Funktion. Nun stellt sich die Frage, ob man auch direkt mit solchen Funktionen rechnen kann.

3.1 Definition einer denotationellen Semantik

Sei $[\![p]\!] : (Vars \to \mathbb{Z}) \rightsquigarrow (Vars \to \mathbb{Z})$ eine partielle Funktion.

Definition:

- $[\![\varepsilon]\!]\sigma = \sigma$ (Identität)
- $[\![s; p]\!]\sigma = [\![p]\!]\,([\![s;]\!]\sigma)$ (Komposition)
- $[\![x := e;]\!]\sigma = \sigma \oplus \{x \mapsto [\![e]\!]\sigma\}$
- $[\![\texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end;}]\!]\sigma = \begin{cases} [\![p_1]\!]\sigma, & \text{falls } [\![b]\!]\sigma = \mathit{True} \\ [\![p_2]\!]\sigma, & \text{falls } [\![b]\!]\sigma = \mathit{False} \end{cases}$
- $[\![\texttt{while } b \texttt{ do } p \texttt{ end;}]\!]\sigma = \begin{cases} [\![\texttt{while } b \texttt{ do } p \texttt{ end;}]\!]\,([\![p]\!]\,\sigma), & \text{falls } [\![b]\!]\sigma = \mathit{True} \\ \sigma, & \text{falls } [\![b]\!]\sigma = \mathit{False} \end{cases}$

Problem: $[\![\texttt{while } b \texttt{ do } p \texttt{ end;}]\!]$ wird durch sich selbst definiert.
Um solche rekursiven Gleichungen zu lösen, nutzen wir strukturelle Eigenschaften der Menge $(Vars \to \mathbb{Z}) \rightsquigarrow (Vars \to \mathbb{Z})$ aus.

Dazu müssen wir zunächst einen Exkurs in die Theorie partiell geordneter Mengen machen.

3.2 Partiell geordnete Mengen

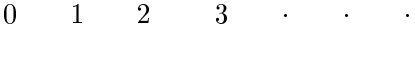
Definition: Sei D eine Menge und $\sqsubseteq \subseteq D \times D$ eine zweistellige Relation über D.
Eine *partielle Ordnung (partial ordered set (po))* ist ein Paar $(D, \sqsubseteq)$ für das gilt:

1. $x \sqsubseteq x \quad \forall x \in D$ (Reflexivität)

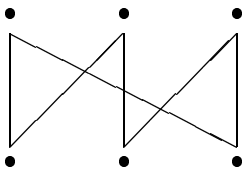
2. $x \sqsubseteq y$ und $y \sqsubseteq x \Rightarrow x = y \quad \forall x, y \in D$ (Antisymmetrie)
3. $x \sqsubseteq y$ und $y \sqsubseteq z \Rightarrow x \sqsubseteq z \quad \forall x, y, z \in D$ (Transitivität)

Bemerkung: Ist '$\sqsubseteq$' bekannt, dann schreiben wir auch D statt $(D, \sqsubseteq)$.

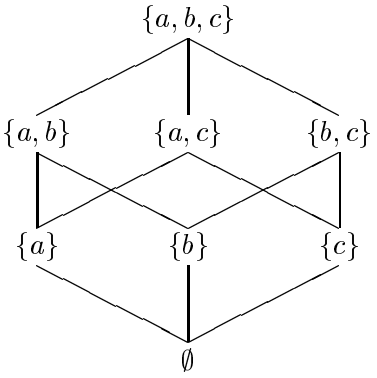
Beispiele für partielle Ordnungen: (HASSE–Diagramme)



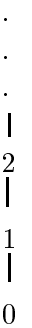
1) 'flache Ordnung' auf $(\mathbb{N}, =)$



2) 3–Crown



3) Teilmengenverband



4) $\mathbb{N}$ mit 'natürlicher Ordnung'

Unter $\mathbb{N}$ verstehen wir hier immer die natürlichen Zahlen einschließlich der 0.

Die strukturerhaltenden Abbildungen zu partiellen Ordnungen sind die monotonen Funktionen.

Definition: Seien $(D_1, \sqsubseteq)$ und $(D_2, \sqsubseteq)$ partielle Ordnungen und f eine Funktion $f : D_1 \to D_2$. Dann heißt f *monoton*, falls für alle $x, y \in D_1$ gilt: $x \sqsubseteq y \Rightarrow fx \sqsubseteq fy$

Definition: Sei $(D, \sqsubseteq)$ eine partielle Ordnung, $X \subseteq D$.
$d \in D$ heißt *obere Schranke* von X, falls $x \sqsubseteq d \ \forall x \in X$. Eine obere Schranke d heißt *kleinste obere Schranke (least upper bound (lub))* von X, falls $d \sqsubseteq d'$ für jede obere Schranke d' von X.

Wir schreiben für kleinste obere Schranken: $d = \bigsqcup X$ oder $d = x \sqcup y$, falls $X = \{x, y\}$ oder $d = \bigsqcup_{i \geq 0} x_i$, falls $X = \{x_i \mid i \geq 0\}$.
Analog definiert man untere Schranken, größte untere Schranken (greatest lower bounds (glb)) und notiert diese mit $\sqcap$.

Definition: Eine *(aufsteigende) Kette (ascending chain (ac))* in einer partiellen Ordnung $(D, \sqsubseteq)$ ist eine (endliche oder unendliche) Folge von Elementen $x_0, x_1, x_2, \ldots \in D$ mit $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$.

Definition: Eine partielle Ordnung $(D, \sqsubseteq)$ heißt *vollständige partielle Ordnung (complete partial order (cpo))*, falls sie die beiden folgenden Eigenschaften besitzt:

1. Es gibt ein kleinstes Element $\bot = \bigsqcap D \in D$
2. Jede aufsteigende Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ besitzt eine kleinste obere Schranke in D.

In den Beispielen zu partiellen Ordnungen ist 3) vollständig.

Die strukturerhaltenden Abbildungen zu vollständigen partiellen Ordnungen sind die stetigen Funktionen.

Definition: Seien $(D_1, \sqsubseteq)$ und $(D_2, \sqsubseteq)$ vollst. partielle Ordnungen und f eine Funktion $f : D_1 \longrightarrow D_2$.
f heißt *stetig*, falls für jede Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ aus D_1 gilt: $f(\bigsqcup_{i\geq 0} x_i) = \bigsqcup_{i\geq 0} f x_i$

Bemerkung: Aus der Stetigkeit einer Funktion f folgt stets deren Monotonie.

Beispiel:

$$\text{Die Funktion } succ : I\!N_\bot \longrightarrow I\!N_\bot \text{ mit } succ(x) = \begin{cases} \bot, & \text{falls } x = \bot \\ x+1, & \text{sonst} \end{cases} \quad \text{ist stetig.}$$

Definition: Sei $(D, \sqsubseteq)$ eine partielle Ordnung.
$d \in D$ heißt *Fixpunkt* einer Funktion $f : D \longrightarrow D$, falls $f\,d = d$ gilt.
Ein Fixpunkt d heißt *kleinster Fixpunkt* einer Funktion f, falls $\forall d' \in D$ mit $f\,d' = d'$ stets $d \sqsubseteq d'$ gilt.

Satz: (KLEENE) Sei $(D, \sqsubseteq)$ eine vollständige partielle Ordnung.
Dann besitzt jede stetige Funktion $f : D \longrightarrow D$ einen kleinsten Fixpunkt $d \in D$.
d ist gegeben durch $d = \bigsqcup_{i\geq 0} f^i\, d$.

Beweis: Definiere eine Folge $d_i,\ i \geq 0$ durch $d_0 = \bot$ und $d_i = f\, d_{i-1}$ für $i > 0$.

1. $d_i,\ i \geq 0$ ist aufsteigende Kette in D, denn:
   $d_0 = \bot \sqsubseteq d_1$
   Für $i > 0$ nehmen wir an, daß $d_{i-1} \sqsubseteq d_i$ gilt, dann ist $d_i = f\, d_{i-1} \sqsubseteq f\, d_i = d_{i+1}$ (da f monoton).

2. $d := \bigsqcup_{i\geq 0} d_i$ ist kleinster Fixpunkt von f, denn:

   - $f\, d = f(\bigsqcup_{i\geq 0} d_i) = \bigsqcup_{i\geq 0} f\, d_i = \bigsqcup_{i\geq 0} d_{i+1} = \bot \sqcup \bigsqcup_{i>0} d_i = \bigsqcup_{i\geq 0} d_i = d$
   - d ist kleinster Fixpunkt von f, d.h. $d \sqsubseteq x\ \ \forall x \in D$ mit $f\, x = x$:
     Sei $x \in D$ mit $f\, x = x$. Zeige: $\bigsqcup_{i\geq 0} d_i \sqsubseteq x$, d.h. $d_i \sqsubseteq x\ \ \forall i \geq 0$:
     $i = 0 : \bot \sqsubseteq x$
     $i > 0$: Annahme $d_i \sqsubseteq x \overset{f \text{ mon.}}{\Longrightarrow} d_{i+1} = f\, d_i \sqsubseteq f\, x = x \quad \square$

Dieser Satz gestattet die Berechnung des kleinsten Fixpunktes einer Funktion f nur dann , falls die Folge $\bot, f\bot, f^2\bot, \ldots$ irgendwann stationär wird ($d := \bigsqcup_{i\geq 0} f^i\bot$).

3.3 Rekursive Gleichungen in der denotationellen Semantik

Wenden wir die Überlegungen des letzten Abschnitts auf die Menge partieller Funktionen $D := (Vars \to \mathbb{Z}) \rightsquigarrow (Vars \to \mathbb{Z})$ an, so stellen wir fest:

- D ist partiell geordnet ($\sqsubseteq$) durch die Relation $\subseteq$. D. h.: Für Funktionen $f, g \in D$ gilt $f \sqsubseteq g$ genau dann, wenn $dom(f) \subseteq dom(g)$ und f und g auf $dom(f)$ übereinstimmen.
- D besitzt die völlig undefinierte Funktion $\emptyset$ als kleinstes Element $\bot$.
- Jede aufsteigende Kette $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \cdots$ in D hat eine kleinste obere Schranke f, denn: Konstruiere f als $f := \bigcup_{i \geq 0} f_i$.

  Sei $\sigma \in dom(f)$.
  $\Longrightarrow \exists i : \sigma \in dom(f_i) \implies \forall j \geq i : \sigma \in dom(f_j) \implies \forall j \geq i : f_i\,\sigma = f_j\,\sigma$
  f ist also wohldefiniert.

  f ist offenbar obere Schranke, da für ein $n \in \mathbb{N}$ stets $f_n \subseteq f$ gilt.

  f ist kleinste obere Schranke, da für jede weitere obere Schranke f' gilt:
  $\forall i : f_i \sqsubseteq f' \implies \forall i : f_i \subseteq f' \implies f = \bigcup_{i \geq 0} f_i \subseteq f' \quad \surd$

Aus diesen Beobachtungen folgt, daß $(D, \subseteq)$ eine vollständige partielle Ordnung ist.

Betrachten wir wieder die rekursive Gleichung

$$\underbrace{[\![\texttt{while } b \texttt{ do } p \texttt{ end;}]\!]}_{X}\,\sigma = \begin{cases} \underbrace{[\![\texttt{while } b \texttt{ do } p \texttt{ end;}]\!]}_{X}\,(\underbrace{[\![p]\!]}_{g}\,\sigma), & \text{falls } [\![b]\!]\sigma = \mathit{True} \\ \sigma, & \text{falls } [\![b]\!]\sigma = \mathit{False} \end{cases}$$

der denotationalen Semantik.

Diese Gleichung hat offenbar die Form $X = F\,X$, wobei $F : D \to D$ die Funktion

$$F\,X\,\sigma = \begin{cases} X\,(g\,\sigma), & \text{falls } [\![b]\!]\sigma = \mathit{True} \\ \sigma, & \text{falls } [\![b]\!]\sigma = \mathit{False} \end{cases}$$

ist.

Satz: $F : D \to D$ ist stetig.

Beweis:

- Wir zeigen zunächst die Monotonie von F. Sei $X_1 \sqsubseteq X_2$ mit $X_1, X_2 \in D$.
  - Fall 1: $[\![b]\!]\sigma = \mathit{False}$
    $\implies \sigma \in dom(F\,X_1) \cap dom(F\,X_2)$ und $F\,X_1\,\sigma = \sigma = F\,X_2\,\sigma$.
  - Fall 2: $[\![b]\!]\sigma = \mathit{True}$
    Sei $\sigma_1 \mapsto \sigma_2 \in F\,X_1$.
    Also gilt $\exists \sigma : \sigma_1 \mapsto \sigma \in g$ und $\sigma \mapsto \sigma_2 \in X_1$.
    Wegen $X_1 \sqsubseteq X_2$ gilt auch $\exists \sigma : \sigma_1 \mapsto \sigma \in g$ und $\sigma \mapsto \sigma_2 \in X_2$.
    Damit folgt $\sigma_1 \mapsto \sigma_2 \in F\,X_2$.

Aus $X_1 \sqsubseteq X_2$ folgt also $F\,X_1 \sqsubseteq F\,X_2$.

Für jede Kette $X_0 \sqsubseteq X_1 \sqsubseteq X_2 \sqsubseteq \cdots$ aus D mit kleinster oberer Schranke $\bigsqcup_{i\geq 0} X_i$ gilt damit $\forall i \geq 0 : F\,X_i \sqsubseteq F(\bigsqcup_{i\geq 0} X_i)$. Also gilt $\bigsqcup_{i\geq 0} F\,X_i \sqsubseteq F(\bigsqcup_{i\geq 0} X_i)$.

- Sei nun $X_0 \sqsubseteq X_1 \sqsubseteq X_2 \sqsubseteq \cdots$ aufsteigende Kette aus D mit kleinster oberer Schranke $X := \bigsqcup_{i\geq 0} X_i$.

  Sei $F\,X = \{\sigma \mapsto \sigma \mid [\![b]\!]\sigma = \mathit{False}\} \cup S$, wobei
  $S = \{\sigma_1 \mapsto \sigma_2 \mid [\![b]\!]\sigma_1 = \mathit{True} \text{ und } \sigma_2 = X(g\,\sigma_1)\}$ und
  $F\,X_i = \{\sigma \mapsto \sigma \mid [\![b]\!]\sigma = \mathit{False}\} \cup S_i$, wobei
  $S_i = \{\sigma_1 \mapsto \sigma_2 \mid [\![b]\!]\sigma_1 = \mathit{True} \text{ und } \sigma_2 = X_i(g\,\sigma_1)\}$.

  Offenbar genügt es $S \subseteq \bigcup_{i\geq 0} S_i$ zu zeigen:
  Sei $\sigma_1 \mapsto \sigma_2 \in S$.
  Dann gilt $\exists \sigma : \sigma_1 \mapsto \sigma \in g$ und $\sigma \mapsto \sigma_2 \in X$.
  Für ein i gilt $\exists \sigma : \sigma_1 \mapsto \sigma \in g$ und $\sigma \mapsto \sigma_2 \in X_i$.
  Wegen $[\![b]\!]\sigma_1 = \mathit{True}$ folgt $\sigma_1 \mapsto \sigma_2 \in S_i$.

  Also gilt $F(\bigsqcup_{i\geq 0} X_i) \sqsubseteq \bigsqcup_{i\geq 0} F\,X_i$.

Da $(D, \sqsubseteq)$ eine vollständige partielle Ordnung und F eine stetige Funktion auf D ist, besitzt F nach dem Satz von KLEENE einen *kleinsten* Fixpunkt.
Die Gleichung $X = F\,X$ von oben besitzt somit nicht nur eine Lösung, sondern sogar eine *kleinste* Lösung $X = \bigsqcup_{i\geq 0} F^i \bot$ mit $\bot = \emptyset$.

Satz: Folgenden drei Aussagen sind äquivalent:

(1) $p \mid \sigma_1 \to^* \varepsilon \mid \sigma_2$,

(2) $\vdash p \mid \sigma_1 \mapsto \sigma_2$,

(3) $\sigma_1 \mapsto \sigma_2 \in [\![p]\!]$.

Beweis:

- (1) $\Leftrightarrow$ (2): Wurde bereits in Abschnitt 2.4 gezeigt.
- (2) $\Rightarrow$ (3): Induktion nach der Tiefe d eines Beweisbaums B.
  $d = 0 : B$ besteht nur aus einem Blatt.
  - Für $p \equiv \varepsilon$ gilt $\vdash p \mid \sigma \mapsto \sigma$ (Axiom 3) und $[\![p]\!]\,\sigma = \sigma$. $\surd$
  - Für $p \equiv x := e;$ gilt $\vdash p \mid \sigma \mapsto \sigma \oplus \{x \mapsto [\![e]\!]\sigma\}$ (Axiom 4) und $[\![p]\!]\,\sigma = \sigma \oplus \{x \mapsto [\![e]\!]\sigma\}$. $\surd$

  $d > 0$: Fallunterscheidung nach der letzten angewandten Regel.
  Beachte, daß die Prämissen der jeweiligen Regel Beweise mit Tiefe $< d$ haben.

- $p \equiv s; p'$. Die entsprechende Regel der Big-Step-Semantik lautet

  $$\text{Conc} \quad \frac{s; \mid \sigma_1 \mapsto \sigma \qquad p' \mid \sigma \mapsto \sigma_2}{s; p' \mid \sigma_1 \mapsto \sigma_2}.$$

  Nach Induktionsvoraussetzung gilt $[\![s;]\!]\,\sigma_1 = \sigma$ und $[\![p']\!]\,\sigma = \sigma_2$.
  $\Longrightarrow [\![s; p']\!]\,\sigma_1 = [\![p']\!]\,([\![s;]\!]\,\sigma_1) = [\![p']\!]\,\sigma = \sigma_2$.
- $p \equiv$ `while` b `do` p' `end`. Die entsprechenden Regeln der Big-Step-Semantik lauten

  $$\text{While1} \quad \frac{[\![b]\!]\,\sigma_1 = \mathit{True} \qquad p' \mid \sigma_1 \mapsto \sigma \qquad p; \mid \sigma \mapsto \sigma_2}{p; \mid \sigma_1 \mapsto \sigma_2}$$

  $$\text{While2} \quad \frac{[\![b]\!]\,\sigma_1 = \mathit{False}}{\texttt{while}\ b\ \texttt{do}\ p'\ \texttt{end}; \mid \sigma_1 \mapsto \sigma_1}$$

  Nach Induktionsvoraussetzung gilt $[\![p']\!]\,\sigma_1 = \sigma$ und $[\![p;]\!]\,\sigma = \sigma_2$.
  $$\Longrightarrow [\![p;]\!]\,\sigma_1 = F\,[\![p;]\!]\,\sigma_1 = \begin{cases} [\![p;]\!]\,([\![p']\!]\,\sigma_1) = [\![p;]\!]\,\sigma = \sigma_2, & \text{falls } [\![b]\!]\sigma_1 = \mathit{True} \\ \sigma_1, & \text{falls } [\![b]\!]\sigma_1 = \mathit{False} \end{cases}$$
- $p \equiv$ `if` b `then` p_1 `else` p_2 `end`.
  Dieser Fall ist dem geneigten Leser zur Übung empfohlen.

- (3) $\Rightarrow$ (2): Strukturelle Induktion.

  - $p \equiv \varepsilon$: Es gilt $[\![p]\!]\,\sigma = \sigma$ und $\vdash p \mid \sigma \mapsto \sigma$. $\surd$.
  - $p \equiv x := e;$: Es gilt $[\![p]\!]\,\sigma = \sigma \oplus \{x \mapsto [\![e]\!]\sigma\}$ und $\vdash p \mid \sigma \mapsto \sigma \oplus \{x \mapsto [\![e]\!]\sigma\}$. $\surd$.
  - $p \equiv s; p'$: Für $\sigma_1 \mapsto \sigma_2 \in [\![s; p']\!]$ gilt $[\![p']\!]\,([\![s;]\!]\,\sigma_1) = \sigma_2$.
    Also gibt es eine Variablenbelegung σ mit $\sigma = [\![s;]\!]\,\sigma_1$ und $\sigma_2 = [\![p']\!]\,\sigma$.
    Nach Induktionsvoraussetzung gilt $\vdash s; \mid \sigma_1 \mapsto \sigma$ und $\vdash p' \mid \sigma \mapsto \sigma_2$.
    Mit der Conc-Regel folgt $\vdash p \mid \sigma_1 \mapsto \sigma_2$.
  - $p \equiv$ `while` b `do` p' `end`: Es gilt $[\![p;]\!] = \bigsqcup_{i \geq 0} [\![p;]\!]^i$, wobei $[\![p;]\!]^i = F^i \bot$.

    Induktion nach der Anzahl der Iterationen i, wobei $\sigma_1 \mapsto \sigma_2 \in [\![p;]\!]^i$:

    $i = 0$: Es gilt $[\![b]\!]\,\sigma_1 = \mathit{False}$, da der Rumpf p' der Schleife in diesem Fall nicht ausgeführt wurde. Also ist $[\![p;]\!]^0\,\sigma_1 = \sigma_1$, und damit $\vdash p \mid \sigma_1 \mapsto \sigma_1$.

    $i > 0$: Nun gilt $[\![b]\!]\,\sigma_1 = \mathit{True}$.
    $\Longrightarrow [\![p;]\!]^i\,\sigma_1 = F^i \bot\,\sigma_1 = F\,(F^{i-1}\bot)\,\sigma_1 = (F^{i-1}\bot)\,(g\,\sigma_1) = [\![p;]\!]^{i-1}\,([\![p']\!]\,\sigma_1)$.
    Also gibt es eine Variablenbelegung σ mit $[\![p']\!]\,\sigma_1 = \sigma$ und $[\![p;]\!]^{i-1}\,\sigma = \sigma_2$.
    Nach Induktionsvoraussetzung gilt $\vdash p' \mid \sigma_1 \mapsto \sigma$ und $\vdash p; \mid \sigma \mapsto \sigma_2$.
    Mit der While1-Regel folgt daraus $\vdash p; \mid \sigma_1 \mapsto \sigma_2$.
  - $p \equiv$ `if` b `then` p_1 `else` p_2 `end`:
    Dieser Fall ist wieder zur Übung empfohlen. □

Fazit: Für unsere Beispielprogrammiersprache sind alle drei Semantik-Ansätze äquivalent.

Wieso betrachten wir dann überhaupt alle drei Ansätze ?

a) Es gibt Programmiersprachen in denen diese drei Ansätze eben nicht äquivalent sind!

b) Jeder dieser Ansätze besitzt Stärken und Schwächen:

	small-step	big-step	denotationell
Vorteil	am generellsten deterministisch	syntaxgerichtet kompositionell (Verifikation)	syntaxgerichtet Übersetzung in funkt. Progr.
Nachteil	nicht-kompositionell	nicht-deterministisch (Rekonstruktion eines Beweisbaums)	nicht "explizit"

Kapitel 4

SML - Eine funktionale Programmiersprache

Funktionale Programmiersprachen bieten sich für eine elegante Implementierung von denotationeller Semantik und Small-Step-Semantik an.

Um eine Implementierung in Standard-ML (SML) anzugeben, betrachten wir zunächst die wesentlichen Merkmale von SML.

4.1 Werte in SML

In imperativen Sprachen bezeichnen Variablen Speicherplätze, die einen Wert enthalten.
Wir notieren dies auch als `x` $\longmapsto \boxed{\textcircled{s}}$, wobei `x` einen Variablennamen, $\boxed{}$ den Speicherplatz und $\textcircled{s}$ den gespeicherten Wert bezeichnet.

In funktionalen Sprachen bezeichnet eine Variable einen Wert oder einen Ausdruck, der einen Wert berechnet (dynamische Konstante).
Wir notieren dies als `x` $\longmapsto \textcircled{s}$.

In SML verknüpfen wir eine Variable auf folgende Weise mit einem Wert (Ausdruck):
`-val x = 1+3;`
Das Laufzeitsystem antortet mit *val x = 4 : int* (der Typ wird immer mit ausgegeben).

Auch Funktionen sind Werte:
```
-fun fac x = if x<= 1 then 1 else x*fac(x-1);
```
Das Laufzeitsystem antortet nun mit *val fac = fn : int -> int* .

Für die Eingabe
`-fac 3;`
erhalten wir als Ausgabe *val it = 6 : int*.

Funktionen werden auch als "First-class Citizens" bezeichnet.

Wir können die Funktionskomposition $(g \circ g)\, x$ in SML ausdrücken als
```
-fun two g x = g (g x);
```

Systemantwort: *val two = fn : ('a -> 'a) -> 'a -> 'a .*
'a ist hier eine sog. *Typvariable*, die für einen beliebigen Typ (etwa *int* oder *real*) stehen kann (SML kennt auch den "eingebauten" Operator `o` ⇝ `fun two g = g o g;`).

Für die Eingabe
`-two fac 3;`
erhalten wir für die Integer-Funktion `fac` als Ausgabe *val it = 720 : int.*

`-fun divby2 x = x/2.0;`
Systemantwort *val divby2 = fn : real -> real .*

Für die Eingabe
`-two divby2 1.0;`
erhalten wir diesmal für die Real-Funktion `divby2` als Ausgabe *val it = 0.25 : real.*

Erhält `two` als Argument nur eine Funktion, so ist das Ergebnis wieder eine Funktion:
`-two fac;` ⇝ Systemantwort *val it = fn : int -> int*
`-two divby2;` ⇝ Systemantwort *val it = fn : real -> real*

Da Funktionen auch Werte sind können wir auch schreiben
`-val divby2 = (fn x => x/2.0);` ⇝ Systemantwort *val divby2 = fn : real -> real.*

`-two (two (fn x => x + 1)) 7;` als Eingabe liefert als Ausgabe *val it = 11 : int.*

`-val four = two two (fn x => x + 1);` als Eingabe liefert *val four = fn : int -> int.*

`-four 7;` ergibt wieder *val it = 11 : int.*

4.2 Listen in SML

Eine Liste ist entweder leer (*leere Liste* `[]`) oder sie besteht aus einem Kopf `x` und einer Restliste `xs`, verbunden durch den Konstruktor '`::`'.

`-val l = 1::(2::(3::[]));` ergibt die Systemantwort *val l = [1,2,3] : int. list.*

Wir können diese Liste auch durch folgende Schreibweise erzeugen:
`-val l = [1,2,3];`.

Die Funktion `append` verbindet zwei Listen:
```
fun append [] ys       = ys
  | append (x::xs) ys = x :: (append xs ys);
```

SML verfügt auch über einen eingebauten Infix-Operator `@` zum Verbinden zweier Listen:
`-val l' = l @ [4,5];` ⇝ *val l' = [1,2,3,4,5] : int list.*
Das gleiche Ergebnis liefert `val l' = append l [4,5];`.

4.3 Datentypen in SML

Nachdem wir nun Werte und Funktionen als besondere Werte kennengelernt haben, werfen wir einen Blick auf die Konstruktion von Datentypen in SML, um eine geeignete Repräsentation für Syntaxbäume zu finden.

`-datatype v = X | Y | Z;` definiert einen Aufzählungstyp `v` mit Werten `X`, `Y` und `Z`.

`-val x = X;` als Eingabe liefert *val x = X : v.*

Wenn wir annehmen, daß der neue Datentyp `v` Variablen definiert, können wir mit einer Funktion `ass` solchen Variablen Werte zuordnen (Variablenbelegung):

```
fun ass X =  3
  | ass Y = ~4  (* ~ entspricht dem unären Operator '-' *)
  | ass Z =  0;
```

$\rightsquigarrow$ *val ass = fn : v -> int*

Die Definition von `ass` ist ein Beispiel dafür, wie man in SML Funktionen durch *Fallunterscheidung* definieren kann.

Wir definieren rekursiv folgenden Datentyp `e` für Ausdrücke:

```
datatype e =  Int  of int
            | Var  of v
            | Plus of e * e
            | Mult of e * e
            | Neg  of e;
```

Wir bezeichen `Int`, `Var`, `Plus`, `Mult` und `Neg` auch als *(Datentyp-)Konstruktoren.*

Der Ausdruck `e * e` besagt hier, daß der Konstruktor ein Tupel von Ausdrücken `e` als Argument erwartet.

Der Ausdruck $3 * X + 1$ kann nun als `Plus(Mult(Int 3,Var X),Int 1)` dargestellt werden
$\rightsquigarrow$ *val it = Plus (Mult (Int 3,Var X),Int 1) : e.*

Die Funktion `update` erzeugt aus einer Variablenbelegung `ass` eine neue Variablenbelegung, in der der Wert von `x` auf `d` gesetzt wurde.

```
fun update x d ass y =
      if y = x then d else (ass y);
```

$\rightsquigarrow$ *val update = fn : 'a -> 'b -> ('a -> 'b) -> 'a -> 'b*

```
update X 7 ass;
```

$\rightsquigarrow$ *val it = fn : v -> int*

Die folgende Funktion `semE` berechnet nun die Semantik eines Ausdrucks für eine Varaiblenbelegung `ass`:

```
fun semE (Int n) ass = n
  | semE (Var x) ass = ass x
  | semE (Plus(e1,e2)) ass = (semE e1 ass) + (semE e2 ass)
  | semE (Mult(e1,e2)) ass = (semE e1 ass) * (semE e2 ass)
  | semE (Neg e) ass = ~(semE e ass);
```

$\rightsquigarrow$ *val it = fn : e -> (v -> int) -> int*

Für die Variablenbelegung von oben erzeugt
`semE (Neg (Plus(Var X,Int ~19))) ass;` die Systemantwort
val it = 16 : int.

Analog zu `e` definieren wir einen Datentyp `b` für Boolesche Ausdrücke zusammen mit einer Funktion `semB` mit *val semB = fn : b -> (v -> int) -> bool*.

Damit erhalten wir das folgende funktionale Programm:

```
datatype v = X | Y | Z     (* Variablen *)

datatype e =  Int  of int  (* arithmetische Ausdruecke *)
            | Var  of v
            | Plus of e*e
            | Mult of e*e
            | Neg  of e

datatype b =  T | F        (* Boolesche Ausdruecke *)
            | Not of b
            | And of b*b
            | Or  of b*b
            | Le  of e*e
            | Eq  of e*e

datatype s =  Ass   of v*e   (* Statements *)
            | If    of b*(s list)*(s list)
            | While of b*(s list)


fun update x d ass y = if y = x then d else (ass y)

(* Semantik fuer arithmetische Ausdruecke *)
fun semE (Int n) ass       = n
  | semE (Var x) ass       = ass x
  | semE (Plus(e1,e2))     = (semE e1 ass) + (semE e2 ass)
  | semE (Mult(e1,e2))     = (semE e1 ass) * (semE e2 ass)
  | semE (Neg e) ass       = ~(semE e ass)

(* Semantik fuer Boolesche Ausdruecke *)
fun semB T ass            = true
  | semB F ass            = false
  | semB (Not b) ass      = not (semB b ass)
  | semB (And(b1,b2)) ass = (semB b2 ass) andalso (semB b2 ass)
  | semB (Or(b1,b2)) ass  = (semB b2 ass) orelse (semB b2 ass)
  | semB (Le(e1,e2)) ass  = (semE e1 ass) < (semE e2 ass)
  | semB (Eq(e1,e2)) ass  = (semE e1 ass) = (semE e2 ass)


(* denotationelle Semantik fuer Statements *)
fun semS (Ass (x,e)) ass     = update x (semE e ass) ass
  | semS (If (b,p1,p2)) ass  = if (semB b ass) then
                                 (sem p1 ass)
                               else
                                 (sem p2 ass)
  | semS (While (b,p)) ass   = if (semB b ass) then
                                 semS (While (b,p)) (sem p ass)
```

```
                              else
                                ass

(* denotationelle Semantik fuer ein Programm = Liste von Statements *)
and sem [] ass      = ass
  | sem (s::p) ass  = sem p (semS s ass)


(* ein Beispielprogramm *)
val example =
      Ass (Y,Int 1) ::                      (* y := 1;              *)
      While (Not (Le (Var X, Int 1)),       (* while (¬(x<1)) do    *)
             Ass (Y, Mult(Var X, Var Y)) ::        (* y := x*y;    *)
             Ass (X, Plus(Var X, Neg (Int 1))) ::  (* x := x+(-1); *)
             []) ::                         (* end;                 *)
      []

fun ass X = ~2
  | ass Y = 0

fun test d = sem example (update X d ass) Y


(* Small-Step-Semantik fuer Statements *)
fun small [] ass         = ass
  | small (Ass (x,e)::p) ass =
             let val ass' = update x (semE e ass) ass
             in
               small p ass'
             end
  | small (If (b,p1,p2) :: p) ass =
             if (semB b ass) then
               small (p1 @ p) ass
             else
               small (p2 @ p) ass
  | small (While (b,p1) :: p) ass =
             if (semB b ass) then
               let val p' = p1 @ (While (b,p1) :: p)
               in
                 small p'  ass
               end
             else
               small p ass

fun test' d  = small example (update X d ass) Y
```

Kapitel 5

Parallele Komposition

Wir erweitern nun die Syntax unserer Programmiersprache um das Parallelkonstrukt $(p_1 \| p_2)$:

$$\begin{array}{rcl} s & ::= & x := e \\ & | & \vdots \\ & | & (p_1 \| p_2) \end{array}$$

Informal bedeutet dies, daß die Statementfolge p_1 parallel zu p_2 ausgeführt werden soll.

Wir sprechen bei solchen parallel ausgeführten Programmen auch von *Prozessen*.

Beispiel: Für $\sigma = \{x \mapsto _, y \mapsto _\}$ gilt $(x := 1; \| y := 2;); | \sigma \quad \rightarrow^* \quad \varepsilon | \{x \mapsto 1, y \mapsto 2\}$

In den folgenden Abschnitten wenden wir uns den verschiedenen Problemen der parallelen Ausführung solcher Prozesse zu.

5.1 Ausführungsreihenfolge

Wir treffen folgende Annahmen:

- Einzelne Zuweisungen und die Auswertungen von Bedingungen geschehen "atomar".
- Die Ausführung der atomaren Schritte in p_1 und p_2 kann beliebig ineinander verschränkt sein.

Beispiel:

$$(x := 2; \| x := 3;); | \sigma$$

$$\begin{array}{ccc} & \swarrow & \\ x := 3; | \{x \mapsto 2, y \mapsto _\} & & (*) \quad x := 2; | \{x \mapsto 3, y \mapsto _\} \\ \downarrow & & \downarrow \\ \varepsilon | \{x \mapsto 3, y \mapsto _\} & & \varepsilon | \{x \mapsto 2, y \mapsto _\} \end{array}$$

Dieses Beispiel macht deutlich, daß, aufgrund von mehreren möglichen Berechnungen (je nachdem welcher Schritt bei $(*)$ zuletzt ausgeführt wird), unterschiedliche Ergebnisse auftreten können.

Wir erweitern die operationelle Small-Step-Semantik um Regeln für die Anwendung des "$\|$"-Konstrukts:

- $(\varepsilon \,\|\, p);\,|\,\sigma \;\rightarrow\; p\,|\,\sigma$ und $(p \,\|\, \varepsilon);\,|\,\sigma \;\rightarrow\; p\,|\,\sigma$
- $(p_1 \,\|\, p);\,|\,\sigma_1 \;\rightarrow\; (p_2 \,\|\, p);\,|\,\sigma_2$, falls $p_1\,|\,\sigma_1 \;\rightarrow\; p_2\,|\,\sigma_2$
- $(p \,\|\, p_1);\,|\,\sigma_1 \;\rightarrow\; (p \,\|\, p_2);\,|\,\sigma_2$, falls $p_1\,|\,\sigma_1 \;\rightarrow\; p_2\,|\,\sigma_2$

Beispiel:

$$p \;\equiv\; (\overset{s_1}{(x := 1;}\; \overset{s_2}{y := x + 3;});\;\|\;(\overset{t_1}{y := 2;}\; \overset{t_2}{x := y - 2;}));$$

Für die Ausführungsreihenfolge $s_1 \rightarrow s_2 \rightarrow t_1 \rightarrow t_2$ erhalten wir

$$\begin{array}{rr|l}
 & ((x := 1;\; y := x + 3;) \,\|\, (y := 2;\; x := y - 2;)); & \{x \mapsto _,\, y \mapsto _\} \\
\rightarrow & (y := x + 3;) \,\|\, (y := 2;\; x := y - 2;)); & \{x \mapsto 1,\, y \mapsto _\} \\
\rightarrow & (\varepsilon \,\|\, (y := 2;\; x := y - 2;)); & \{x \mapsto 1,\, y \mapsto 4\} \\
\rightarrow & y := 2;\; x := y - 2; & \{x \mapsto 1,\, y \mapsto 4\} \\
\rightarrow & x := y - 2; & \{x \mapsto 1,\, y \mapsto 2\} \\
\rightarrow & \varepsilon & \{x \mapsto 0,\, y \mapsto 2\}
\end{array}$$

Alle anderen möglichen Programmausführungen liefern:

Folge	σ
$s_1 \rightarrow t_1 \rightarrow s_2 \rightarrow t_2$	$\{x \mapsto 2,\, y \mapsto 4\}$
$s_1 \rightarrow t_1 \rightarrow t_2 \rightarrow s_2$	$\{x \mapsto 0,\, y \mapsto 3\}$
$t_1 \rightarrow s_1 \rightarrow s_2 \rightarrow t_2$	$\{x \mapsto 2,\, y \mapsto 4\}$
$t_1 \rightarrow s_1 \rightarrow t_2 \rightarrow s_2$	$\{x \mapsto 0,\, y \mapsto 3\}$
$t_1 \rightarrow t_2 \rightarrow s_1 \rightarrow s_2$	$\{x \mapsto 1,\, y \mapsto 4\}$

Wir erhalten also vier mögliche Ergebnisse für dieses Programm(fragment).

Folgerung: Die Semantik von $(p_1 \,\|\, p_2)$ läßt sich nicht einfach auf die Eigenschaften von p_1 und p_2 zurückführen!

5.2 Fairness

Wie folgendes Programmbeispiel zeigt, kann ein Prozeß durch einen parallelen Prozeß blockiert werden:

$$p \;\equiv\; x := 1;\; (\underbrace{(\texttt{while}\,\neg(x = 0)\,\texttt{do end};\; y := 7;)}_{p_1} \,\|\, \underbrace{x := 0;}_{p_2});$$

Die Idee ist hier, daß der Prozeß p_1 solange wartet, bis Prozeß p_2 die Variable x auf 0 setzt (*busy waiting*).

Wird nach der Zuweisung $x := 1$ der Kontrollfluß immer mit Prozeß p_1 fortgesetzt, dann wird Prozeß p_2 nie ausgeführt und x wird niemals auf 0 gesetzt.

Dieses Phänomen bezeichnet man auch als *Starvation* (Verhungern) eines Prozesses.
Dies ist u. U. nicht nur ein Problem der Semantik, sondern auch der Implementierung: Eventuell steht allen Prozessen des Systems nur ein Prozessor zur Verfügung. Daher müssen alle Prozesse sequentiell ausgeführt werden, d. h., daß $(p_1 \| p_2)$ als Folge $p_1\ p_2$ implementiert wird.

Das obige Programm terminiert nur dann garantiert (mit $\sigma = \{x \mapsto 0,\ y \mapsto 7\}$), wenn wir die Zusatzannahme der *Fairness* treffen.
Eine faire Programmausführung verbietet bestimmte nicht-terminierenden – sogenannte *unfairen* – Berechnungen.

Als Konsequenz dieser Überlegungen unterstreichen wir nun für jeden Berechnungsschritt zusätzlich das Statement, an dem dieser Schritt *ansetzt*.

Beispiele:

$$\underline{x := exp;}\ p \mid \sigma_1 \ \rightarrow\ p \mid \sigma_2$$

$$(p_1 \| p_2 \| \underline{\texttt{while}\ (x = 0)\ \texttt{do end};} \| \cdots);\ \mid \{x \mapsto 0, \ldots\}$$
$$\rightarrow\ (p_1 \| p_2 \| \texttt{while}\ (x = 0)\ \texttt{do end};\ \| \cdots);\ \mid \{x \mapsto 0, \ldots\}$$

$$\underline{(\varepsilon \| p);} \mid \sigma \ \rightarrow\ p \mid \sigma$$

Definition: Eine Folge von Berechnungsschritten

$$p_1 \mid \sigma_1 \ \rightarrow\ p_2 \mid \sigma_2 \ \rightarrow\ \cdots \ \rightarrow\ p_k \mid \sigma_k \ \rightarrow\ \cdots$$

heißt *unfair*, falls für alle $j \geq k$ gilt:

- $p_j \ \equiv\ (p'_j \| p);\ q$ bzw. $p_j \ \equiv\ (p \| p'_j);\ q$ und
- der Berechnungsschritt $p_j \ \rightarrow\ p_{j+1}$ setzt stets in p'_j an.

Das Verbieten unfairer Berechnungen garantiert Terminierung.

Problem: Die Realisierung von Fairness auf einem Monoprozessor erfordert einen aufwendigen Scheduling-Mechanismus, wie etwa die Unterbrechung laufender Prozesse bei jeder bedingten Verzweigung (Occam). Daher wird dies oft nicht realisiert.

Kapitel 6

Verifikation

Unsere bisherigen Definitionen von Semantik gaben Auskunft über das Verhalten eines Programms für eine bestimmte **feste** Ausgangsvariablenbelegung.

In der Regel wird ein Programm aber für **verschiedene** Eingabewerte aufgerufen.

Wir wollen nun gerne Aussagen der folgenden Form beweisen:

> Falls vor der Ausführung eines Programms p für eine Variable x stets $x \leq 1$ gilt, dann gilt nach der Ausführung von p immer $x > 17$ (sofern p terminiert).

Wir schreiben dafür $\{x \leq 1\}\ p\ \{x > 17\}$ (HOARE-Tripel).

Wir bezeichnen die Aussage A in $\{A\}\ p\ \{B\}$ als *Precondition* und B als *Postcondition*. A und B heißen auch *Zusicherungen*.

Die Precondition A beschreibt dabei die Menge aller Variablenbelegungen σ, die A *erfüllen*. Symbolisch $\sigma \models A$. Wir schreiben $\models A$, falls A für jede Variablenbelegung gilt ($\forall \sigma : \sigma \models A$).

Im Beispiel von oben gilt $\sigma \models A \iff (\sigma\, x) \leq 1$.

Insbesondere gilt:

- $\sigma \models \mathit{True}$ für jede Variablenbelegung σ.
- $\sigma \models \mathit{False}$ für keine Variablenbelegung σ.
- $\sigma \models b \iff [\![b]\!]\sigma = \mathit{True}$ für einen BOOLEschen Ausdruck b.

Definition: Die Aussage $\{A\}\ p\ \{B\}$ heißt *(partiell) korrekt*, falls für alle $\sigma_1 \models A$ gilt:
$p \mid \sigma_1 \mapsto \sigma_2 \implies \sigma_2 \models B$.
Wir schreiben dann $\models \{A\}\ p\ \{B\}$.

Achtung: Für den Fall, daß p nicht terminiert, machen wir keine Aussagen.

Beispiele:

- Für die Endlosschleife $p \equiv$ `while` (True) `do` ε `end`; gilt $\{\mathit{True}\}\ p\ \{\mathit{False}\}$ und damit sogar $\{\mathit{True}\}\ p\ \{A\}$ für beliebige Aussagen A.

- Für $p \equiv$ `while` $(x \leq 2)$ `do` ε `end`; gilt $\{True\}\ p\ \{x > 2\}$, da p nur für $x > 2$ terminiert.

Für eine Aussage A, die eine Variable x enthält, bezeichnen wir die Aussage, die durch uniformes Ersetzen von x durch einen Ausdruck e entsteht, als $A[e/x]$.

Wir geben nun ein Beweissystem für partielle Korrektheit an. Ein solches System heißt auch HOARE-*Logik*.

Definition:

Axiome des Systems sind

1. Skip $\{A\}\ \varepsilon\ \{A\}$ und
2. Assign $\{A[e/x]\}\ x := e;\ \{A\}$.

Regeln:

- Conc $\dfrac{\{A\}\ s;\ \{B\} \qquad \{B\}\ p\ \{C\}}{\{A\}\ s;\ p\ \{C\}}$
- If $\dfrac{\{A \wedge b\}\ p_1\ \{B\} \quad \{A \wedge \neg b\}\ p_2\ \{B\}}{\{A\}\ \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end};\ \{B\}}$
- While $\dfrac{\{A \wedge b\}\ p\ \{A\}}{\{A\}\ \texttt{while}\ b\ \texttt{do}\ p\ \texttt{end};\ \{A \wedge \neg b\}}$
- Weak $\dfrac{\models A \Rightarrow A' \qquad \{A'\}\ p\ \{B'\} \qquad \models B' \Rightarrow B}{\{A\}\ p\ \{B\}}$

Dabei gilt $\models A \Rightarrow B$ genau dann, wenn für alle σ gilt: $\sigma \models A \implies \sigma \models B$
(d. h. $\{\sigma \mid \sigma \models A\} \subseteq \{\sigma \mid \sigma \models B\}$).

Die Weak-Regel bedeutet, daß die Precondition stets "verstärkt" und die Postcondition "abgeschwächt" werden kann.

Die Aussage A der While-Regel bezeichnet man auch als *Schleifeninvariante* (sie ist notorisch schwer zu bestimmen).

Gibt es in der HOARE-Logik einen Beweis für $\{A\}\ p\ \{B\}$, dann schreiben wir $\vdash \{A\}\ p\ \{B\}$.

Folgender Satz garantiert uns, daß mittels HOARE-Logik keine falschen Aussagen herleitbar sind.

Satz: $\vdash \{A\}\ p\ \{B\}$ impliziert $\models \{A\}\ p\ \{B\}$.

Beweis: Induktion über einen Beweisbaum t für p

Besteht t nur aus einem Blatt, dann unterscheiden wir zwei Fälle:

- $p \equiv \varepsilon$: $\quad\surd$
- $p \equiv x := e;$: Es gelte $\sigma_1 \models A[e/x]$ und es sei $\sigma_2 = \sigma_1 \oplus \{x \mapsto [\![e]\!]\sigma_1\}$.

  Zu zeigen $\sigma_2 \models A$:

1. Sei e' ein Integerausdruck.
   Man zeigt mittels struktureller Induktion, daß $[\![e']\!]\sigma_2 = [\![e'[e/x]]\!]\sigma_1$ gilt.
   Z. B.: $e' \equiv y$ mit $y \neq x$: $y[e/x] = y$
   $e' \equiv x$: $[\![x]\!]\sigma_2 = [\![e]\!]\sigma_1 = [\![x[e/x]]\!]\sigma_1$
   usw.
2. $[\![e_1 \Box e_2]\!]\sigma_2 = [\![e_1[e/x] \Box e_2[e/x]]\!]\sigma_1$ für $\Box \in \{=, <\}$.
3. Weitere BOOLEsche Ausdrücke mit $\Box \in \{\neg, \wedge, \vee\}$ analog.

Besteht t nicht nur aus einem Blatt, dann unterscheiden wir nach der zuletzt angewandten Regel:

- Weak $\dfrac{\models A \Rightarrow A' \qquad \{A'\}\ p\ \{B'\} \qquad \models B' \Rightarrow B}{\{A\}\ p\ \{B\}}$

  Es gelte $\vdash \{A'\}\ p\ \{B'\}$.
  Sei $\sigma_1 \models A$. Wegen $\models A \Rightarrow A'$ gilt dann auch $\sigma_1 \models A'$.
  Sei nun $p \mid \sigma_1 \mapsto \sigma_2$.
  Nach IV. gilt dann $\sigma_2 \models B'$ und damit auch $\sigma_2 \models B$ (wegen $\models B' \Rightarrow B$).

- Conc $\dfrac{\{A\}\ s;\ \{B\} \qquad \{B\}\ p\ \{C\}}{\{A\}\ s;\ p\ \{C\}}$

  Es gelte $\{A\}\ s;\ \{B\}$ und $\{B\}\ p\ \{C\}$.
  Es sei $s;\ p \mid \sigma_1 \mapsto \sigma_2$ mit $\sigma_1 \models A$.
  Dann gibt es ein σ mit $s; \mid \sigma_1 \mapsto \sigma$ und $p \mid \sigma \mapsto \sigma_2$.
  Nach IV. für $\{A\}\ s;\ \{B\}$ gilt $\sigma \models B$. Nach IV. für $\{B\}\ p\ \{C\}$ gilt schließlich $\sigma_2 \models C$.

- If $\dfrac{\{A \wedge b\}\ p_1\ \{B\} \quad \{A \wedge \neg b\}\ p_2\ \{B\}}{\{A\}\ \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end};\ \{B\}}$

  Es gelte $\{A \wedge b\}\ p_1\ \{B\}$ und $\{A \wedge \neg b\}\ p_2\ \{B\}$.
  Es sei $\texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; \mid \sigma_1 \mapsto \sigma_2$ mit $\sigma_1 \models A$.
  - **Fall 1** $\sigma_1 \models b$: Insbesondere gilt $\sigma_1 \models A \wedge b$.
    Wegen $[\![b]\!]\sigma_1 = \mathit{True}$ gilt $p_1 \mid \sigma_1 \mapsto \sigma_2$.
    Nach IV. für $\{A \wedge b\}\ p_1\ \{B\}$ gilt $\sigma_2 \models B$.
  - **Fall 2** $\sigma_1 \models \neg b$: Man schließt mit $\sigma_1 \models A \wedge \neg b$ analog $\sigma_2 \models B$.

- While $\dfrac{\{A \wedge b\}\ p\ \{A\}}{\{A\}\ \texttt{while}\ b\ \texttt{do}\ p\ \texttt{end};\ \{A \wedge \neg b\}}$

  Es gelte $\{A \wedge b\}\ p\ \{A\}$.
  Es sei $\texttt{while}\ b\ \texttt{do}\ p\ \texttt{end}; \mid \sigma_1 \mapsto \sigma_2$ $(*)$ mit $\sigma_1 \models A$.
  Wir zeigen $\sigma_2 \models A \wedge \neg B$ durch Induktion nach der Tiefe d eines Beweisbaums für $(*)$:
  - $d = 0$: $[\![b]\!]\sigma_1 = \mathit{False} \implies \sigma_1 \models \neg b$ und $\sigma_1 \equiv \sigma_2$ $\surd$
  - $d > 0$: Für ein σ gilt dann (a) $\sigma_1 \models b$
    (b) $p \mid \sigma_1 \mapsto \sigma$ und
    (c) $\texttt{while}\ b\ \texttt{do}\ p\ \texttt{end}; \mid \sigma \mapsto \sigma_2$ mit kürzerem Beweis
    Wegen (a) gilt $\sigma_1 \models A \wedge b$. Aus (b) und der IV. folgt $\sigma \models A$.
    Aus (c) und der IV. folgt $\sigma_2 \models A \wedge \neg b$. $\Box$

Beispiel: $p \equiv$ `while` $(x \neq 0)$ `do` $y := y * x;\ x := x - 1;$ `end`;

Wir zeigen $\vdash \{A\}\ p\ \{B\}$ für die Precondition $A \equiv (x = n) \wedge (y = 1)$ und die Postcondition $B \equiv (y = n * \cdots * 1)$ (d. h. $y = n!$).

Um die While-Regel anwenden zu können, müssen wir die Precondition geeignet verstärken und die Postcondition abschwächen (beachte $\neg b \equiv (x = 0)$):

$$\frac{\models \overbrace{A \Rightarrow C}^{P_1} \qquad \{C\}\ p\ \{C \wedge \neg b\} \qquad \models \overbrace{(C \wedge \neg b) \Rightarrow B}^{P_2}}{\{A\}\ p\ \{B\}}$$

Nun können wir auf $\{C\}\ p\ \{C \wedge \neg b\}$ mit Schleifeninvariante C die While-Regel anwenden und die Schleife auflösen:

$$\frac{\{C \wedge b\}\ y := y * x;\ x := x - 1;\ \{C\}}{\{C\}\ p\ \{C \wedge \neg b\}}$$

Die Conc-Regel zerlegt den Schleifenrumpf und führt eine neue Aussage D ein:

$$\frac{\overbrace{\{C \wedge b\}\ y := y * x;\ \{D\}}^{A_1} \qquad \{D\}\ x := x - 1;\ \{C\}}{\{C \wedge b\}\ y := y * x;\ x := x - 1;\ \{C\}}$$

Wie wir unten sehen werden, müssen wir noch einmal mit der Weak-Regel abschwächen:

$$\frac{\overbrace{\{D\}\ x := x - 1;\ \{E\}}^{A_2} \qquad \models \overbrace{E \Rightarrow C}^{P_3}}{\{D\}\ x := x - 1;\ \{C\}}$$

Nachdem wir die Struktur des Beweisbaums bestimmt haben, müssen wir noch die Aussagen C, D und E angeben.
Sie sind $C \equiv (y = n * \cdots * (x + 1))$ (d. h. $y = n!/x!$) .
$D \equiv (y = n * \cdots * (x + 1) * x) \wedge (x \neq 0)$
$E \equiv (y = n * \cdots * (x + 1)) \wedge (x \neq -1)$

Um uns von der Richtigkeit dieser Aussagen zu überzeugen müssen wir die Aussagen P_1 – P_3 und die Axiome A_1 und A_2 überprüfen:

- P_1 – P_3 gelten trivialerweise.
- A_1: $D[y * x/y] \equiv (y * x = n * \cdots * (x + 1) * x) \wedge (x \neq 0)$.
  $\Leftrightarrow (y = n * \cdots * (x + 1)) \wedge (x \neq 0)$
  $\equiv C \wedge b$

  A_2: $E[x - 1/x] \equiv (y = n * \cdots * (x - 1 + 1)) \wedge ((x - 1) \neq -1)$
  $\Leftrightarrow (y = n * \cdots * x) \wedge (x \neq 0)$
  $\equiv D$

Achtung: Für $n < 0$ terminiert p nicht. Dies berücksichtigen wir hier einfach nicht!

Problem: Unsere Formeln müssen "ausdrucksstark" genug sein, um Aussagen wie etwa $y = n * \cdots * (x + 1)$ treffen zu können!

Ausweg: Wir führen Prädikate ein, wie etwa das dreistellige Prädikat q:

$$(Q)\quad \forall n,x,y : q(n,x,y) \iff [\ (n=x)\wedge(y=1)\ \vee\ (n>x)\wedge(\exists z : q(n,x+1,z)\wedge(y=z*(x+1)))\ ]$$

(ähnlich den Prädikatsdefinitionen in Prolog).

Wir quantifizieren nun Variablen und benötigen zusätzlich eine Menge von Prädiaktssymbolen $p(\ldots)$.

Eine Variablenbelegung σ erfüllt eine Aussage A jetzt nur noch relativ zu einer geeigneten *Interpretation* I der Prädikatsymbole.
Für ein n-stelliges Prädikatsymbol p gilt dabei für eine Interpretation von p: $I\,p \subseteq \mathbb{Z}^n$.

In unserem Beispiel von oben soll I etwa die Definition (Q) erfüllen.

Wir schreiben dann genauer: $\sigma \models_I A$ bzw. $\models_I C$.

Beispiel: ggT-Berechnung

$p \equiv$ `while` $(x \neq y)$ `do`
  `if` $(x > y)$ `then`
    $x := x - y;$
  `else`
    $y := y - x;$
  `end;`
`end;`

$A \equiv (x>0)\wedge(y>0)\wedge(ggT(x,y)=d)$ und $B \equiv (d>0)\wedge(x=d)\wedge(x=y)$

$$\frac{\{A\}\ p\ \{A\wedge\neg b\} \qquad \models (A\wedge\neg b)\Rightarrow B}{\{A\}\ p\ \{B\}} \qquad (\neg b \equiv (x=y))$$

While-Regel $\rightsquigarrow$

$$\frac{\{A\wedge b\}\ \texttt{if}\ (x>y)\ \texttt{then}\ x := x-y;\ \texttt{else}\ y := y-x;\ \texttt{end};\ \{A\}}{\{A\}\ p\ \{A\wedge\neg b\}}$$

If-Regel $\rightsquigarrow$

$$\frac{\overbrace{\{A\wedge b\wedge(x>y)\}\ x := x-y;\ \{A\}}^{A_1} \quad \overbrace{\{A\wedge b\wedge(x\leq y)\}\ y := y-x;\ \{A\}}^{A_2}}{\{A\wedge b\}\ \texttt{if}\ (x>y)\ \texttt{then}\ x := x-y;\ \texttt{else}\ y := y-x;\ \texttt{end};\ \{A\}}$$

A_1 und A_2 lassen sich leicht zeigen, indem $ggT(x,y)=d$ allein durch BOOLEsche Ausdrücke und Quantoren darstellt wird.