Semantik von Programmiersprachen

Vorlesung SoSe 2022
Wolfgang Mulzer

Jim Neuendorf

27. Juli 2022

Zusammenfassung

Diese Vorlesung vermittelt Techniken zur Formalisierung der Semantik (Bedeutungsinhalte) von Programmiersprachen. Zunächst werden unterschiedliche Formalisierungsansätze (die operationelle, denotationelle und axiomatische Semantik) vorgestellt und diskutiert. Anschließend wird die mathematische Theorie der semantischen Bereiche behandelt, die bei der denotationellen Methode, Anwendung findet. Danach wird schrittweise eine umfassende, imperative Programmiersprache entwickelt und die Semantik der einzelnen Sprachelemente denotationell spezifiziert. Dabei wird die Fortsetzungstechnik (continuation sem) systematisch erklärt und verwendet. Schließlich wird auf die Anwendung dieser Techniken eingegangen, insbesondere im Rahmen des Compilerbaus und als Grundlage zur Entwicklung funktionaler Programmiersprachen.

Inhaltsverzeichnis

22.04.

1 Einführung

Diese Vorlesung orientiert sich an folgender Literatur:

- Semantics with applications, Hanne Riis Nielson and Flemming Nielson, 1999
- Semantik von Programmiersprachen, Elfriede Fehr, 1989

1.1 Programmiersprachen

Eine Programmiersprache ist eine künstliche, entworfene Sprache zur Kommunikation zwischen Mensch und Rechner. Davon gibt es heutzutage sehr viele.

Keine Programmiersprache ist perfekt: Es gibt verschiedene Ziele und Aspekte beim Entwurf von Programmiersprachen und somit verschiedene Vor- und Nachteile:

- Komfort
  - Ausdrucksfähigkeit vs. Nützlichkeit
  - Lesbarkeit (z. B. COBOL) vs. Prägnanz (z. B. FORTRAN)
- Vermeidung von Fehlern im Programm
- leichte algorithmische Verarbeitung ($\rightsquigarrow$ parsing)
- effizienter erzeugter Code
  - Dies ist eigentlich Aufgabe des Übersetzers ($\rightsquigarrow$ Vorlesung Übersetzer-Bau)
  - abstrakt vs. hardware-nah

In dieser Vorlesung befassen wir uns mit der theoretische Analyse von Programmiersprachen. Konkret beinhaltet dies:

(a) einzelne Sprachkonstrukte mathematisch modellieren,
(b) Beziehung zwischen Kontrukten verstehen,
(c) Güte der Konstrukte beweisen,
(d) theoretische Garantien ableiten.

Jede Programmiersprache besitzt drei wesentliche Eigenschaften:

- Syntax
- Semantik
- Idiomatik

1.1.1 Syntax

Die Syntax ist eine endliche Folge von Zeichen über einem Alphabet Σ, z. B. `x = 5 + 2;` aus dem Alphabet ASCII.

Dabei sind nicht alle Zeichenfolgen korrekte Programme.

Die *konkrete Syntax* beschreibt Zeichenfolgen, die gültige Programme darstellen. Dafür werden eine kontextfreie Grammatik und zusätzliche Regeln (z. B. dass nur zuvor deklarierte Variablen verwendet werden dürfen) benutzt.

Die *abstrakte Syntax* ist die aus der Grammatik resultierende, hierarchische Struktur eines gültigen Programms. Diese wird durch einen Syntaxbaum dargestellt.

$$\text{Block} - \text{Anweisung} - \text{Zuweisung} - \begin{cases} \text{Variable} - x \\ \\ \text{Ausdruck} - \oplus - \begin{cases} 5 \\ \\ 2 \end{cases} \end{cases}$$

Die *Syntaxanalyse* erzeugt einen Syntaxbaum aus einem konkreten Programm. Dies ist Inhalt der Vorlesung Übersetzer-Bau. Hier, in dieser Vorlesung gehen wir davon aus, dass der abstrakte Syntaxbaum gegeben ist.

1.1.2 Semantik

Diese Eigenschaft beleuchten wir in dieser Vorlesung ab dem nächsten Abschnitt 2.

1.1.3 Idiomatik

Die *Idiomatik* umfasst Konventionen, Muster bzw. Faustregeln bei der Verwendung einer bestimmten Programmiersprache ($\rightsquigarrow$ pattern, anti-patterns, best practices). Somit macht sie in der Praxis den eigentlichen Gehalt / Kultur einer Programmiersprache aus.

29.04.

2 Semantik

Die *Semantik* eines Programmes ist die Bedeutung eines Programmes einer Programmiersprache.

Ziel: Finde eine klare, einfache mathematische Methode, um einem Programm eine *Bedeutung* zuzuordnen.

Motivation:

- Verifikation:
  - Erfüllt mein Programm die Spezifikation (tut es das, was es soll)?
  - Setzt der Übersetzer/Interpretierer die Spezifikation der Sprache korrekt um?
- Programmumformung
  - Haben zwei unterschiedliche Programme die gleiche Bedeutung?
  - Optimierung
- Programmanalyse
  - Ist das Programm "sicher" (secure vs. safe)?
  - Ist das Programm "effizient"?

Definition 2.1 (Programmierparadigma)**.** Programmierparadigma: z. B. deklarativ ("Was?") (funktional vs. logisch), imperativ ("Wie?"). In verschiedenen Paradigmen haben (potenziell) Programme verschiedene Bedeutungen.

Wir konzentrieren uns auf *imperative* Programmierung.

Frage. Was ist die "mathematische Bedeutung" eines imperativen Programms?

Frage (folgend)*.* Was ist ein imperatives Programm?

```
x = 1
y = x + 2
x = y + 5
for ...
```

Listing 1: Imperatives Programm

```
foo :: Int -> Int
foo 0 = 1
foo x = x + 1
foo 3
```

Listing 2: Funktionales Programm

Das zentrale Konzept der imperativen Programmierung ist der *Zustand* (state). Der Zustand ist der Inhalt aller Speicherzellen und Register, die Position des Programmzählers und der Zustand der Eingabe-/Ausgabe-Geräte.

Ein imperatives Programm ist eine Folge von *Anweisungen* (statement / instruction). Diese haben *Wirkungen* (effects), welche den Zustand verändern (selbst `nop` ändert den Programmzähler und somit den Zustand). Darüber hinaus gibt es Nebenwirkungen bzw. Seiteneffekte (side effects). Es gibt unterschiedliche Arten von Anweisungen:

- Zuweisungen (direkte Änderung des Zustandes)
- Kontrollfluss (Änderung des Programmzählers: Verzweigungen, Schleifen, Funktionsaufrufe bzw. Sprünge)
- Eingabe / Ausgabe

3 Mathematische Formalisierung

Definition 3.1 (Zustand)**.** Es gibt eine abzählbar unendliche Menge von Variablen $V = \{x_1, x_2, \ldots, y, z, \ldots\}$ (Speicher ist begrenzt aber beliebig groß). Der Zustand ist eine (partielle) Funktion

$$\sigma : V \to \mathbb{Z} \cup \{\perp, true, false\}$$

($\perp$ bedeutet undefiniert, d. h. eine Speicherzelle hat noch keinen Wert und die Funktion gibt nichts aus).

Die Teile des Zustandes “Eingabe / Ausgabe” ignorieren wir erst einmal, d. h. die initiale Eingabe ist implizit durch den Wert der Variablen am Anfang. Der Programmzähler wird an anderer Stelle thematisiert.

Bemerkung. Diese Definition dient als Beispiel, d. h. in anderen Szenarien mit anderen Variablen außer Ganzzahlen und Boolesche Wert kann eine andere Definition sinnvoller sein.

Definition 3.2 (Imperatives Programm)**.** Ein imperatives Programm ist eine Funktion auf der Menge alles Zustände. Jedem Startzustand wird ein Endzustand zugeordnet (wir ignorieren E/A).

Notation. Sei $\Pi \in \Sigma^*$ ein gültiges Programm (eine Zeichenkette). Wir bezeichnen mit

$$S[\![\Pi]\!] \in [State \to State]$$

(S ist die semantische Funktion) die Funktion, welche durch Π definiert wird.

3.1 `while`-Sprache

Definition 3.3. Wir verwenden in dieser Vorlesung eine einfache, turing-vollständige, imperative Programmiersprache als durchgängiges Beispiel namens `while`-*Sprache*, die durch folgende kontextfreie Grammatik gegeben ist:

$$\begin{aligned}
&A \to \texttt{Zahl} \mid \texttt{Var} \mid A + A \mid A * A \mid A - A\\
&B \to \texttt{true} \mid \texttt{false} \mid A = A \mid A \leqslant A \mid \neg B \mid B \wedge B\\
&S \to \texttt{Var := A | skip | S; S | if B then S else S | while B do S}
\end{aligned}$$

Bemerkung. Es gibt die syntaktischen Kategorien “arithmetischer Ausdruck” (A), “Boolescher Ausdruck” (B) und “Statement” (S, Anweisung).

Beispiel.

$$\Pi = \texttt{x = z + 1}$$

$$\mathcal{S}[\![\texttt{x = z + 1}]\!](\underbrace{[x \mapsto 5, z \mapsto -4, a \mapsto 2]}_{\text{Startzustand}}) = \underbrace{[x \mapsto -3, z \mapsto 4, a \mapsto 2]}_{\text{Endzustand}}$$

$$\mathcal{S}[\![\texttt{x = z + 3}]\!]([x \mapsto 10, z \mapsto 12]) = [x \mapsto 10, z \mapsto 12]$$

Für diese Veranstaltung stellen wir uns die Frage: Wie komme ich von Π zu $S[\![\Pi]\!]$?

Dafür gibt es drei Ansätze:

(a) axiomatische Semantik
(b) operationelle Semantik
(c) denotationelle Semantik

3.2 Axiomatische Semantik

Wir verzichten auf die vollständige Spezifikation von $S[\![\cdot]\!]$. Stattdessen arbeiten wir mit *Zusicherungen* (Assertions), welche wesentliche Aspekte des Zustands zu einem gegebenen Zeitpunkt widerspiegeln.

Wir definieren ein logisches System, das Beziehungen zwischen Zuständen aufstellt (Vorbedingungen, Nachbedingungen). Das System muss $S[\![\cdot]\!]$ verträglich sein.

Die Details sind Thema einer anderen Vorlesungen, z. B. Hoare-Kalkül.

Beispiel.

$$\underbrace{\{x = n \wedge y = m\}}_{\text{Vorbedingung}} \quad \texttt{z := x; x := y; y := z} \quad \underbrace{\{x = m \wedge y = n\}}_{\text{Nachbedingung}}$$

3.3 Operationelle Semantik

Definiere $S[\![\Pi]\!]$ durch schrittweise Simulation der Ausführung von Π (ein Interpretierer in mathematischer Form / Abstraktion).

Genauer gesagt bedeutet das: Wir definieren ein *Transitionssystem*

$$\langle \Pi, s\rangle \Rightarrow \langle \Pi', s'\rangle$$
$$\langle \Pi, s\rangle \Rightarrow s'$$

das die Ausführung von Π auf Zustand s darstellt.

Beispiel.

$$\begin{aligned}
&\langle \texttt{'z = x; x = y; y = z;'}, [x \mapsto 2, y \mapsto 3, z \mapsto 6]\rangle \\
\rightsquigarrow\ &\langle \texttt{'x = y; y = z;'}, [x \mapsto 2, y \mapsto 3, z \mapsto 2]\rangle \qquad \text{(1. Befehl ausgeführt)} \\
\rightsquigarrow\ &\langle \texttt{'y = z;'}, [x \mapsto 3, y \mapsto 3, z \mapsto 2]\rangle \\
\rightsquigarrow\ &[x \mapsto 3, y \mapsto 2, z \mapsto 2] \qquad \text{(Endzustand)}
\end{aligned}$$

06.05.

3.4 Denotationelle Semantik

Definiere $S[\![\Pi]\!]$ direkt als mathematische Funktion anhand der Syntax von Π, z. B.

$$\mathcal{S}[\![\texttt{z := x; x := y; y := z}]\!] = \mathcal{S}[\![\texttt{y := z}]\!] \circ \mathcal{S}[\![\texttt{x := y}]\!] \circ \mathcal{S}[\![\texttt{z := x}]\!]$$

Es wird also z. B. die sequenzielle Ausführung von Anweisungen als Funktionskomposition übersetzt.

Bemerkung (Problem). Wie kann man beispielsweise Schleifen darstellen (insbesondere `while`)? Ein möglicher Ansatz sind Grenzwerte, aber das geht tiefer in die Analysis.

Bei der operationellen Semantik wird die Schleife durch das Transitionssystem realisiert.

4 Operationelle Semantik

Das Folgende bezieht sich auf die `while`-Sprache (siehe Abschnitt 3.1).

Definition 4.1. $\mathrm{AExp}, \mathrm{BExp}, \mathrm{Stm}$: Mengen aller gültigen Ableitungen aus A, B bzw. S als Syntaxbaum. Der Ausdruck `5+7-2*8` lässt sich aus A ableiten. Der entsprechende Syntaxbaum (siehe Abbildung 1) ist dann Teil von AExp.

$$a \in \mathrm{AExp}$$

Mengen wie AExp bezeichnen wir als **syntaktische Kategorien**.

Zahl ist eine ganze Zahl aus $\mathbb{Z}$. Die zugehörige syntaktische Kategorie Num. Num ist die Menge aller Zeichenketten, die ganze Zahlen darstellen.

$$\texttt{1234} \in \mathrm{Num}$$
$$1234 \in \mathbb{Z}$$

Beachte, dass eigentlich der Syntaxbaum von `1234` gemeint ist.

Var sind Variablen, die nach Belieben vorhanden sind. Es sind abzählbar unendlich viele.

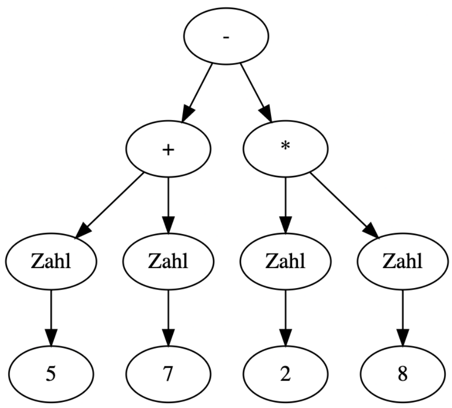


Abbildung 1: `5+7-2*8` $\in$ AExp als verkürzt dargestellter Syntaxbaum

Bemerkung. Unterbäume können auch Elemente einer anderen Kategorie sein.

Beispiel. Division mit Rest

- Eingabe: $a, b > 0$
- Ausgabe: $m, r \geqslant 0$, $r < b$, $a = m \cdot b + r$

```
m := 0;
while b <= a do (
    m := m + 1;
    a := a - b
)
r := a
```

Listing 3: Division mit Rest

Die *Semantik* in `while` wird gegeben durch eine *semantische Funktion*, eine für jede *syntaktische Kategorie*.

Sei $State = \{\sigma \mid \sigma : \mathrm{Var} \to \mathbb{Z}\}$, z. B.

$$\mathcal{N} : \mathrm{Num} \to \mathbb{Z}$$
$$\mathcal{N}[\![\texttt{-123}]\!] = -123$$

$$\mathcal{A} : \underbrace{\mathrm{AExp}}_{\text{“Compiler”}} \to \underbrace{(\mathrm{State} \to \mathbb{Z})}_{\text{“Interpreter”}}$$
$$\mathcal{A}\,[\![\texttt{x+x*5}]\!] = (\sigma \mapsto 6 \cdot \sigma(x)) \quad (\text{da } \texttt{x+x*5} = \texttt{6*x})$$

$$\mathcal{B} : \mathrm{BExp} \to (\mathrm{State} \to \{w, f\})$$
$$\mathcal{B}\,[\![\texttt{x<=10}]\!] = \left(\sigma \mapsto \begin{cases} w & \sigma(x) \leqslant 10 \\ f & \sigma(x) > 10 \end{cases}\right)$$

$$S : \mathrm{Stm} \to (\mathrm{State} \to \mathrm{State})$$

Jetzt: Definition von $\mathcal{A}$ durch Induktion über die Struktur des Syntaxbaums. Die Definition von $\mathcal{N}$ und $\mathcal{B}$ ist eine Übung.

Später: Definition von S.

4.1 Semantik arithmetischer Ausdrücke

Definition 4.2 (Induktive Definition von $\mathcal{A}$)**.** Sei $n \in \mathrm{Num}, x \in \mathrm{Var}, \sigma \in \mathrm{State}$ und $a_1, a_2 \in \mathrm{AExp}$. Wir definieren:

(i) $\mathcal{A}\,[\![n]\!](\sigma) = \mathcal{N}\,[\![n]\!]$ (konstante Funktion)

(ii) $\mathcal{A}\,[\![x]\!](\sigma) = \sigma(x)$ alternativ: $\mathcal{A}\,[\![n]\!] = (\sigma \mapsto \sigma(n))$

(iii) $\mathcal{A}\,[\![a_1 + a_2]\!](\sigma) = \mathcal{A}\,[\![a_1]\!](\sigma) + \mathcal{A}\,[\![a_2]\!](\sigma)$

Bemerkung. In anderen Sprachen könnte der Aufruf des ersten Summanden Seiteneffekte haben, d. h. ggf. muss man an dieser Stelle aufpassen. In diesem Fall würde der potenziell veränderte Zustand mit zurückgegeben werden.

(iv) Analog für Subtraktion

(v) Analog für Multiplikation

Bemerkung. Für die Definition von semantischen Funktionen fordern wir **Zusammengesetztheit**, d. h. in der induktiven Definition darf nur auf Bestandteile des Ausdrucks/Syntaxbaums zugegriffen werden.

Beispiel. Führe Negation ein: $\mathcal{A} \to \ldots \mid -A$.

Erlaubt ist $\mathcal{A}\llbracket -a_1 \rrbracket(\sigma) = 0 - \mathcal{A}\llbracket a_1 \rrbracket(\sigma)$ aber nicht $\mathcal{A}\llbracket -a_1 \rrbracket(\sigma) = \mathcal{A}\llbracket 0 - a_1 \rrbracket(\sigma)$, da der Ausdruck hier um eine Null erweitert wurde.

13.05.

Satz 4.1. *$\mathcal{A}$ besitzt die folgenden Eigenschaften:*

(a) für alle $a \in \text{AExp}$, für alle $\sigma \in \text{State}$ existiert genau eine Zahl $n \in \mathbb{Z}$, sodass $\mathcal{A}[\![a]\!](\sigma) = n$. (Voraussetzung: σ ist eine totale Funktion.)

(b) $\mathcal{A}$ ist eine totale Funktion.

Beweis. Skizze:

(b) folgt aus (a)

(a) wird bewiesen durch strukturelle Induktion nach a.

□

Nächste Schritte:

(i) Der Wert eines Ausdrucks hängt *nur* von den Variablen ab, die in ihm vorkommen. (Abschnitt 4.2)

(ii) Was passiert in einem arithmetischen Ausdruck, wenn wir eine Variable durch einen Ausdruck ersetzen ($\rightsquigarrow$ *Substitution*)? (Abschnitt 4.3)

4.2 Freie Variablen

Definition 4.3 (Freie Variablen)**.** Sei $a \in \text{AExp}$. $\text{FV}(a)$ ("freie Variablen") ist die Menge aller Variablen, die in a vorkommen. Formal ist das induktiv definiert:

(a) $\text{FV}(n) = \varnothing$ (falls $a = n$ (Zahl))

(b) $\text{FV}(x) = \{x\}$ (falls $a = x$ (Variable))

(c) $\text{FV}(a_1 \ \Box \ a_2) = \text{FV}(a_1) \cup \text{FV}(a_2)$ für $\Box \in \{+, -, *\}$

Gebundene Variablen betrachten wir im Kontext der `while`-Sprache nicht. In anderen Sprachen können aber lokale Variablen z. B. als solche betrachtet werden.

Lemma 4.4. *Sei $a \in \text{AExp}$, sei $\sigma, \sigma' \in \text{State}$, sodass $\sigma(x) = \sigma'(x)$ für alle x in $\text{FV}(a)$ gilt.*

Dann gilt:

$$\mathcal{A}[\![a]\!](\sigma) = \mathcal{A}[\![a]\!](\sigma')$$

Beweis. Durch strukturelle Induktion nach a.

Induktionsanfang:

(a) $a = n$, n Zahl

$$\begin{aligned}\mathcal{A}\,[\![a]\!](\sigma) &= \mathcal{A}\,[\![n]\!](\sigma)\\ &\overset{\text{D. }4.2(i)}{=} \mathcal{N}\,[\![n]\!]\end{aligned}$$

$$\begin{aligned}\mathcal{A}\,[\![a]\!](\sigma') &= \mathcal{A}\,[\![n]\!](\sigma')\\ &= \mathcal{N}\,[\![n]\!]\end{aligned}$$

(b) $a = x$, x Variable

$$\begin{aligned}\mathcal{A}\,[\![a]\!](\sigma) &= \mathcal{A}\,[\![x]\!](\sigma)\\ &\overset{\text{D. }4.2(ii)}{=} \sigma(x)\end{aligned}$$

$$\begin{aligned}\mathcal{A}\,[\![a]\!](\sigma') &= \mathcal{A}\,[\![x]\!](\sigma')\\ &= \sigma'(x)\end{aligned}$$

$$\mathrm{FV}(a) = \mathrm{FV}(x) = \{x\}$$

Nach Annahme ist $\sigma(x) = \sigma'(x)$, da $x \in \mathrm{FV}(a)$. Daher folgt

$$\mathcal{A}\,[\![a]\!](\sigma) = \mathcal{A}\,[\![a]\!](\sigma')$$

Induktionsschritt:

$a = a_1 \;\square\; a_2$ mit $\square \in \{+, -, *\}$

$$\begin{aligned}\mathcal{A}\,[\![a]\!](\sigma) &= \mathcal{A}\,[\![a_1 \;\square\; a_2]\!](\sigma)\\ &= \mathcal{A}\,[\![a_1]\!](\sigma) \;\square\; \mathcal{A}\,[\![a_2]\!](\sigma)\end{aligned}$$

$$\begin{aligned}\mathcal{A}\,[\![a]\!](\sigma') &= \mathcal{A}\,[\![a_1 \;\square\; a_2]\!](\sigma')\\ &= \mathcal{A}\,[\![a_1]\!](\sigma') \;\square\; \mathcal{A}\,[\![a_2]\!](\sigma')\end{aligned}$$

$$\begin{aligned}\mathrm{FV}(a) &= \mathrm{FV}(a_1 \;\square\; a_2)\\ &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)\end{aligned}$$

Insbesondere gilt, dass $\mathrm{FV}(a_1) \subseteq \mathrm{FV}(a)$ und $\mathrm{FV}(a_2) \subseteq \mathrm{FV}(a)$. Daher folgt:

Da nach Annahme $\sigma(x) = \sigma(x')$ für alle $x \in \mathrm{FV}(a)$, gilt

$$\sigma(x) = \sigma'(x) \text{ für alle } x \in \mathrm{FV}(a_1)$$

und

$$\sigma(x) = \sigma'(x) \text{ für alle } x \in \mathrm{FV}(a_2)$$

Also gelten nach Induktionsvoraussetzung

$$\mathcal{A}[\![a_1]\!](\sigma) = \mathcal{A}[\![a_1]\!](\sigma') \wedge \mathcal{A}[\![a_2]\!](\sigma) = \mathcal{A}[\![a_2]\!](\sigma')$$

Daher folgt:

$$\mathcal{A}[\![a_1]\!](\sigma) \;\square\; \mathcal{A}[\![a_2]\!](\sigma) = \mathcal{A}[\![a_1]\!](\sigma') \;\square\; \mathcal{A}[\![a_2]\!](\sigma')$$

□

4.3 Substitution

Definition 4.5 (Substitution für Ausdrücke)**.** Seien $a, a_0 \in \text{AExp}$, $y \in \text{Var}$. Wir definieren $a[y \mapsto a_0]$ als den arithmetischen Ausdruck, in dem jedes Vorkommen von y in a durch a_0 ersetzt wird.

Formal:

(i) $n[y \mapsto a_0] = n$

(ii) $x[y \mapsto a_0] = \begin{cases} a_0 & \text{falls } x = y \\ x & \text{sonst} \end{cases}$

(iii) $(a_1 \;\square\; a_2)[y \mapsto a_0] = a_1[y \mapsto a_0] \;\square\; a_2[y \mapsto a_0] \qquad \text{für } \square \in \{+, -, *\}$

Beispiel.

$$a = \texttt{x + y*2 - z(x + y)}$$
$$\texttt{y} \mapsto \texttt{4*t + 5}$$

$$a[\texttt{y} \mapsto \texttt{4*t + 5}] = \texttt{x + (4*t + 5)*2 - z(x + (4*t + 5))}$$

Es werden die Blätter am Syntaxbaum ersetzt, woraus implizit die Präzedenz klar ist (wodurch die obigen Klammern entstehen).

Ersetzungen dürfen die zu ersetzende Variable enthalten. Da nur einmal ersetzt wird, ist das in Ordnung.

Definition 4.6 (Substitution für Zustände)**.** Sei $\sigma \in \text{State}$, $x \in \text{Var}$, $n \in \mathbb{Z}$. Dann ist $\sigma[x \mapsto n] \in \text{State}$ der Zustand, der wie folgt definiert ist:

$$\sigma[x \mapsto n](z) = \begin{cases} n & \text{falls } z = x \\ \sigma(z) & \text{sonst} \end{cases}$$

Lemma 4.7. *Sei* $a, a_0 \in \text{AExp}$, $y \in \text{Var}$, $\sigma \in \text{State}$*. Dann gilt:*

$$\mathcal{A}[\![a[y \mapsto a_0]]\!](\sigma) = \mathcal{A}[\![a]\!]\Big(\sigma\big[y \mapsto \mathcal{A}[\![a_0]\!](\sigma)\big]\Big)$$

Das Lemma setzt Semantik und Syntax in Verbindung: Wir können syntaktisch Variablen durch Teilausdrücke ersetzen und das ist äquivalent dazu, erst den Teilausdruck auszuwerten und mit diesen Wert im Zustand den ursprünglichen Ausdruck auszuwerten.

Beispiel.

$$\begin{aligned} a &= x + y \\ a_0 &= x * y \\ \sigma &: [x \mapsto 2; y \mapsto 10] \end{aligned}$$

$$\begin{aligned} (x + (x * y))[x \mapsto 2; y \mapsto 10] \quad &\square\ (x + y)[x \mapsto 2; y \mapsto 20] \\ 2 + (2 * 10) \quad &\square\ 2 + 20 \\ 22 \quad &= 22 \end{aligned}$$

Beweis. Durch strukturelle Induktion nach a (nicht a_0!).

Induktionsanfang:

(a) $a = n$, n Zahl

$$\begin{aligned} \mathcal{A}[\![a[y \mapsto a_0]]\!](\sigma) &= \mathcal{A}[\![n[y \mapsto a_0]]\!](\sigma) \\ &\overset{\text{D. } 4.5(i)}{=} \mathcal{A}[\![n]\!](\sigma) \\ &= n \end{aligned}$$

$$\mathcal{A}[\![n]\!](\sigma[y \mapsto \mathcal{A}[\![a_0]\!](\sigma)]) = n \qquad \text{(rechte Seite)}$$

(b) $a = x$, x Variable

$$\mathcal{A}[\![x[y \mapsto a_0]]\!](\sigma) \overset{\text{D. } 4.5(iii)}{=} \begin{cases} \mathcal{A}[\![a_0]\!](\sigma) & \text{falls } x = y \\ \mathcal{A}[\![x]\!](\sigma) = \sigma(x) & \text{sonst} \end{cases}$$

$$\begin{aligned} \mathcal{A}[\![x]\!](\sigma[y \mapsto \mathcal{A}[\![a_0]\!](\sigma)]) &\overset{\text{D. } 4.6}{=} (\sigma[y \mapsto \mathcal{A}[\![a_0]\!](\sigma)])(x) \qquad \text{(rechte Seite)} \\ &= \begin{cases} \mathcal{A}[\![a_0]\!](\sigma) & \text{falls } x = y \\ \sigma(x) & \text{sonst} \end{cases} \end{aligned}$$

Induktionsschritt: Straight forward. $\square$

20.05.

4.4 Natürliche operationelle Semantik ("big step")

$\mathcal{A}[\![\cdot]\!]$ und $\mathcal{B}[\![\cdot]\!]$ *benutzen* den Zustand σ während die Semantik von Anweisungen den Zustand *verändern* soll.

Idee: Definiere $\mathcal{S}[\![\cdot]\!]$ mithilfe einer Zustandsüberführungerelation.

Definition 4.8. Sei $s \in \mathrm{Stm}$ eine Anweisungsfolge und seien $\sigma, \sigma' \in \mathrm{State}$ Zustände. Die Zustandsüberführungsrelation

$$\langle S, \sigma\rangle \rightarrow \sigma'$$

spezifiziert die Beziehung zwischen Startzustand σ und dem Endzustand σ' gemäß der Anweisungsfolge S.

Bemerkung (Bedeutung). Die Ausführung von S auf Startzustand σ terminiert mit Endzustand σ'.

Notation. Wir definieren die Zustandsübergangsrelation mithilfe von Schlussregeln. Eine solche Schlussregel besitzt die folgende Form:

$$\frac{\text{Voraussetzung}}{\text{Folgerung}} \rightsquigarrow \frac{\langle S_1, \sigma_1\rangle \rightarrow \sigma_1', \langle S_2, \sigma_2\rangle \rightarrow \sigma_2',\ \dots\ , \langle S_n, \sigma_n\rangle \rightarrow \sigma_n'}{\langle S, \sigma\rangle \rightarrow \sigma'}$$

(ggf. mit zusätzlichen Bedingungen) wobei $S_1, \dots, S_n$ Bestandteile von S sind.

Gibt es keine Voraussetzung, nennen wir die Schlussregel ein *Axiom*.

4.4.1 Schlussregeln für die natürliche Semantik von `while`

Im Folgenden schreiben wir $[\cdot]_{\mathrm{ns}}$ um anzuzeigen, dass es sich um die natürliche Semantik handelt.

(a) Zuweisung $[\mathrm{zuw}_{\mathrm{ns}}]$ (Axiom)

$$\overline{\langle x\ \texttt{:=}\ a, \sigma\rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma)]}$$

(b) Skip $[\mathrm{skip}_{\mathrm{ns}}]$ (Axiom)

$$\overline{\langle \texttt{skip}, \sigma\rangle \rightarrow \sigma}$$

(c) Hintereinanderausführung $[\mathrm{seq}_{\mathrm{ns}}]$

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

Dabei können S_1 und S_2 zusammengesetzte Anweisungen sein.

(d) Verzweigung [$\text{if}_{\text{ns}}^{\mathbf{w}}$]

$$\frac{\langle S_1, \sigma \rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma \rangle \to \sigma'}$$

falls $\mathcal{B}[\![b]\!](\sigma) = \mathbf{w}$. Dabei muss S_2 nicht terminieren.

(e) Verzweigung [$\text{if}_{\text{ns}}^{\mathbf{f}}$]

$$\frac{\langle S_2, \sigma \rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma \rangle \to \sigma'}$$

falls $\mathcal{B}[\![b]\!](\sigma) = \mathbf{f}$. Dabei muss S_1 nicht terminieren.

(f) Schleife [$\text{while}_{\text{ns}}^{\mathbf{w}}$]

$$\frac{\langle S, \sigma \rangle \to \sigma', \langle \texttt{while } b \texttt{ do } S, \sigma' \rangle \to \sigma''}{\langle \texttt{while } b \texttt{ do } S, \sigma \rangle \to \sigma''}$$

falls $\mathcal{B}[\![b]\!](\sigma) = \mathbf{w}$. Im Allgemeinen kann es sein, dass die Schleife nicht terminiert. Deshalb müssen wir diesen Fall in der Definition der semantischen Funktion beachten. Das bedeutet, diese Relation ist nicht total.

(g) Schleife [$\text{while}_{\text{ns}}^{\mathbf{f}}$]

$$\frac{}{\langle \texttt{while } b \texttt{ do } S, \sigma \rangle \to \sigma}$$

falls $\mathcal{B}[\![b]\!](\sigma) = \mathbf{f}$.

Wie ist die Zustandsüberführungsfunktion überhaupt definiert?

Definition 4.9. Sei $S \in \text{Stm}$ ein Programm und seien $\sigma, \sigma' \in \text{State}$. Dann gilt

$$\langle S, \sigma \rangle \to \sigma'$$

gdw. ein *endlicher Ableitungsbaum* dafür existiert.

Der Ableitungsbaum entsteht durch wiederholte Anwendung der Schlussregeln. Die Wurzel ist $\langle S, \sigma \rangle \to \sigma'$, die Blätter sind Axiome, die Knoten entsprechen der korrekten Anwendung der Schlussregeln.

Beispiel. Sei $\sigma \in \text{State}$ mit $\sigma(x) = 1$ und $\sigma(y) = 5$.

Behauptung: $\langle (z := x; x := y); y := z, \sigma \rangle \to \sigma[z \mapsto 1][x \mapsto 5][y \mapsto 1]$

Nun müssen wir den endlichen Ableitungsbaum erzeugen.

(a) [seq_{ns}]

$$\frac{\langle z := x; x := y, \sigma \rangle \to \sigma', \langle y := z, \sigma' \rangle \to \sigma[z \mapsto 1][x \mapsto 5][y \mapsto 1]}{\langle \underbrace{(z := x; x := y)}_{s_1}; \underbrace{y := z}_{S_2}, \sigma \rangle \to \sigma[z \mapsto 1][x \mapsto 5][y \mapsto 1]}$$

Welches σ' brauchen wir? $\rightsquigarrow \sigma[z \mapsto 1][x \mapsto 5]$. Somit erhalten wir

$$\frac{\langle z := x; x := y, \sigma\rangle \to \sigma[z \mapsto 1][x \mapsto 5], \langle y := z, \sigma[z \mapsto 1][x \mapsto 5]\rangle \to \sigma[z \mapsto 1][x \mapsto 5][y \mapsto 1]}{\langle \underbrace{(z := x; x := y)}_{S_1}; \underbrace{y := z}_{S_2}, \sigma\rangle \to \sigma[z \mapsto 1][x \mapsto 5][y \mapsto 1]}$$

(b) $[\text{zuw}_{\text{ns}}]$ für den rechten Teil

$$\frac{\mathcal{A}[\![z]\!](\sigma[z \mapsto 1][x \mapsto 5]) \stackrel{?}{=} 1}{\langle y := z, \sigma[z \mapsto 1][x \mapsto 5]\rangle \to \sigma[z \mapsto 1][x \mapsto 5][y \mapsto 1]}$$

$$\mathcal{A}[\![z]\!](\sigma[z \mapsto 1][x \mapsto 5]) \stackrel{Def}{=} \sigma[z \mapsto 1][x \mapsto 5](z) \stackrel{Def}{=} 1$$

(c) $[\text{seq}_{\text{ns}}]$

$$\frac{\langle z := x, \sigma\rangle \to \sigma[z \mapsto 1], \langle x := y, \sigma[z \mapsto 1]\rangle \to \sigma[z \mapsto 1][x \mapsto 5]}{\langle z := x; x := y, \sigma\rangle \to \sigma[z \mapsto 1][x \mapsto 5]}$$

Ab hier analog zum rechten Teil (mit zwei Mal $[\text{zuw}_{\text{ns}}]$).

Bemerkung (Rechtseindeutigkeit / Determiniertheit). Es ist noch nicht klar, dass "$\to$" *rechtseindeutig* ist. D. h. möglicherweise existiert ein Programm S, ein Startzustand σ und Zustände $\sigma_1 \neq \sigma_2 \in \text{State}$, sodass sowohl

$$\langle S, \sigma\rangle \to \sigma_1 \text{ als auch } \langle S, \sigma\rangle \to \sigma_2$$

gilt. Daher muss die Rechtseindeutigkeit bzw. *Determiniertheit* bewiesen werden!

Bemerkung (Ableitungsbaum). Der Ableitungsbaum ist statisch. D. h. man kann nicht erkennen, in welcher Reihenfolge die Schlussregeln angewendet werden.

27.05.

Definition 4.10. Sei S ein Programm und σ ein Startzustand. Das Programm S *terminiert* bei Startzustand σ falls ein σ' existiert, sodass $\langle S, \sigma \rangle \to \sigma'$ gilt.

? Analog *terminiert S nicht* bei Startzustand σ.

Das Programm S *terminiert immer*, falls S für jeden Startzustand σ terminiert. S *terminiert nie*, falls S für keinen Startzustand σ terminiert.

Definition 4.11. Seien S_1, S_2 zwei Programme. S_1 und S_2 heißen *semantisch äquivalent*, falls für alle Zustände σ, σ' gilt

$$\langle S_1, \sigma \rangle \to \sigma' \Leftrightarrow \langle S_2, \sigma \rangle \to \sigma'$$

Beispiel. Die Programme

$$S_1 = \texttt{while b do S}$$

und

$$S_2 = \texttt{if b then (S; while b do S) else skip}$$

sind semantisch äquivalent.

Beweis. Seien σ, σ' Zustände. Z. z.: $\langle S_1, \sigma \rangle \to \sigma' \Leftrightarrow \langle S_2, \sigma \rangle \to \sigma'$.

(a) "$\Rightarrow$"

Angenommen, es gilt $\langle S_1, \sigma \rangle \to \sigma'$. Also existiert nach Definition ein endlicher Ableitungsbaum für $\langle S_1, \sigma \rangle \to \sigma'$.

(a) $\mathcal{B}[\![b]\!](\sigma) = \mathbf{w}$

Dann hat der Ableitungsbaum T_1 die Form

$$[\text{while}^{\mathbf{w}}_{\text{ns}}] \ \frac{\dfrac{T_a}{\langle S, \sigma \rangle \to \sigma''}, \ \dfrac{T_b}{\langle \texttt{while b do S}, \sigma'' \rangle \to \sigma'}}{\langle \texttt{while b do S}, \sigma \rangle \to \sigma'}$$

Z. z.: es existiert ein endlicher Ableitungsbaum für $\langle S_2, \sigma \rangle \to \sigma'$

$$[\text{if}^{\mathbf{w}}_{\text{ns}}] \ \frac{[\text{seq}_{\text{ns}}] \ \dfrac{\dfrac{T_a}{\langle S, \sigma \rangle \to \sigma''}, \dfrac{T_b}{\langle \texttt{while b do S}, \sigma'' \rangle \to \sigma'}}{\langle \texttt{S; while b do S}, \sigma \rangle \to \sigma'}}{\langle \texttt{if b then (S; while b do S) else skip}, \sigma \rangle \to \sigma'} \quad \mathcal{B}[\![b]\!](\sigma) = \mathbf{w}$$

(b) $\mathcal{B}[\![b]\!](\sigma) = \mathbf{f}$

Dann hat T_1 die Form

$$[\text{while}^{\mathbf{f}}_{\text{ns}}] \ \frac{}{\langle \texttt{while b do S}, \sigma \rangle \to \sigma} \quad \mathcal{B}[\![b]\!](\sigma) = \mathbf{f}$$

Dieses Axiom ist wahr, gdw. $\sigma = \sigma'$.

Z. z.: Es existiert ein Ableitungsbaum für $\langle S_2, \sigma\rangle \to \sigma$.

$$[\mathrm{if}^{\mathbf{f}}_{\mathrm{ns}}]\ \frac{[\mathrm{skip}_{\mathrm{ns}}]\ \dfrac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma}}{\langle \texttt{if b then (S; while b do S) else skip}, \sigma\rangle \to \sigma}\quad \mathcal{B}[\![b]\!](\sigma) = \mathbf{f}$$

(b) "$\Leftarrow$"

Analog.

□

Lemma 4.12. *Die natürliche Semantik der* `while`*-Sprache ist* determiniert*, d. h. für alle Anweisungen S und Zustände $\sigma, \sigma_1, \sigma_2$ gilt:*

Wenn $\langle S, \sigma\rangle \to \sigma_1$ und $\langle S, \sigma\rangle \to \sigma_2$, dann $\sigma_1 = \sigma_2$.

Beweis. Durch strukturelle Induktion nach Tiefe des Ableitungsbaums. (Übung) □

Definition 4.13. Die semantische Funktion $\mathcal{S}_{\mathrm{ns}} : \mathrm{State} \to (\mathrm{State} \to \mathrm{State})$ ist definiert als

$$\mathcal{S}_{\mathrm{ns}}[\![S]\!](\sigma) = \begin{cases} \sigma' & \text{falls } \exists\, \sigma' : \langle S, \sigma\rangle \to \sigma' \\ \perp & \text{sonst} \end{cases}$$

4.5 Strukturelle operationelle Semantik ("small step")

Hier geht es um die genaue Reihenfolge der Schritte bei der Ausführung. Das ist beispielsweise nützlich bei der parallelen Ausführungen eines Programms.

Wir definieren wieder eine Zustandsüberführungsrelation "⇒". Sie hat die Form

$$\langle S, \sigma \rangle \Rightarrow \langle S', \sigma' \rangle \tag{*}$$

oder

$$\langle S, \sigma \rangle \Rightarrow \sigma' \tag{**}$$

Interpretation:

(*) Ausführung ist noch nicht vorbei, sondern erreicht in *einem Schritt* die *Zwischenkonfiguration* $\langle S', \sigma' \rangle$.

(**) Ausführung ist nach einem Schritt vorbei und erreicht den Endzustand σ'.

Wir definieren ⇒ durch folgende Schlussregeln. Im Folgenden schreiben wir $[\cdot]_{\text{sos}}$ um anzuzeigen, dass es sich um die strukturelle operationelle Semantik handelt.

(a) $[\text{zuw}_{\text{sos}}]$

$$\langle x \ \texttt{:=} \ a, \sigma \rangle \Rightarrow \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma)]$$

(b) $[\text{skip}_{\text{sos}}]$

$$\langle \texttt{skip}, \sigma \rangle \Rightarrow \sigma$$

(c) $[\text{seq}^1_{\text{sos}}]$

$$\frac{\langle S_1, \sigma \rangle \Rightarrow \langle S_1', \sigma' \rangle}{\langle S_1 ; S_2, \sigma \rangle \Rightarrow \langle S_1' ; S_2, \sigma' \rangle}$$

(d) $[\text{seq}^2_{\text{sos}}]$

$$\frac{\langle S_1, \sigma \rangle \Rightarrow \sigma'}{\langle S_1 ; S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma' \rangle}$$

(e) $[\text{if}^{\mathbf{w}}_{\text{sos}}]$

$$\langle \texttt{if} \ b \ \texttt{then} \ S_1 \ \texttt{else} \ S_2, \sigma \rangle \Rightarrow \langle S_1, \sigma \rangle$$

falls $\mathcal{B}[\![b]\!](\sigma) = \mathbf{w}$

(f) $[\text{if}^{\mathbf{f}}_{\text{sos}}]$

$$\langle \texttt{if} \ b \ \texttt{then} \ S_1 \ \texttt{else} \ S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma \rangle$$

falls $\mathcal{B}[\![b]\!](\sigma) = \mathbf{f}$

(g) $[\text{while}_{\text{sos}}]$

$$\langle \texttt{while} \ b \ \texttt{do} \ S, \sigma \rangle \Rightarrow \langle \texttt{if} \ b \ \texttt{then} \ \texttt{(S;} \ \texttt{while} \ b \ \texttt{do} \ S) \ \texttt{else} \ \texttt{skip}, \sigma \rangle$$

Definition 4.14. Sei S eine Anweisung und σ ein Zustand.

Eine Ableitungsfolge für $\langle S, \sigma \rangle$ ist entweder

(a) eine endliche Folge $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_k$ von Konfigurationen, sodass $\gamma_0 = \langle S, \sigma \rangle$ ist, $\gamma_i \Rightarrow \gamma_{i+1}$ für $0, \ldots, k-1$ gilt und γ_k entweder ein Zustand σ' oder eine Konfiguration $\langle S', \sigma' \rangle$ ist, für die es mit $\Rightarrow$ nicht weiter geht *(steckengebliebene Konfiguration)*.

(b) eine unendliche Folge $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots$ von Konfigurationen mit $\gamma_0 = \langle S, \sigma \rangle$ mit $\gamma_i \Rightarrow \gamma_{i+1}$ für $i \geqslant 0$.

Notation. Wir schreiben

$\gamma_0 \Rightarrow^i \gamma'$ für "γ' geht aus i Schritten hervor" und

$\gamma_0 \Rightarrow^* \gamma'$ für "γ' geht aus endlich vielen Schritten hervor" (auch null).

03.06.

Beispiel. Sei σ ein Zustand mit $\sigma(x) = 5, \sigma(y) = 7$. Betrachte die Auswertung von `(z := x; x := y); y = z;` in der SOS für Startzustand σ.

$$\begin{aligned}
& \langle \texttt{(z := x; x := y); y = z;}, \sigma \rangle \\
\overset{[\text{seq}^1_{\text{sos}}][\text{seq}^2_{\text{sos}}][\text{zuw}_{\text{sos}}]}{\Rightarrow} \; & \langle \texttt{x := y; y := z}, \sigma[z \mapsto 5] \rangle && \text{(i)} \\
\overset{[\text{seq}^2_{\text{sos}}][\text{zuw}_{\text{sos}}]}{\Rightarrow} \; & \langle \texttt{y := z}, \sigma[z \mapsto 5][y \mapsto 7] \rangle && \text{(ii)} \\
\overset{[\text{zuw}_{\text{sos}}]}{\Rightarrow} \; & \sigma[z \mapsto 5][x \mapsto 7][y \mapsto 5]
\end{aligned}$$

zu (i):

$$[\text{seq}^1_{\text{sos}}] \; \dfrac{[\text{seq}^2_{\text{sos}}] \; \dfrac{[\text{zuw}_{\text{sos}}] \; \dfrac{}{\langle \texttt{z := x}, \sigma \rangle \Rightarrow \sigma[z \mapsto 5]}}{\langle \texttt{z := x; x := y}, \sigma \rangle \Rightarrow \langle \texttt{x := y}, \sigma[z \mapsto 5] \rangle}}{\langle \texttt{(z := x; x := y); y := z}, \sigma \rangle \Rightarrow \langle \texttt{x := y; y := z}, \sigma[z \mapsto 5] \rangle}$$

zu (ii):

$$[\text{seq}^2_{\text{sos}}] \; \dfrac{[\text{zuw}_{\text{sos}}] \; \dfrac{}{\langle x := y, \sigma[z \mapsto 5] \rangle \Rightarrow \langle y := z, \sigma[z \mapsto 5][x \mapsto 7] \rangle}}{\langle x := y; y := z, \sigma[z \mapsto 5] \rangle \Rightarrow \langle y := z, \sigma[z \mapsto 5][x \mapsto 7] \rangle}$$

4.6 Eigenschaften der SOS

Lemma 4.15. *Seien S_1, S_2 Anweisungen, σ, σ'' Zustände und $k \in \mathbb{N}$. Dann gilt: Falls $\langle S_1; S_2, \sigma \rangle \Rightarrow^k \sigma''$ gilt, es existieren zwei Zahlen $k_1, k_2 \in \mathbb{N}$ mit*

$$\langle S_1, \sigma \rangle \Rightarrow^{k_1} \sigma' \,, \quad \langle S_2, \sigma' \rangle \Rightarrow^{k_2} \sigma''$$

und

$$k_1 + k_2 = k$$

Beweis. Induktion nach k.

Induktionsanfang:

(a) $k = 1$: Voraussetzung kann dafür nicht erfüllt sein, also stimmt die Aussage.

(b) $k = 2$: Wie kann $\langle S_1; S_2, \sigma \rangle \Rightarrow^2 \sigma''$ gelten? Das kann nur sein, wenn im ersten Schritt $[\text{seq}^2_{\text{sos}}]$ augewendet wird. D. h. die Schlussregel

$$\frac{\langle S_1, \sigma \rangle \Rightarrow \sigma'}{\langle S_1; S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma' \rangle}$$

wurde erfüllt für ein σ'.

Wir wissen also, es existiert ein Zwischenzustand σ' mit $\langle S_1, \sigma\rangle \Rightarrow \sigma'$ und erster Schritt von $\langle S_1; S_2, \sigma\rangle \Rightarrow^2 \sigma''$ ist $\langle S_1; S_2, \sigma\rangle \Rightarrow \langle S_2, \sigma'\rangle$. Der zweite Schritt $\langle S_1; S_2, \sigma\rangle \Rightarrow^2 \sigma''$ muss jetzt aber der Form $\langle S_2, \sigma'\rangle \Rightarrow \sigma''$ sein.

D. h. die Aussage gilt mit $k_1 = 1, k_2 = 1$ und σ'.

Induktionsschritt: $k - 1 \mapsto k$ mit $k \geqslant 3$

Betrachten den ersten Schritt $\langle S_1; S_2, \sigma\rangle \Rightarrow^k \sigma''$. Zwei Fälle:

(a) $[\text{seq}^1_{\text{sos}}]$ Der erste Schritt hat die Form $\langle S_1; S_2, \sigma\rangle \Rightarrow^k \langle S_1'; S_2, \sigma'''\rangle$

Dann muss aber gelten $\langle S_1'; S_2, \sigma'''\rangle \Rightarrow^{k-1} \sigma''$. Nach IV existiert $k_1', k_2' \in \mathbb{N}, \sigma'$, sodass $\langle S_1', \sigma'''\rangle \Rightarrow^{k_1'} \sigma'$ und $\langle S_2', \sigma'\rangle \Rightarrow^{k_2'} \sigma''$ und $k_1' + k_2' = k - 1$.

Da wir im ersten Schritt $[\text{seq}^1_{\text{sos}}]$ angewandt haben, muss die Schlussregel dafür erfüllt gewesen sein, d. h. es gilt $\langle S_1, \sigma\rangle \Rightarrow \langle S_1', \sigma'''\rangle$. Also gilt auch $\langle S_1', \sigma\rangle \Rightarrow^{k_1'} \sigma'$ und somit $\langle S_1, \sigma\rangle \Rightarrow^{k_1'+1} \sigma'$.

Also gilt die Aussage für $k_1 = k_1' + 1, k_2 = k_2', \sigma'$.

(b) $[\text{seq}^2_{\text{sos}}]$ Der erste Schritt hat die Form $\langle S_1; S_2, \sigma\rangle \Rightarrow \langle S_2, \sigma'\rangle$ und es gilt $\langle S_1, \sigma\rangle \Rightarrow \sigma'$.

Also gilt die Aussage für $k_1 = 1, k_2 = k - 1, \sigma'$.

□

Lemma 4.16 (Determiniertheit)**.** *SOS ist* determiniert. *Anders als bei der natürlichen Semantik müssen auch alle Zwischenzustände gleich sein, d. h.:*

Für jedes S, σ existiert eine Ableitungsfolge, die mit $\langle S, \sigma\rangle$ beginnt.

Definition 4.17 (Semantische Äquivalenz)**.** Seien S_1, S_2 zwei Anweisungen. S_1, S_2 heißen *semantisch äquivalent* gdw. folgendes für alle Zustände σ gilt:

(a) Für alle steckengebliebenen Konfigurationen γ und alle Endzustände σ' gilt

$$\langle S_1, \sigma\rangle \Rightarrow^* \gamma \Leftrightarrow \langle S_2, \sigma\rangle \Rightarrow^* \gamma$$

und

$$\langle S_1, \sigma\rangle \Rightarrow^* \sigma' \Leftrightarrow \langle S_2, \sigma\rangle \Rightarrow^* \sigma'$$

(b) Es existiert eine undendliche Ableitungsfolge für $\langle S_1, \sigma\rangle$ gdw. es existiert eine unendliche Folge für $\langle S_2, \sigma\rangle$.

Beispiel. S_1; $(S_2$; $S_3)$ und $(S_1$; $S_2)$; S_3 sind semantisch äquivalent.

4.7 Semantische Funktion $\mathcal{S}_{\text{sos}}$

Definition 4.18. Definiere $\mathcal{S}_{\text{sos}} : \text{Stm} \to (\text{State} \to \text{State})$ als

$$\mathcal{S}_{\text{sos}}[\![S]\!](\sigma) = \begin{cases} \sigma' & \text{falls } \langle S, \sigma\rangle \Rightarrow^* \sigma' \\ \bot & \text{sonst} \end{cases}$$

Diese Funktion ist wohldefiniert, da SOS determiniert ist.

Satz 4.2. *Sei S eine Anweisung und seien σ, σ' Zustände. Dann gilt*

$$\mathcal{S}_{ns}[\![S]\!](\sigma) = \sigma' \quad \Leftrightarrow \quad \mathcal{S}_{sos}[\![S]\!](\sigma) = \sigma'$$

D. h. SOS und NS sind äquivalent für unser konkretes Beispiel *der* `while`*-Sprache.*

Beweis. Zwei Richtungen:

(a) "$\Rightarrow$": $\mathcal{S}_{\text{ns}}[\![S]\!](\sigma) = \sigma' \Rightarrow \mathcal{S}_{\text{sos}}[\![S]\!](\sigma) = \sigma'$

d. h. $\langle S, \sigma\rangle \to \sigma' \implies \langle S, \sigma\rangle \Rightarrow^* \sigma'$

Wir wissen $\langle S, \sigma\rangle \to \sigma'$, d. h. es existiert ein endlicher Ableitungsbaum T dafür. Führe Induktion nach der Tiefe von T durch.

Induktionsanfang: T hat Tiefe 0, d. h. T besteht nur aus einer Wurzel. Das bedeutet, $\langle S, \sigma\rangle \to \sigma'$ erfolgt durch Anwendung eines Axioms. Davon gibt es drei Stück: $[\text{zuw}_{\text{ns}}], [\text{skip}_{\text{ns}}], [\text{while}^{\mathbf{f}}_{\text{ns}}]$

Exemplarisch für $[\text{while}^{\mathbf{f}}_{\text{ns}}]$:

Wir wissen S hat die Form $\texttt{while}\ b\ \texttt{do}\ S$ und $\mathcal{B}[\![b]\!](\sigma) = \mathbf{f}$. D. h. T hat die Form

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S', \sigma\rangle \to \underbrace{\sigma'}_{\sigma}}$$

Jetzt gilt

$$\begin{aligned}
& \langle \texttt{while}\ b\ \texttt{do}\ S', \sigma\rangle \\
\overset{[\text{while}_{\text{sos}}]}{\Rightarrow}\ & \langle \texttt{if}\ b\ \texttt{then}\ (S';\ \texttt{while}\ b\ \texttt{do}\ S')\ \texttt{else skip}, \sigma\rangle \\
\overset{[\text{if}^{\mathbf{f}}_{\text{sos}}]}{\Rightarrow}\ & \langle \texttt{skip}, \sigma\rangle \qquad \text{da } \mathcal{B}[\![b]\!](\sigma) = \mathbf{f} \\
\overset{[\text{skip}_{\text{sos}}]}{\Rightarrow}\ & \sigma
\end{aligned}$$

Also $\langle \texttt{while}\ b\ \texttt{do}\ S', \sigma\rangle \Rightarrow^* \sigma$ wie gewünscht.

10.06.

Induktionsschritt:

(a) $[\text{while}^{\mathbf{w}}_{\text{ns}}]$

(b) $[\text{if}^{*}_{\text{ns}}]$

(c) $[\text{seq}_{\text{ns}}]$

Wir machen exemplarisch (a).

Es gilt $\langle S, \sigma\rangle \to \sigma'$, d. h. T:

$$\cfrac{\cfrac{T_1}{\langle S', \sigma\rangle \to \sigma''}\ , \quad \cfrac{T_2}{\langle \texttt{while b do S'}, \sigma''\rangle \to \sigma'}}{\langle \texttt{while b do S'}, \sigma\rangle \to \sigma'}$$

T_1 und T_2 existieren, da T existiert. Außerdem sind die Höhen von $T_1, T_2 <$ Höhe von T. Also folgt aus der IV, dass

wegen $\langle S', \sigma\rangle \to \sigma''$ auch

$$\langle S', \sigma\rangle \Rightarrow^* \sigma''$$

und wegen $\langle S', \sigma''\rangle \to \sigma'$ auch

$$\langle S, \sigma''\rangle \Rightarrow^* \sigma'$$

gilt.

$$\begin{aligned}
\langle \texttt{while b do S'}, \sigma\rangle \Rightarrow\ & \langle \texttt{if b then (S'; while b do S') else skip}, \sigma\rangle \\
\overset{[\text{if}^{\mathbf{w}}_{\text{sos}}]}{\Rightarrow}\ & \langle \texttt{S'; while b do S'}, \sigma\rangle \quad \text{weil } \mathcal{B}[\![b]\!](\sigma) = \mathbf{w} \\
\overset{\text{IV}}{\Rightarrow^*}\ & \sigma' \quad (\text{Übung})
\end{aligned}$$

(b) "$\Leftarrow$": Übung

$\square$

4.8 Erweiterungen der `while`-Sprache

4.8.1 Programmabbruch

Füge eine neue Anweisung `abort` zur Sprache hinzu. Die Bedeutung ist, dass das Programm fehlerhaft abgebrochen wird.

$$S \to \ldots \mid \texttt{abort}$$

Wir fügen für `abort` *keine neuen Schlussregeln* hinzu – weder für die NS noch die SOS).

Beispiel. Wir betrachten folgende Anweisung, in der die Schleife die Fakultät berechnet:

S = `if` $\neg(x \geqslant 1)$ `then abort else (y := 1; while` $\neg(x = 1)$ `do (y := y * x; x := x - 1))`

Wie sieht es bei der NS aus?

Sie $\sigma(x) = -1$. Wie sieht dann der Ableitungsbaum aus?

$$[\text{if}^{\mathbf{w}}_{\text{ns}}]\ \frac{\dfrac{T}{\langle \texttt{abort}, \sigma\rangle \to \sigma'}}{\langle S, \sigma\rangle \to \sigma'}$$

Dabei existiert T nicht! Daraus folgt

$$\mathcal{S}[\![S]\!](\sigma) = \bot$$

(für unseren konkreten Zustand σ).

S ist **semantisch äquivalent** zu

S' = `if` $\neg(x \geqslant 1)$ `then (while true do skip) else (y := 1; while` $\neg(x = 1)$ `do (y := y * x; x := x - 1))`

denn auch für `while true do skip` existiert auch kein endlicher Ableitungsbaum. D. h. in der natürlichen Semantik können wir nicht zwischen Abbruch und Endlossschleife unterscheiden.

Wie sieht es bei der SOS aus?

$$\langle S, \sigma\rangle \overset{[\text{if}^{\mathbf{w}}_{\text{sos}}]}{\Rightarrow} \langle \texttt{abort}, \sigma\rangle$$

Die rechte Konfiguration ist eine steckengebliebene Konfiguration.

$$\begin{aligned} \langle S', \sigma\rangle &\overset{[\text{if}^{\mathbf{w}}_{\text{sos}}]}{\Rightarrow} \langle \texttt{while true do skip}, \sigma\rangle \\ &\Rightarrow^* \langle \texttt{while true do skip}, \sigma\rangle \\ &\Rightarrow^* \ \ldots \end{aligned}$$

D. h. in der SOS sind S und S' **nicht semantisch äquivalent**.

4.8.2 Blöcke und lokale Variablen

Idee: Erlaube Definition lokaler Variablen innerhalb eines Bereichs.

$$\begin{aligned} S &\to \ldots \mid \texttt{begin}\ D_V\ S\ \texttt{end} \\ D_V &\to \texttt{var}\ \mathrm{Var}\ \texttt{:=}\ \texttt{A;}\ D_V \mid \varepsilon \end{aligned}$$

D_V entspricht der syntaktischen Kategorie Dec_V.

Beispiel. Betrachte

```
begin
    var y := 1;
    x := 1;
    begin
        var x := 2;
        y := x + 1  (* (1) 2 + 1 *)
    end
    y := y + x (* (2) 3 + 1 *)
end
```

In (1) überdeckt das lokale x das globale x *in diesem Block*. In (2) wird wieder das globale x verwendet.

4.8.3 Formalisierung lokaler Blöcke in der NS

Definition 4.19. Sei D_V eine Variablendeklaration. Die Menge der in ihr deklarierten Variablen $\mathrm{DV}(D_V)$ ist definiert als:

$$\begin{aligned} \mathrm{DV}(\varepsilon) &:= \varnothing \\ \mathrm{DV}(\texttt{var}\ \ x := a;\ \ D'_V) &:= \{x\} \cup \mathrm{DV}(D'_V) \end{aligned}$$

Wir führen eine neue Relation "$\to_D$" ein. Sie hat die Form

$$\langle D_V, \sigma\rangle \to_D \sigma'$$

wobei σ' ein lokaler Zustand ist.

4.8.4 Schlussregeln für D_V, $\to_D$ und Blöcke

$$\frac{}{\langle \varepsilon, \sigma\rangle \to_D \sigma}$$

$$\frac{\langle D_V, \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma)]\rangle \to_D \sigma'}{\langle \texttt{var}\ \ x := a;\ \ D_V, \sigma\rangle \to_D \sigma'}$$

$$\frac{\langle D_V, \sigma\rangle \to_D \sigma'', \langle S, \sigma''\rangle \to \sigma'}{\langle \texttt{begin}\ \ D_V\ \ S\ \ \texttt{end}, \sigma\rangle \to \sigma'[\mathrm{DV}(D_V) \mapsto \sigma]}$$

wobei

$$\sigma'[X \mapsto \sigma](x) = \begin{cases} \sigma(x) & \text{falls } x \in X \\ \sigma'(x) & \text{falls } x \notin X \end{cases}$$

d. h. alle lokal deklarierten Variablen werden auf ihren ursprünglichen Wert zurückgesetzt.

17.06.

4.8.5 Unterprogramme / Prozeduren / Subroutine / Methode

Wir wollen ein neues Sprachkonstrukt für `while` einführen: Prozeduren (wie in Pascal).

Dazu erweitern wir im ersten Schritt die Grammatik:

$$\begin{aligned} S &\rightarrow \ldots \mid \texttt{begin}\ D_V\ D_p\ S\ \texttt{end} \mid \texttt{call}\ p \\ D_p &\rightarrow \texttt{proc}\ p\ \texttt{is}\ S\texttt{;}\ D_p \mid \varepsilon \end{aligned}$$

p steht für den Bezeichner der Prozedur. D_p entspricht der syntaktischen Kategorie p_{Name}. D_p entspricht der syntaktischen Kategorie für Unterprogramm-Deklarationen.

Beispiel. Betrachte

```
begin
    var x := 0;
    proc p is x := x * 2;
    proc q is call p;

    begin
        var x := 5;
        proc p is x := x + 1;
        call q;
        y := x
    end
end
```

Listing 4: Programm mit Unterprogrammen

Frage: Was soll hier herauskommen?

Das hängt von den Regeln über die Gültigkeitsbereiche ab (*scope rules*).

Beim Aufruf von q gibt es zwei Möglichkeiten: statisches Scoping (Bindung durch Syntax) oder dynamisches Scoping (Bindung durch aktuellen Kontext bei der Ausführung).

Wir müssen uns also entscheiden: Wenn ein Bezeichner in einer Funktion verwendet wird, bezieht er sich auf die Umgebung im Quelltext (*statischer Gültigkeitsbereich*) oder auf die Umgebung zum Zeitpunkt der Programmausführung (*dynamischer Gültigkeitsbereich*)?

Diese Entscheidung kann man separat für jede Klasse von Bezeichnern treffen.

Unterprogramme \Variablen	statisch	dynamisch
statisch	$y = 5$	$y = 10$
dynamisch	$y = 6$	$y = 6$

Tabelle 1: Ergebnisse in Abhängigkeit von den scoping rules

4.8.6 Natürlichen operationelle Semantik für Unterprogramme mit dynamischem Gültigkeitsbereich

Um die Zuordnung von Prozedurname und Prozedurkörper (*procedure body*) zu verwalten, müssen wir die Übergangsrelation "$\rightarrow$" um einen *lokalen Kontext* erweitern. Dieser ist als Funktion

$$\mathrm{Env}_p : p_{\mathrm{Name}} \rightarrow \mathrm{Stm}$$

formalisiert. Die Übergangsrelation hat nun die Form

$$\mathrm{env}_p \vdash \langle S, \sigma \rangle \rightarrow \sigma'$$

mit $\mathrm{env}_p \in \mathrm{Env}_p$ (also eine Funktion), $S \in \mathrm{Stm}$ und $\sigma, \sigma' \in \mathrm{State}$. Das neue Symbol "$\vdash$" bedeutet "im Kontext von".

Interpretation: Im lokalen Kontext env_p und mit Startzustand σ terminiert die Anweisung S nach endliche vielen Schritten und erreicht den Endzustand σ'.

Bemerkung. Für statische Gültigkeitsbereiche müssten wir auch für Variablen einen Kontext erzeugen.

4.8.7 Schlussregeln für Unterprogramme mit dynamischem Gültigkeitsbereich

Die meisten Schlussregeln werden übernommen, nur dass env_p "mitgeschleppt" wird.

Beispiel. [$\mathrm{seq_{ns}}$]

$$\frac{\mathrm{env}_p \vdash \langle S_1, \sigma \rangle \rightarrow \sigma', \mathrm{env}_p \vdash \langle S_2, \sigma' \rangle \rightarrow \sigma''}{\mathrm{env}_p \vdash \langle S_1;\ S_2, \sigma \rangle \rightarrow \sigma''}$$

Wichtig / interessant sind nun [$\mathrm{block_{ns}}$] und [$\mathrm{call_{ns}}$] (neu):

(a) [$\mathrm{block_{ns}}$]

$$\frac{\langle D_V, \sigma \rangle \rightarrow_D \sigma'', \quad \mathrm{akt}_p(D_p, \mathrm{env}_p) \vdash \langle S, \sigma'' \rangle \rightarrow \sigma'}{\mathrm{env}_p \vdash \langle \texttt{begin}\ D_V\ \ D_p\ \ S\ \ \texttt{end}, \sigma \rangle \rightarrow \sigma'[DV(D_V) \mapsto \sigma]}$$

Dabei bedeutet akt_p: "Passe den lokalen Kontext an!" und ist wie folgt definiert:

$$\begin{aligned} \mathrm{akt}_p : D_p \times \mathrm{Env}_p &\rightarrow \mathrm{Env}_p \\ \mathrm{akt}_p(\varepsilon, \mathrm{env}_p) &= \mathrm{env}_p \\ \mathrm{akt}_p(\texttt{proc}\ p\ \texttt{is}\ S;\ D_p, \mathrm{env}_p) &= \mathrm{akt}_p(D_p, \mathrm{env}_p[p \mapsto S]) \end{aligned}$$

(b) [$\mathrm{call_{ns}}$]

$$\frac{\mathrm{env}_p \vdash \langle S, \sigma \rangle \rightarrow \sigma'}{\mathrm{env}_p \vdash \langle \texttt{call}\ \ p, \sigma \rangle \rightarrow \sigma'}$$

wobei $\mathrm{env}_p(p) = S$, d. h. nur wenn p im aktuellen Kontext deklariert ist. Ansonsten kann man diese Regel nicht anwenden, d. h. die Anweisung würde dann steckenbleiben.

Bemerkung. Bei $\rightarrow_D$ benötigen wir keinen lokalen Kontext.

Alles ist dynamisch, d. h. env_p und σ hängen vom aktuellen Programmzustand ab.

Mit mehr Arbeit ($\rightsquigarrow$ zusätzliche Indirektion und umfangreicherer lokaler Kontext) kann man auch statische Gültigkeitsbereiche modellieren.

24.06.

5 Denotationelle Semantik

Bemerkung (Ziel). Die Definition einer semantischen Funktion direkt ohne Umweg über eine simulierte Programmausführung (d. h. ohne eine Übergangsrelation).

Dabei ist die Zusammengesetztheit (*compositionality*) bei der Definition der semantischen Funktion für ein Sprachkonstrukt wichtig, d. h. man darf nur die Werte der semantischen Funktion für die Bestandteile des Sprachkonstrukts verwenden.

Beispiel.

$$\mathcal{A} : \text{AExp} \to \mathbb{Z}, \quad \mathcal{B} : \text{BExp} \to \{\mathbf{w}, \mathbf{f}\}$$

sind denotationell definiert.

Beispiel (Gegenbeispiel)**.** [$\text{while}_{\text{ns}}^{\mathbf{w}}$]:

$$\frac{\langle S, \sigma\rangle \to \sigma'', \langle \texttt{while}\ b\ \texttt{do}\ S, \sigma''\rangle \to \sigma'}{\langle \underbrace{\texttt{while}\ b\ \texttt{do}\ S}_{\text{gleiches Konstrukt}}, \sigma\rangle \to \sigma'}$$

5.1 Direkte Definition der semantischen Funktion

Definition 5.1. Wir definieren $\mathcal{S}_{\text{ds}} : \text{Stm} \to (\text{State} \to \text{State})$.

Die grundlegenden Definition bleiben gleich:

$$\begin{aligned}
\mathcal{S}_{\text{ds}}[\![\texttt{x := a}]\!](\sigma) &= \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma)] \\
\mathcal{S}_{\text{ds}}[\![\texttt{skip}]\!](\sigma) &= \sigma \\
\mathcal{S}_{\text{ds}}[\![S_1\texttt{;}\ S_2]\!](\sigma) &= (\underbrace{\mathcal{S}_{\text{ds}}[\![S_2]\!]}_{\text{State}\to\text{State}} \circ \mathcal{S}_{\text{ds}}[\![S_1]\!])(\sigma)
\end{aligned}$$

Achtung: Die Zustandsüberführungsfunktionen können eventuell nur partiell definiert sein. Dann liefert die Komposition auch nur $\perp$.

Zur Definition von Verzweigungen ($\rightsquigarrow$ `if`) benutzen wir eine Hilfsfunktion *cond* (siehe Definition 5.2). Damit ist die Definition von Verzweigungen wie folgt:

$$\mathcal{S}_{\text{ds}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{then}\ S_2]\!] = \text{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\text{ds}}[\![S_1]\!], \mathcal{S}_{\text{ds}}[\![S_2]\!])$$

Definition 5.2 (cond)**.**

$$\text{cond} : (\text{State} \to \{\mathbf{w}, \mathbf{f}\}) \times (\text{State} \to \text{State}) \times (\text{State} \to \text{State}) \to (\text{State} \to \text{State})$$

$$\text{cond}(p, f_1, f_2) = \begin{cases} f_1(\sigma) & \text{falls } p(\sigma) = \mathbf{w} \\ f_2(\sigma) & \text{falls } p(\sigma) = \mathbf{f} \end{cases}$$

Herausforderung: Definition von $\mathcal{S}_{\text{ds}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ unter Berücksichtigung der Zusammengsetztheit.

Beobachtung: `while` b `do` S und `if` b `then (S; while` b `do` S`) else skip` sollen semantisch äquivalent sein.

Also eigentlich wollen wir schreiben:

$$\mathcal{S}_{\text{ds}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \text{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\text{ds}}[\![\underbrace{\texttt{while}\ b\ \texttt{do}\ S}_{\text{gleiches Konstrukt}}]\!] \circ \mathcal{S}_{\text{ds}}[\![S]\!], \text{id})$$

Das verletzt aber die Zusammengesetztheit.

Wenn wir diese Gleichung betrachten, erkennen wir

Bemerkung (Erkenntnis). $\mathcal{S}_{\text{ds}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ ist die Lösung der Gleichung

$$f = \text{cond}(\mathcal{B}[\![b]\!], f \circ \mathcal{S}_{\text{ds}}[\![S]\!], \text{id})$$

Wie lösen wir diese Gleichung?

5.1.1 Problemtransfer zu Fixpunkt eines Funktionals

Wir überführen das Problem des Lösens der Gleichung zu einem Problem des Findens eines Fixpunktes.

Definition 5.3 (Funktional)**.** Dafür definieren wir ein *Funktional* (Argument und Ergebnis sind Funktionen):

$$F : (\text{State} \to \text{State}) \to (\text{State} \to \text{State})$$

durch

$$F(f) = \text{cond}(\mathcal{B}[\![b]\!], f \circ \mathcal{S}_{\text{ds}}[\![S]\!], \text{id})$$

Ein *Fixpunkt* von F ist ein $f^* : \text{State} \to \text{State}$ mit $F(f^*) = f^*$, d. h. Stellen an denen die Funktion Werte auf sich selbst abbildet.

Durch diese Herangehensweise, können wir die Funktion auch im Umfeld der Lösung betrachten oder sie benutzen, um sich der Lösung anzunähern.

Definition 5.4. Wir definieren einen *Fixpunktoperator*

$$\text{FIX} : ((\text{State} \to \text{State}) \to (\text{State} \to \text{State})) \to (\text{State} \to \text{State})$$

der einem Funktional F einen Fixpunkt f^* zuordnet.

Dann können wir schreiben

$$\mathcal{S}_{\text{ds}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \text{FIX}(F)$$

wobei

$$F : f \mapsto \text{cond}(\mathcal{B}[\![b]\!], f \circ \mathcal{S}_{\text{ds}}[\![S]\!], \text{id})$$

Das heißt die Semantik der `while`-Schleife ist der Fixpunkt des Funktionals.

Frage. Was können wir über FIX(f) sagen?

(a) Existiert für *jedes* Funktional $G : (\text{State} \to \text{State}) \to (\text{State} \to \text{State})$ ein Fixpunkt?

Nein.

Betrachte G mit

$$G(g) = \begin{cases} g_1 & \text{falls } g = g_2 \\ g_2 & \text{sonst} \end{cases}$$

für $g_1 \neq g_2$.

Aber vielleicht hat unser Funktional F immer einen Fixpunkt.

(b) Falls es einen Fixpunkt gibt, ist dieser eindeutig?

Im Allgemeinen nein.

Betrachte G mit

$$G(g) = g$$

Hier sind alle g Fixpunkte.

Aber vielleicht hat unser Funktional F immer höchstens einen Fixpunkt.

Wir müssen uns also anschauen, was der Fixpunktoperator FIX mit unserem speziellen F macht.

Bemerkung (Intuition). Wie löst man eigentlich eine Fixpunkt-Gleichung $f = F(f)$?

Fixpunktiteration: Start mit

$$\begin{aligned} &f_0 \\ &f_1 = F(f_0) \\ &f_2 = F(f_1) \\ &f_3 = F(f_2) \\ &\dots \end{aligned}$$

Beispiel (Kein Fixpunkt)**.**

$$\begin{aligned} x &= 2x - 7 \\ x_0 &= 20 \\ x_1 &= 33 \\ x_2 &= 59 \\ x_3 &= 111 \\ &\dots \end{aligned}$$

01.07.

5.2 Der Fixpunktoperator

Bemerkung (Recap)*.* Aus der letzten Vorlesung:

Aufgabe: Definiere die Semantik von `while` b `do` S denotationell.

Idee: Betrachte Funktional F (siehe Definition 5.3). Dann sollte die Zustandsüberführungsfunktion ein Fixpunkt von F sein, d. h. $f^* = F(f^*)$.

Frage. Woher wissen wir, dass F einen Fixpunkt besitzt? Wie sieht dieser Fixpunkt aus?

Versuch einer Visualisierung:

Wir haben $\mathcal{B}[\![b]\!]$ (Punkte sind Zustände mit Wahrheitswerten) und $\mathcal{S}[\![S]\!]$ (partielle Zustandsüberführungsfunktion, gelb), eine Funktion (Kandidat für Fixpunkt, blau) und das f-Funktional $F(f)$ (grün).

F erzeugt also eine neue Zustandsüberführungsfunktion, die für Zustände mit **f** auf den Zustand selbst abbildet und für Zustände mit **w** durch $f \circ \mathcal{S}$ abbildet, d. h. erst dem gelben, dann dem blauen Pfeil folgt.

Die Aufgabe des Findens eines Fixpunktes bedeutet also, ein f (blau) zu finden, sodass nach Anwendung von $F(f)$ (grün) die gleichen Pfeile entstehen wie in f.

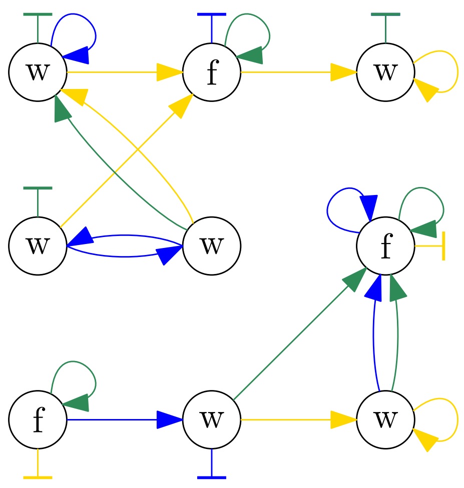
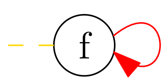


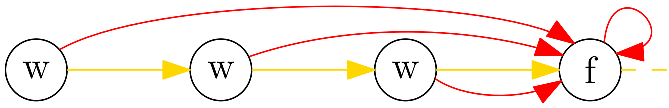
Abbildung 2: Visualisierung der Funktionsverkettung durch F

5.2.1 Wie muss ein Fixpunkt aussehen?

(a) Für alle Zustände mit Wahrheitswert **f** muss der Fixpunkt eine Schleife (id) liefern.



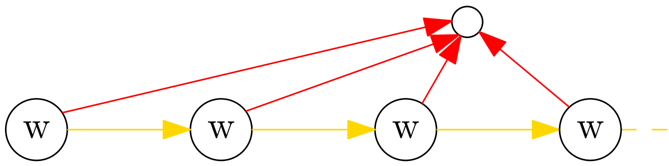
(b) Es gibt Zustände mit Wahrheitswert **w**, die nach endlich vielen Schritten mit $\mathcal{S}$ (gelb) einen Zustand mit Wahrheitswert **f** erreichen.



(c) Es gibt Zustände mit Wahrheitswert **w**, die nicht nach endliche vielen Schritten einen Zustand mit Wahrheitswert **f** erreichen.

(a) $w \to w \to w \to \dots$

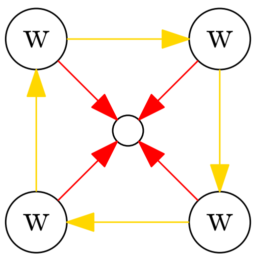
```
x := 1;
while true do
    x := x + 1
```



Hier müssen alle Pfeile zum selben Zustand zeigen, egal welcher genau das ist.

(b) $w_0 \to w_1 \to w_2 \to w_3 \to w_0$

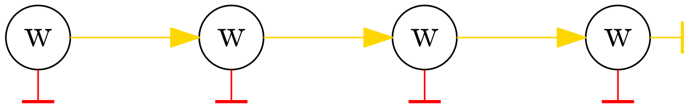
```
x := 0;
while x != -1 do
    x := (x + 1) mod 4
```



Hier müssen ebenfalls alle Pfeile zum selben Zustand zeigen, egal welcher genau das ist.

(c) $w \to w \to w \to \bot$

```
x := 1;
while x < 20 do
    x := x + 1;
    if x = 10 then
        while true do skip
    else
        ...
```



Für die Fälle (a) und (b) ist der Fixpunkt also fix aber beliebig (inklusive $\perp$).

Das Ergebnis unserer informellen Überlegung ist:

- Wir erwarten, dass *immer ein Fixpunkt existiert.*
- Ein solcher Fixpunkt ist nicht immer eindeutig. Für Zustände, bei denen die `while`-Schleife nicht terminiert, kann es ggf. mehrere Möglichkeiten für einen Fixpunkt geben.

  In diesem Fall hätten wir gern den Fixpunkt, der $\perp$ für nicht-terminierende Schleifen liefert.

5.2.2 Fixpunktiteration

Bemerkung (Idee). Finde den richtigen Fixpunkt durch Fixpunktiteration. Starte mit der Zustandsüberführungsfunktion

$$f_\perp : f_\perp(\sigma) = \perp \quad \forall \sigma \in \text{State}$$

welche F wiederholt auf $f_\perp$ an, schaue was passiert.

$\Rightarrow$ Der "Grenzwert" ist der gewünschte Fixpunkt.

Z. B. bei (c.a) und (c.b) bleibt die Funktion nach wiederholter Anwendung überall $\perp$.

Definition 5.5 (Fixpunktoperator)**.**

$$\begin{aligned} f_0 &= f_\perp \\ f_n &= F(f_{n-1}) \quad n \geqslant 1 \\ \text{FIX}(f) &:= \lim_{n\to\infty} f_n \end{aligned}$$

Aber wie ist der Grenzwert in diesem Kontext definiert?

5.3 Relationen und Ordnungen

Definition 5.6. Sei $\mathcal{F} = \{f \mid f : \text{State} \to \text{State}\}$ die Menge aller *partiellen* Zustandsüberführungsfunktionen. Wir definieren auf $\mathcal{F}$ eine Relation $\sqsubseteq$ durch

$$f \sqsubseteq g :\Longleftrightarrow \forall \sigma, \sigma' \in \text{State} : f(\sigma) = \sigma' \Rightarrow g(\sigma) = \sigma'$$

d. h. überall, wo f definiert ist, ist auch g definiert und liefert denselben Wert.

Beispiel. $g_1, g_2, g_3, g_4 : \text{State} \to \text{State}$

$$g_1(\sigma) = \sigma \quad \forall \sigma \in \text{State}$$
$$g_2(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(x) \geqslant 0 \\ \bot & \text{sonst} \end{cases}$$
$$g_3(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(x) \leqslant 0 \\ \bot & \text{sonst} \end{cases}$$
$$g_4(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(x) = 0 \\ \bot & \text{sonst} \end{cases}$$

Behauptung:

$$g_4 \sqsubseteq g_2 \sqsubseteq g_1$$
$$g_4 \sqsubseteq g_3 \sqsubseteq g_1$$
$$g_4 \sqsubseteq g_1 \tag{*}$$

$(*)$ folgt aus Transitivität da die Relation eine Ordnungsrelation ist, aber das haben wir noch nicht bewiesen.

Jedoch sind g_2 und g_3 nicht vergleichbar.

Bemerkung (Fakt). $\sqsubseteq$ ist eine Ordnungsrelation auf $\mathcal{F}$. Das bedeutet, sie ist

- reflexiv: $f \sqsubseteq f$
- transitiv: $f \sqsubseteq g \wedge g \sqsubseteq h \Rightarrow f \sqsubseteq h$
- antisymmetrisch: $f \sqsubseteq g \wedge g \sqsubseteq f \Rightarrow f = g$

$(\mathcal{F}, \sqsubseteq)$ ist eine *Halbordnung* (poset bzw. partially ordered set).

Definition 5.7. Ein Element $f \in \mathcal{F}$ heißt *Minimum*, falls für alle $g \in \mathcal{F}$ gilt

$$f \sqsubseteq g$$

Beobachtungen. Im Allgemeinen, muss es kein Minimum in einem poset geben (z. B. unendliche Ordnung oder mehrere nicht vergleichbare minimale Elemente). Wenn ein *Minimum* existiert, dann ist es *eindeutig* (wegen Antisymmetrie).

Aber $(\mathcal{F}, \sqsubseteq)$ hat ein Minimum und zwar $f_\bot$.

08.07.

Bemerkung (Ziel für den Rest der Vorlesung)**.** Konvergenz der Fixpunktiteration beweisen.

Definition 5.8 (Obere Schranke, Supremum)**.** Sei $\mathcal{G} \subseteq \mathcal{F}$, dann ist auch $(\mathcal{G}, \sqsubseteq)$ eine Halbordnung.

Ein Element $f \in \mathcal{F}$ heißt *obere Schranke* von $\mathcal{G}$, falls

$$\forall g \in \mathcal{G} : g \sqsubseteq f$$

Eine obere Schranke von $\mathcal{G}$ heißt *Supremum* "$\sup \mathcal{G}$", falls für alle oberen Schranken f' von $\mathcal{G}$ gilt $f \sqsubseteq f'$ d. h. es ist die kleinste obere Schranke.

Falls ein *Supremum* existiert, so ist es *eindeutig*. Falls die obere Schranke innerhalb von $\mathcal{G}$ liegt, so ist sie auch das Maximum und Supremum von $\mathcal{G}$.

Definition 5.9. Sei $\mathcal{G} \subseteq \mathcal{F}$. Wir nennen $\mathcal{G}$ eine *Kette* (chain), falls $\mathcal{G}$ total geordnet ist, d. h.

$$\forall g_1, g_2 \in \mathcal{G} : g1 \sqsubseteq g_2 \vee g_2 \sqsubseteq g_1$$

Beispiel. Folgende Beispiele sind Ketten:

- Die leere Menge $\emptyset \subseteq \mathcal{F}$ ist eine Kette.
- Jede 1-elementige Menge ist eine Kette.
- Sei $x \in \mathrm{Var}$ eine Variable. Für $n \in \mathbb{N}$ definiere wir $g_n \in \mathcal{F}$ durch

$$g_n : \mathrm{State} \to \mathrm{State}$$

$$g_n(\sigma) = \begin{cases} \perp & \text{falls } \sigma(x) > n \\ \sigma[x \mapsto -1] & \text{falls } \sigma(x) \in \{0, \dots, n\} \\ \sigma & \text{falls } \sigma(x) < 0 \end{cases}$$

Die Menge $\{g_n \mid n \in \mathbb{N}\}$ ist eine Kette in $(\mathcal{F}, \sqsubseteq)$, denn es gilt

$$g_n \sqsubseteq g_m \Leftrightarrow n \leqslant m$$

```
while x >= 0 do
    x := x - 1
```

Listing 5: Snippet für g_n

Ziel: Die Funktion

$$g_\infty : \mathrm{State} \to \mathrm{State}$$

$$g_\infty = \begin{cases} \sigma[x \mapsto -1] & \text{falls } \sigma(x) \geqslant 0 \\ \sigma & \text{sonst} \end{cases}$$

ist die Semantik von Listing 1.

Nun sehen wir, dass

$$g_\infty = \sup\{g_n \mid n \in \mathbb{N}\}$$

Definition 5.10 (Kettenvollständigkeit)**.** Eine Halbordnung heißt *kettenvollständig* (ccpo: chain-complete-partial order), falls jede Kette ein Supremum besitzt.

Beispiel. $(\mathbb{N}, \leqslant)$ ist nicht kettenvollständig. Die Menge der geraden Zahlen $\{2, 4, \dots\}$ ist eine Kette ohne Supremum.

$(\mathcal{P}(\{1, 2, 3\}), \subseteq)$ ist kettenvollständig.

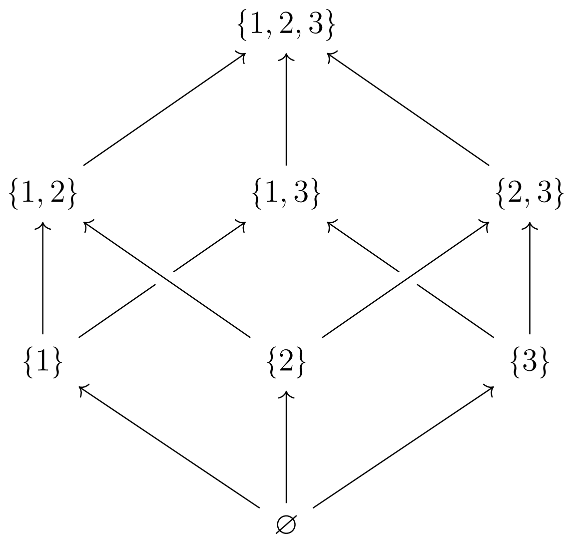


Abbildung 3: Hasse-Diagramm für $(\mathcal{P}(\{1, 2, 3\}), \subseteq)$

$(\mathcal{P}(\mathbb{N}), \subseteq)$ ist auch kettenvollständig. Sei $X \in (\mathcal{P}(\mathbb{N}), \subseteq)$ eine Kette, also eine Menge von Teilmengen von $\mathbb{N}$, sodass $\forall A, B \in X : A \subseteq B \vee B \subseteq A$.
Es ist $\sup X = \bigcup_{A \in X} A = Z$, da

(a) Sei $A = X$. Dann ist $A \subseteq Z$.

(b) Z ist kleinste obere Schranke. Sei Y obere Schranke von X.

Zu zeigen: $Z \subseteq Y$.

Nimm $z \in Z$, zeige, dass $z \in Y$.

$$z \in Z \Rightarrow \exists A \subseteq Z \text{ mit } z \in A$$

Da Y obere Schranke von X ist und $a \in X$, muss $A \subseteq Y$. Also folgt, $z \in Y$.

Bemerkung (Fakt)*.* Sei $(\mathcal{D}, \sqsubseteq)$ ccpo. Dann besitzt $\mathcal{D}$ ein Minimum, nämlich

$$d_\perp = \sup \varnothing$$

Beweis. Sei $d_\perp = \sup \varnothing$. $d_\perp$ existiert nach Annahme. Und sei $d \in \mathcal{D}$. Nach Definition ist d eine obere Schranke von $\varnothing$. Da $d_\perp = \sup \varnothing$, muss $d_\perp \sqsubseteq d$ sein, d. h. $d_\perp$ ist Minimum. □

Satz 5.1. *$(\mathcal{F}, \sqsubseteq)$ ist kettenvollständig.*

Sei $\mathcal{G} \subseteq \mathcal{F}$ eine Kette. Dann ist $\sup \mathcal{G}$ die Funktion mit

$$\operatorname{graph}(\sup \mathcal{G}) = \bigcup \{\operatorname{graph}(g) \mid g \in \mathcal{G}\}$$

d. h.

$$(\sup \mathcal{G})(\sigma) = \sigma' \Leftrightarrow \exists g \in \mathcal{G} : g(\sigma) = \sigma'$$

Beweis. Wir müssen die folgenden Eigenschaften beweisen:

(a) Wohldefiniertheit

Seien $g_1, g_2 \in \mathcal{G}$ und sei $\sigma \in \text{State}$, sodass $g_1(\sigma) \neq \perp \neq g_2(\sigma)$. Da $\mathcal{G}$ eine Kette ist, muss gelten $g_1 \sqsubseteq g_2 \vee g_2 \sqsubseteq g_1$. In beiden Fällen folgt $g_1(\sigma) = g_2(\sigma)$. Also ist der Wert von $(\sup \mathcal{G})(\sigma)$ wohldefiniert.

(b) obere Schranke

Aus der Definition folgt direkt, dass $(\sup \mathcal{G})$ eine obere Schrank ist:

$$g \in \mathcal{G},\ \sigma, \sigma' \in \text{State},\ g(\sigma) = \sigma' \implies (\sup \mathcal{G})(\sigma) = \sigma'$$

(c) kleinste obere Schranke

$(\sup \mathcal{G})$ ist obere Schranke. Sei h eine obere Schranke.

Zu zeigen: $(\sup \mathcal{G}) \sqsubseteq h$.

Sei $\sigma, \sigma' \in \text{State}$. $(\sup \mathcal{G})(\sigma) = \sigma'$, d. h. $\exists g \in \mathcal{G} : g(\sigma) = \sigma'$, d. h. h ist obere Schranke, d. h. $g \sqsubseteq h$, also muss $h(\sigma) = \sigma'$.

□

Idee der Fixpunktiteration:

$$\begin{aligned} f_0 &= f_\perp \\ f_1 &= F(f_0) \\ &\vdots \\ f_\infty &= \sup \{f_n \mid n \in \mathbb{N}\} \end{aligned}$$

Dann soll gelten: $f_\infty = F(f_\infty)$.

Wenn also gelten würde

$$F\left(\lim_{n\to\infty} f_n\right) = \lim_{n\to\infty} F(f_n)$$

wären wir fertig. Das ist die *Stetigkeit*.

15.07.

Bemerkung (Letztes Mal). Satz 5.1: $(\mathcal{F}, \sqsubseteq)$ ist kettenvollständig.

Was bisher fehlt ist ein in Konzept von Stetigkeit, sodass man Folgendes schreiben kann:

$$\lim_{n \to \infty} f(x_n) = f\left(\lim_{n \to \infty} x_n\right)$$

Definition 5.11 (Monotonie). Eine totale Funktion $f : D \to Z$ heißt *monoton*, falls für alle $d_1, d_2 \in D$ gilt

$$d_1 \sqsubseteq d_2 \Rightarrow f(d_1) \sqsubseteq' f(d_2)$$

Beobachtungen. Wir beobachten folgende Eigenschaften:

(a) Seien $(\mathcal{D}, \sqsubseteq), (\mathcal{D}', \sqsubseteq'), (\mathcal{D}'', \sqsubseteq'')$ ccpos und $f_1 : \mathcal{D} \to \mathcal{D}'$ und $f_2 : \mathcal{D}' \to \mathcal{D}''$ monoton Funktionen.

Dann ist auch $f_1 \circ f_2$ monoton.

(b) Seien $(\mathcal{D}, \sqsubseteq), (\mathcal{D}', \sqsubseteq')$ ccpos und $f : \mathcal{D} \to \mathcal{D}'$ eine monotone Funktion. Sei $\mathcal{K}$ eine Kette. Dann ist $f(\mathcal{K})$ eine Kette in $\mathcal{D}'$. Wenn $\mathcal{D}$ endlich ist, dann ist das Supremum von $\mathcal{D}'$:

$$\sup{}' \mathcal{D}' = f(\sup \mathcal{D})$$

Im Allgemeinen gilt: $f(\sup S) \sqsubsetneq' \sup' f(S)$

Definition 5.12 (Stetigkeit). Seien $(\mathcal{D}, \sqsubseteq), (\mathcal{D}', \sqsubseteq')$ ccpos und $f : \mathcal{D} \to \mathcal{D}'$ heißt *stetig* wenn:

(a) f ist monoton.

(b) Für alle nichtleeren Ketten $\mathcal{S} \in \mathcal{D}$ gilt $\sup' f(\mathcal{S}) = f(\sup \mathcal{S})$.

Beobachtung. Seien $(\mathcal{D}, \sqsubseteq), (\mathcal{D}', \sqsubseteq'), (\mathcal{D}'', \sqsubseteq'')$ ccpos und seien die beiden Funktionen $f : \mathcal{D} \to \mathcal{D}'$ und $f' : \mathcal{D}' \to \mathcal{D}''$ stetig.

Dann ist $f' \circ f : \mathcal{D} \to \mathcal{D}''$ stetig.

Satz 5.2. *Sei $(\mathcal{D}, \sqsubseteq)$ ccpo mit Minimum d_h und sei $F : \mathcal{D} \to \mathcal{D}$ eine stetige Funktion. Setze $f_0 = d_\perp$ und $f_n = f(f_{n-1})$.*

(a) Dann ist $\{f_n \mid n \in \mathbb{N}\}$ eine Kette.

(b) Sei $\mathrm{FIX}(F) = \sup\{f_n \mid n \in \mathbb{N}\}$. Dann gilt $F(\mathrm{FIX}(F)) = \mathrm{FIX}(F)$.

(c) Außerdem gilt $\forall g \in \mathcal{D}, F(g) = g : g \geqslant \mathrm{FIX}(F)$.

Beweis. Wir führen den Beweis stückweise.

(a) $\{f_n \mid n \in \mathbb{N}\}$ ist eine Kette.
Seien f_n, f_m mit $m > n$. Zu zeigen $f_n \sqsubseteq f_m$.

Setze $a = n - m > 0$. Da $f_0 = d_\perp$ Minimum ist, gilt $f_0 \sqsubseteq f_a$. Da F stetig ist, folgt

$$\begin{aligned} F(f_0) &\sqsubseteq F(f_n) \\ \Rightarrow f_1 &\sqsubseteq f_{a+1} \\ &n\text{-mal} \ldots \\ \Rightarrow f_n &\sqsubseteq f_{a+n} \end{aligned}$$

(b) $\mathrm{FIX}(F)$ ist Fixpunkt von F.

$$\begin{aligned} F(\mathrm{FIX}(F)) &= F(\sup\{f_n \mid n \in \mathbb{N}\}) && \text{(Def)} \\ &= \sup F(\{f_n \mid n \in \mathbb{N}\}) && \text{(Stetigkeit)} \\ &= \sup\{F(f_n) \mid n \in \mathbb{N}\} \\ &= \sup\{f_{n+1} \mid n \in \mathbb{N}\} && \text{(Def)} \\ &= \sup(\{f_{n+1} \mid n \in \mathbb{N}\} \cup \{d_\perp\}) && \text{(Minimum)} \\ &= \mathrm{FIX}(F) && \text{(Def)} \end{aligned}$$

(c) $\mathrm{FIX}(f)$ ist minimaler Fixpunkt.

Sei $g \in \mathcal{D}$ und $F(g) = g$. Es gilt $g \sqsupseteq d_\perp$ ($d_\perp$ ist Minimum).

Also auch $F(g) \sqsupseteq F(d_\perp)$ bzw. $g \sqsupseteq f_1$. Also auch $F(g) \sqsupseteq F(f_1)$ bzw. $g \sqsupseteq f_2$ usw.

Also gilt nach Induktion $g \sqsupseteq f_n, n \in \mathbb{N}$. g ist eine obere Schranke von $\{f_n \mid n \in \mathbb{N}\}$. Nach Definition vom Supremum, ist g die kleinste obere Schranke.

$$\mathrm{FIX}(f) = \sup\{f_n \mid n \in \mathbb{N}\} \sqsubseteq g$$

□

Zurück zum ursprünglichem Ziel: Nun möchten wir schreiben:

$$\mathcal{S}_{\mathrm{ds}}[\![\texttt{while } b \texttt{ do } S]\!] = \mathrm{FIX}(F)$$

wobei $F(g) = \mathrm{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\mathrm{ds}}[\![S]\!], \mathrm{id})$ und $f \circ g : \mathrm{State} \to \mathrm{State}$.

Nach dem Satz müssen wir jetzt nur noch überprüfen, dass F stetig ist. Das funktioniert so: Wir beobachten, dass sich F schreiben lässt als $F = F_2 \circ F_1$, wobei $F_1(g) = g \circ \mathcal{S}_{\mathrm{ds}}[\![S]\!]$ und $F_2(g) = \mathrm{cond}(\mathcal{B}[\![b]\!], g, \mathrm{id})$.

$\mathcal{S}_{\mathrm{ds}}[\![\,\cdot\,]\!]$ ist schwierig, also beweisen wir die Stetigkeit getrennt.

Skizze. Zu zeigen:

(a) F_2 ist stetig, durch detaillierte Analyse.

(b) F_1 ist stetig. Beobachtung: Für jedes feste $g_0 : \text{State} \to \text{State}$:

Ist $F_1(f) = f \circ g_0$ stetig?

$\Rightarrow F_2 \circ F_1$ ist stetig. □

Satz 5.3. $\mathcal{S}_{ds}[\![\cdot]\!]$ *definiert eine semantische Funktion auf der Menge aller* `while`*-Programme.*

Skizze. Durch strukturelle Induktion nach der Anweisung S. □

5.4 Eigenschaften der denotationellen Semantik

Seien S_1, S_2 Anweisungen. Wir sagen, sie sind semantisch äquivalent, wenn

$$\mathcal{S}_{\text{ds}}[\![S_1]\!] = \mathcal{S}_{\text{ds}}[\![S_2]\!]$$

(Vergleich als mathematische Funktionen).

Beispiel. S und $S;$ `skip` sind semantisch äquivalent.

$$\begin{aligned}\mathcal{S}_{\text{ds}}[\![S_1]\!] &= \mathcal{S}_{\text{ds}}[\![S_1; \texttt{skip}]\!]\\ &= \mathcal{S}_{\text{ds}}[\![\texttt{skip}]\!] \circ \mathcal{S}_{\text{ds}}[\![S_1]\!]\\ &= \text{id} \circ \mathcal{S}_{\text{ds}}[\![S_1]\!]\\ &= \mathcal{S}_{\text{ds}}[\![S_1]\!] \quad \checkmark\end{aligned}$$

oder auch

$$S_1; (S_2; S_3) \text{ und } (S_1; S_2); S_3$$

durch Assoziativität der Funktionskomposition. Und auch

$$\texttt{while } b \texttt{ do } S \text{ und } \texttt{if } b \texttt{ then } (S\texttt{; while } b \texttt{ do } S) \texttt{ else skip}$$

Satz 5.4. *Sei S eine Anweisung in der* `while`*-Sprache, dann gilt:*

$$\mathcal{S}_{ds}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]$$

Skizze. Durch Fallunterscheidung/Induktion.

Der Beweis verwendet die Antisymmetrie von $(\mathcal{F}, \sqsubseteq)$. Wir zeigen, dass

$$\mathcal{S}_{\text{ds}}[\![\cdot]\!] \sqsubseteq \mathcal{S}_{\text{sos}}[\![\cdot]\!] \text{ und } \mathcal{S}_{\text{sos}}[\![\cdot]\!] \sqsubseteq \mathcal{S}_{\text{ds}}[\![\cdot]\!]$$

(a) $\mathcal{S}_{\text{sos}}[\![\,\cdot\,]\!] \sqsubseteq \mathcal{S}_{\text{ds}}[\![\,\cdot\,]\!]$
Sei S eine Anweisung, dann gilt $\mathcal{S}_{\text{sos}}[\![\,\cdot\,]\!] \sqsubseteq \mathcal{S}_{\text{ds}}[\![\,\cdot\,]\!]$, d. h. für alle σ, σ' gilt: Falls $\mathcal{S}_{\text{sos}}[\![S]\!](\sigma) = \sigma'$, dann auch $\mathcal{S}_{\text{ds}}[\![S]\!](\sigma) = \sigma'$.

Nach Definition gilt

$$\mathcal{S}_{\text{sos}}[\![S]\!](\sigma) = \sigma' \Longleftrightarrow \langle S, \sigma\rangle \Rightarrow^* \sigma'$$

Dann zeigen wir: Wenn $\langle S, \sigma\rangle = \sigma'$, dann gilt $\mathcal{S}_{\text{ds}}[\![S]\!](\sigma) = \sigma'$. Und wenn gilt $\langle S, \sigma\rangle = \langle S', \sigma'\rangle$, dann gilt $\mathcal{S}_{\text{ds}}[\![S]\!](\sigma) = \mathcal{S}_{\text{ds}}[\![S']\!](\sigma')$.

(b) $\mathcal{S}_{\text{ds}}[\![\,\cdot\,]\!] \sqsubseteq \mathcal{S}_{\text{sos}}[\![\,\cdot\,]\!]$

Sei S eine Anweisung, dann gilt $\mathcal{S}_{\text{ds}}[\![\,\cdot\,]\!] \sqsubseteq \mathcal{S}_{\text{sos}}[\![\,\cdot\,]\!]$. Also gilt

$$\mathcal{S}_{\text{ds}}[\![S]\!](\sigma) = \sigma' \implies \mathcal{S}_{\text{sos}}[\![S]\!](\sigma) \Rightarrow^* \sigma'$$

Beweis: Strukturelle Induktion und Definition des Fixpunktoperators (Beweis ist lang).

$\square$

22.07.

5.5 Programmanalyse mit denotationeller Semantik

Ziel: Erhalte Informationen über das *dynamische* Verhalten eines Programms durch eine *statische* Analyse ohne das Programm auszuführen, z. B. mit `lint` für die Sprache C.

Beispiel. Verschiedene Anwendungen für statische Programmanalyse:

- definition-use-Analyse: Sind Variablen initialisiert bevor sie benutzt werden?
- constant-propagation: Werte/Ausdrücke, im Vorhinein auswerten, die nicht vom Zustand des Programms abhängen.
- Intervall-Analyse: Bestimme Intervalle für mögliche Variablenwerte.

Hier: Sonderfall der Intervall-Analyse: *Vorzeichenanalyse*, d. h. bestimme statisch mögliche Vorzeichen für die Variablen im Programm.

5.5.1 Vorzeichenanalyse

Idee: Arbeite mit Funktionen, die Eigenschaften von Zuständen abbilden und nicht die Zustände selbst (*property states*). Dazu benötigen wir die Menge abstrakter Eigenschaften von Programmvariablen $\mathcal{P}$.

Definition 5.13.

$$\mathcal{P} = \{\text{POS}, \text{NEG}, \text{NULL}, \text{BEL}\}$$

wobei BEL "beliebig" bedeutet.

Wir definieren außerdem eine partielle Ordnung $\sqsubseteq_{\mathcal{P}}$ auf $\mathcal{P}$ mit der Bedeutung "$p_1 \sqsubseteq_{\mathcal{P}} p_2$": p_1 ist mindestens so spezifisch wie p_2, z. B. POS $\sqsubseteq_{\mathcal{P}}$ BEL.

Wir nehmen an, dass $\sqsubseteq_{\mathcal{P}}$ so definiert ist, dass alle Teilmengen $Y \subseteq \mathcal{P}$ ein Supremum besitzen (*vollständiger Verband*). Dann gibt das Supremum die allgemeinste Eigenschaft einer Menge von Eigenschaften an.

In der Programmanalyse arbeiten wir mit *abstrakten Zustandseigenschaften* (property states), die jeder Variable eine Eigenschaft zuordnen statt eines konkreten Wertes:

$$\text{PState} : \text{Var} \to \mathcal{P}$$

Lemma 5.14. *Wenn $\mathcal{P}$ ein vollständiger Verband ist (siehe Definition 5.13), dann gilt das auch für* PState*.*

Genaue Formulierung nicht hier ...

Idee: Bisher hatten wir $\mathcal{S} : \text{Stm} \to (\text{State} \to \text{State})$. Nun betrachten wir die Analysefunktion $\mathcal{DS} : \text{Stm} \to (\text{PState} \to \text{PState})$.

Bemerkung (Halteproblem). Wir können nicht erwarten, dass die Programmanalyse immer die genauest mögliche Antwort liefert.

```
y := 1;
if bla then
    x := 1;
else
    S;
    x := -1;
y := x
```

Hier hängt die Vorzeichenanalyse für y davon ab, ob S terminiert oder nicht, d. h. unsere Analyse ist notwendigerweise approximativ und wird sagen, dass y negativ sein könnte.

5.5.2 Vorzeichenanalyse von `while`

(a) Wir definieren die gewünschte Eigenschaft, d. h. eine Variable ist entweder POS, NEG oder NULL.

Füge zusätzliche Eigenschaften hinzu, um die Analyse detaillierter zu machen und einen vollständigen Verband zu erhalten: NONPOS, NONNEG, NONNULL, BEL und NONE (letzteres aus technischen Gründen, um für die leere Menge ein Supremum zu haben).

Definiere Ordnung:

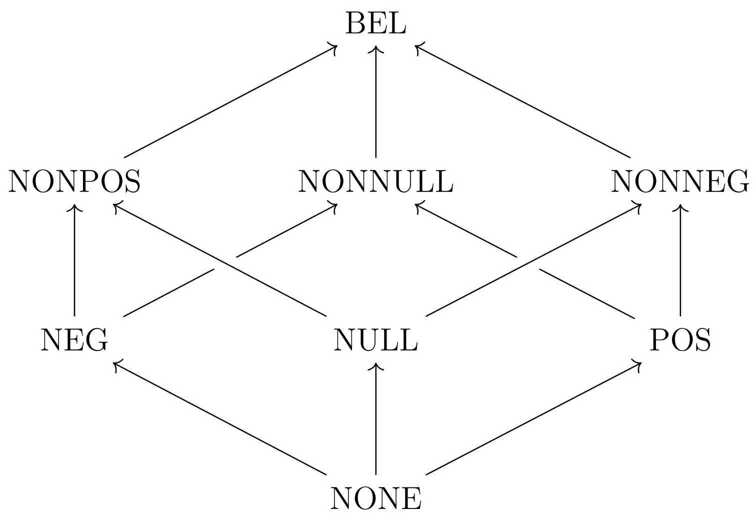


Abbildung 4: Hasse-Diagramm für $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$ (oben allgemein, unten spezifisch)

(b) Definiere property states: Definiere Verhältnis zwischen Zuständen und Zustandseigenschaften:

$$\mathrm{abs}_{\mathbb{Z}} : \mathbb{Z} \to \mathcal{P}$$
$$\mathrm{abs}_{\mathbb{Z}}(z) = \begin{cases} \text{POS} & \text{falls } z > 0 \\ \text{NULL} & \text{falls } z = 0 \\ \text{NEG} & \text{falls } z < 0 \end{cases}$$

Dann ist PState : Var $\to \mathcal{P}$. Wir definieren

$$\mathrm{abs} : \mathrm{State} \to \mathrm{PState}$$
$$\big(\mathrm{abs}(\sigma)\big)(x) = \mathrm{abs}_{\mathbb{Z}}(\sigma(x))$$

(c) Da die Wahrheitswerte im Programm vom Zustand abhängen, benötigen wir abstrakte Eigenschaften für die Wahrheitswerte, die aus den abstrakten Eigenschaften der Variablen folgen.

$$\mathcal{T} = \{\mathbf{w}, \mathbf{f}, \mathrm{BEL}, \mathrm{NONE}\}$$

Wieder brauchen wir eine Ordnung:

$$\begin{tikzcd}
 & \mathrm{BEL} & \\
\mathbf{w} \arrow[ur] & & \mathbf{f} \arrow[ul] \\
 & \mathrm{NONE} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Abbildung 5: Hasse-Diagramm für $\mathcal{T}$

(d) Definiere *Analysefunktion*: Analog zur semantischen Funktion, operiert aber auf Zustandseigenschaften anstelle von Zuständen: Für alle syntaktischen Kategorien definiere:

(a) Arithmetische Ausdrücke

$$\mathcal{DA} : \mathrm{AExp} \to (\mathrm{PState} \to \mathcal{P})$$
$$\mathcal{DA}[\![z]\!](\sigma) = \mathrm{abs}_{\mathbb{Z}}(\mathcal{N}(z))$$
$$\mathcal{DA}[\![x]\!](\sigma) = \sigma(z)$$
$$\mathcal{DA}[\![a_1 \mathbin{\square} a_2]\!](\sigma) = \mathcal{DA}[\![a_1]\!](\sigma) \mathbin{\square_v} \mathcal{DA}[\![a_2]\!](\sigma) \tag{*}$$

Hierbei (in (*)) simulieren $+_v$, $-_v$ und $\cdot_v$ das Verhalten des Vorzeichens `+`, `-` und `*`:

$$\begin{aligned} \square_v &: \mathcal{P} \times \mathcal{P} \to \mathcal{P} \\ \text{z. B. POS} +_v \text{POS} &= \text{POS} \\ \text{POS} +_v \text{NEG} &= \text{BEL} \\ &\vdots \end{aligned}$$

(b) Boolesche Ausdrücke

$$\begin{aligned} \mathcal{DB} &: \text{BExp} \to (\text{PState} \to \mathcal{T}) \\ \mathcal{DB}[\![\texttt{true}]\!](\sigma) &= \mathbf{w} \\ \mathcal{DB}[\![\texttt{false}]\!](\sigma) &= \mathbf{f} \\ \mathcal{DB}[\![a_1 \square a_2]\!](\sigma) &= \mathcal{DA}[\![a_1]\!](\sigma) \square_v \mathcal{DA}[\![a_2]\!](\sigma) \\ \mathcal{DB}[\![\neg b]\!](\sigma) &= \neg_{\mathcal{T}} \mathcal{DB}[\![b]\!](\sigma) \\ \mathcal{DB}[\![b_1 \square b_2]\!](\sigma) &= \mathcal{DB}[\![b_1]\!](\sigma) \square_{\mathcal{T}} \mathcal{DB}[\![b_2]\!](\sigma) \end{aligned}$$

(c) Anweisungen

$$\begin{aligned} \mathcal{DS} &: \text{Stm} \to (\text{PState} \to \mathcal{P}) \\ \mathcal{DS}[\![x \texttt{ := } a]\!](\sigma) &= \sigma[x \mapsto \mathcal{DA}[\![a]\!](\sigma)] \\ \mathcal{DS}[\![\texttt{skip}]\!](\sigma) &= \sigma \\ \mathcal{DS}[\![S_1\texttt{;}\ S_2]\!](\sigma) &= \mathcal{DS}[\![S_2]\!](\sigma) \circ \mathcal{DS}[\![S_1]\!](\sigma) \\ \mathcal{DS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!](\sigma) &= \text{cond}_\infty(\mathcal{DB}[\![b]\!], \mathcal{DS}[\![S_1]\!], \mathcal{DS}[\![S_2]\!]) \\ \mathcal{DS}[\![\texttt{while } b \texttt{ do } S]\!](\sigma) &= \text{FIX}(H) \\ H : h &\mapsto \text{cond}_\infty(\mathcal{DB}(b), h \circ \mathcal{DS}(S), \text{id}) \end{aligned}$$

mit

$$\text{cond}_\infty(f, h_1, h_2)(\sigma) = \begin{cases} \text{INIT} & \text{falls } f(\sigma) = \text{NONE} \\ h_1(\sigma) & \text{falls } f(\sigma) = \mathbf{w} \\ h_2(\sigma) & \text{falls } f(\sigma) = \mathbf{f} \\ \sup\{h_1(\sigma), h_2(\sigma)\} & \text{falls } f(\sigma) = \text{BEL} \end{cases}$$

mit

$$\begin{aligned} \text{INIT} &: \text{Var} \to \mathcal{P} \\ \forall x \in \text{Var} &: \text{INIT}(x) = \text{NONE} \end{aligned}$$