

Semantik von Programmiersprachen

Vorlesung SoSe 2022

Wolfgang Mulzer

Jim Neuendorf

7. 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

1	Einführung	3
1.1	Programmiersprachen	3
1.1.1	Syntax	4
1.1.2	Semantik	4
1.1.3	Idiomatik	4
2	Semantik	5
3	Mathematische Formalisierung	7
3.1	while-Sprache	7
3.2	Axiomatische Semantik	8
3.3	Operationelle Semantik	8
3.4	Denotationelle Semantik	9
4	Operationelle Semantik	10
4.1	Semantik arithmetischer Ausdrücke	11
4.2	Freie Variablen	13
4.3	Substitution	15
4.4	Natürliche operationelle Semantik (“big step”)	17
4.4.1	Schlussregeln für die natürliche Semantik von while	17
4.5	Strukturelle operationelle Semantik (“small step”)	22
4.6	Eigenschaften der SOS	24
4.7	Semantische Funktion \mathcal{S}_{sos}	25
4.8	Erweiterungen der while -Sprache	27
4.8.1	Programmabbruch	27
4.8.2	Blöcke und lokale Variablen	28
4.8.3	Formalisierung lokaler Blöcke in der NS	29
4.8.4	Schlussregeln für D_V , \rightarrow_D und Blöcke	29
4.8.5	Unterprogramme / Prozeduren / Subroutine / Methode	30
4.8.6	Natürlichen operationelle Semantik für Unterprogramme mit dynamische Gültigkeitsbereichen	31
4.8.7	Schlussregeln für Unterprogramme mit dynamischen Gültigkeitsbereichen	31
5	Denotationelle Semantik	33
5.1	Direkte Definition der semantischen Funktion	33
5.1.1	Problemtransfer zu Fixpunkt eines Funktionals	34
5.2	Der Fixpunktoperator	36
5.2.1	Wie muss ein Fixpunkt aussehen?	36
5.2.2	Fixpunktiteration	38

1 Einführung

Diese Vorlesung orientiert sich an folgender Literatur:

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

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 Konstrukten 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 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.

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?

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

Listing 1: Imperatives Programm

```
1 foo :: Int -> Int
2 foo 0 = 1
3 foo x = x + 1
4 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 $V = \{x_1, x_2, \dots, y, z, \dots\}$ (Speicher ist begrenzt aber beliebig groß). Der Zustand ist eine (partielle) Funktion

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

(\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 Werte kann eine andere Definition sinnvoller sein.

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

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

$$S[\Pi] \in [State \rightarrow State]$$

(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:

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

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

Beispiel.

$$\begin{aligned} \Pi &= \mathbf{x} = \mathbf{z} + 1 \\ S[\mathbf{x} = \mathbf{z} + 1] &(\underbrace{([x \mapsto 5, z \mapsto -4, a \mapsto 2])}_{\text{Startzustand}}) = \underbrace{([x \mapsto -3, z \mapsto 4, a \mapsto 2])}_{\text{Endzustand}} \\ S[\mathbf{x} = \mathbf{z} + 3] &([x \mapsto 10, z \mapsto 12]) = [x \mapsto 10, z \mapsto 12] \end{aligned}$$

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.

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

3.3 Operationelle Semantik

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

Genauer gesagt bedeutet das: Wir definieren ein *Transitionssystem*

$$\begin{aligned} \langle \Pi, s \rangle &\Rightarrow \langle \Pi', s' \rangle \\ \langle \Pi, s \rangle &\Rightarrow s' \end{aligned}$$

das die Ausführung von Π auf Zustand s darstellt.

Beispiel.

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

3.4 Denotationelle Semantik

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

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

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. AExp, BExp, 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.

$$a \in \text{AExp}$$

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 Zeichenkette, die ganze Zahlen darstellen.

$$1234 \in \text{Num}$$

$$1234 \in \mathbb{Z}$$

Beachte, dass eigentlich der Syntaxbaum von 1234 gemeint ist

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

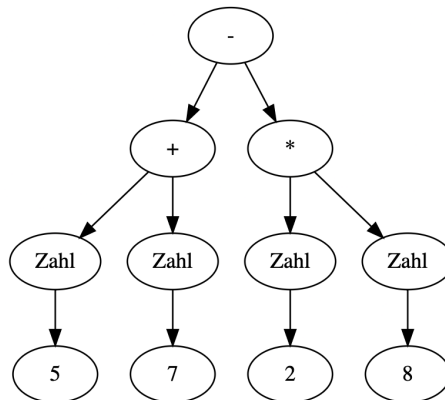


Abbildung 1: $5+7-2*8 \in \text{AExp}$ als verkürzt dargestellter Syntaxbaum

Bemerkung. Unterbäume können auch Elemente einer anderen Kategorie sein.

Beispiel. Division mit Rest

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

```
1 m := 0;  
2 while b <= a do (  
3     m := m + 1;  
4     a := a - b  
5 )  
6 r := a
```

Listing 3: Division mit Rest

4 OPERATIONELLE SEMANTIK

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

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

$$\begin{aligned}\mathcal{N} : \text{Num} &\rightarrow \mathbb{Z} \\ \mathcal{N}[-123] &= -123\end{aligned}$$

$$\begin{aligned}\mathcal{A} : \underbrace{\text{AExp}}_{\text{“Compiler”}} &\rightarrow \underbrace{(\text{State} \rightarrow \mathbb{Z})}_{\text{“Interpreter”}} \\ \mathcal{A}[\mathbf{x}+\mathbf{x}*5] &= (\sigma \mapsto 6 \cdot \sigma(x)) \quad (\text{da } \mathbf{x}+\mathbf{x}*5 = 6*\mathbf{x})\end{aligned}$$

$$\begin{aligned}\mathcal{B} : \text{BExp} &\rightarrow (\text{State} \rightarrow \{w, f\}) \\ \mathcal{B}[\mathbf{x} \leq 10] &= \left(\sigma \mapsto \begin{cases} w & \sigma(x) \leq 10 \\ f & \sigma(x) > 10 \end{cases} \right)\end{aligned}$$

$$S : \text{Stm} \rightarrow (\text{State} + \text{State})$$

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 $n \in \text{Num}$, $x \in \text{Var}$, $\sigma \in \text{State}$ und $a_1, a_2 \in \text{AExp}$. Wir definieren:

- (i) $\mathcal{A}[\mathbf{n}](\sigma) = \mathcal{N}[\mathbf{n}]$ (konstante Funktion)
- (ii) $\mathcal{A}[\mathbf{x}](\sigma) = \sigma(x)$ alternativ: $\mathcal{A}[\mathbf{n}] = (\sigma \mapsto \sigma(n))$
- (iii) $\mathcal{A}[\mathbf{a}_1 + \mathbf{a}_2](\sigma) = \mathcal{A}[\mathbf{a}_1](\sigma) + \mathcal{A}[\mathbf{a}_2](\sigma)$

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: $\mathcal{A} \rightarrow \dots \mid - A$.

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 $\mathcal{A} \llbracket a \rrbracket(\sigma) = n$. (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) $\text{FV}(n) = \emptyset$ (falls $a = n$ (Zahl))
- (b) $\text{FV}(x) = \{x\}$ (falls $a = x$ (Variable))
- (c) $\text{FV}(a_1 \square a_2) = \text{FV}(a_1) \cup \text{FV}(a_2)$ für $\square \in \{+, -, *\}$

Gebundene Variable 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:*

$$\mathcal{A} \llbracket a \rrbracket(\sigma) = \mathcal{A} \llbracket a \rrbracket(\sigma')$$

Beweis. Durch strukturelle Induktion nach a .*Induktionsanfang:*

(a) $a = n$, n Zahl

$$\begin{aligned}\mathcal{A} \llbracket a \rrbracket(\sigma) &= \mathcal{A} \llbracket n \rrbracket(\sigma) \\ &\stackrel{\text{D. 4.2(i)}}{=} \mathcal{N} \llbracket n \rrbracket\end{aligned}$$

$$\begin{aligned}\mathcal{A} \llbracket a \rrbracket(\sigma') &= \mathcal{A} \llbracket n \rrbracket(\sigma') \\ &= \mathcal{N} \llbracket n \rrbracket\end{aligned}$$

(b) $a = x$, x Variable

$$\begin{aligned}\mathcal{A} \llbracket a \rrbracket(\sigma) &= \mathcal{A} \llbracket x \rrbracket(\sigma) \\ &\stackrel{\text{D. 4.2(ii)}}{=} \sigma(x)\end{aligned}$$

$$\begin{aligned}\mathcal{A} \llbracket a \rrbracket(\sigma') &= \mathcal{A} \llbracket x \rrbracket(\sigma') \\ &= \sigma'(x)\end{aligned}$$

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

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

$$\mathcal{A} \llbracket a \rrbracket(\sigma) = \mathcal{A} \llbracket a \rrbracket(\sigma')$$

Induktionsschritt:

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

$$\begin{aligned}\mathcal{A} \llbracket a \rrbracket(\sigma) &= \mathcal{A} \llbracket a_1 \square a_2 \rrbracket(\sigma) \\ &= \mathcal{A} \llbracket a_1 \rrbracket(\sigma) \square \mathcal{A} \llbracket a_2 \rrbracket(\sigma)\end{aligned}$$

$$\begin{aligned}\mathcal{A} \llbracket a \rrbracket(\sigma') &= \mathcal{A} \llbracket a_1 \square a_2 \rrbracket(\sigma') \\ &= \mathcal{A} \llbracket a_1 \rrbracket(\sigma') \square \mathcal{A} \llbracket a_2 \rrbracket(\sigma')\end{aligned}$$

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

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

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

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

und

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

Also gelten nach Induktionsvoraussetzung

$$\mathcal{A} \llbracket a_1 \rrbracket(\sigma) = \mathcal{A} \llbracket a_1 \rrbracket(\sigma') \wedge \mathcal{A} \llbracket a_2 \rrbracket(\sigma) = \mathcal{A} \llbracket a_2 \rrbracket(\sigma')$$

Daher folgt:

$$\mathcal{A} \llbracket a_1 \rrbracket(\sigma) \sqcap \mathcal{A} \llbracket a_2 \rrbracket(\sigma) = \mathcal{A} \llbracket a_1 \rrbracket(\sigma') \sqcap \mathcal{A} \llbracket a_2 \rrbracket(\sigma')$$

□

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:

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

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

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

Beispiel.

$$\begin{aligned} a &= x + y * 2 - z(x + y) \\ y &\mapsto 4 * t + 5 \end{aligned}$$

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

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:

$$\mathcal{A} \llbracket a[y \mapsto a_0] \rrbracket(\sigma) = \mathcal{A} \llbracket a \rrbracket(\sigma[y \mapsto \mathcal{A} \llbracket a_0 \rrbracket(\sigma)])$$

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

Beispiel.

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

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

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

Induktionsanfang:

(a) $a = n$, n Zahl

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

$$\mathcal{A} \llbracket n \rrbracket(\sigma[y \mapsto \mathcal{A} \llbracket a_0 \rrbracket(\sigma)]) = n \quad (\text{rechte Seite})$$

(b) $a = x$, x Variable

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

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

Induktionsschritt: Straight forward. □

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 Zustandsüberföhrungsrelation.

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

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

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 $[\cdot]_{\text{ns}}$ um anzuzeigen, dass es sich um die natürliche Semantik handelt.

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

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

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

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

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

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

Dabei können S_1 und S_2 zusammengesetzte Anweisungen sein.

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

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

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

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

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

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

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

$$\frac{\langle S, \sigma \rangle \rightarrow \sigma', \langle \text{while } b \text{ do } S, \sigma' \rangle \rightarrow \sigma''}{\langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \sigma''}$$

falls $\mathcal{B}[[b]](\sigma) = \text{w}$. 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 $[\text{while}_{\text{ns}}^{\text{f}}]$

$$\overline{\langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \sigma}$$

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

Wie ist die Zustandsüberföhrungsfunktion überhaupt definiert?

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

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

gdw. ein *endlicher Ableitungsbaum* dafür existiert.

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

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

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

Nun müssen wir den endlichen Ableitungsbaum erzeugen.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Beispiel. Die Programme

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

und

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

sind semantisch äquivalent.

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

(a) “ \Rightarrow ”

Nimm an, es gilt $\langle S_1, \sigma \rangle \rightarrow \sigma'$. Also existiert nach Definition ein endlicher Ableitungsbaum für $\langle S_1, \sigma \rangle \rightarrow \sigma'$.

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

Dann hat der Ableitungsbaum T_1 die Form

$$[\text{while}_{\text{ns}}^w] \frac{\frac{T_a}{\langle S, \sigma \rangle \rightarrow \sigma''} \quad \frac{T_b}{\langle \text{while } b \text{ do } S, \sigma'' \rangle \rightarrow \sigma'}}{\langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \sigma'}$$

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

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

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

Dann hat T_1 die Form

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

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

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

$$[\text{if}_{\text{ns}}^f] \frac{[\text{skip}_{\text{ns}}] \overline{\langle \text{skip}, \sigma \rangle \rightarrow \sigma}}{\langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, \sigma \rangle \rightarrow \sigma} \quad \mathcal{B}[\![b]\!](\sigma) = f$$

(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 $\langle S, \sigma \rangle \rightarrow \sigma_1$ und $\langle S, \sigma \rangle \rightarrow \sigma_2$, dann $\sigma_1 = \sigma_2$.

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

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

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

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 “ \Rightarrow ”. Sie hat die Form

$$\langle S, \sigma \rangle \Rightarrow \langle S', \sigma' \rangle \quad (*)$$

oder

$$\langle S, \sigma \rangle \Rightarrow \sigma' \quad (**)$$

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

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

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

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

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

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

$$\frac{\langle S_1, \sigma \rangle \Rightarrow \langle S'_1, \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \Rightarrow \langle S'_1; S_2, \sigma' \rangle}$$

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

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

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

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

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

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

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

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

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

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

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

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

- (a) eine endliche Folge $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_k$ von Konfigurationen, sodass $\gamma_0 = \langle S, \sigma \rangle$ ist, $\gamma_i \Rightarrow \gamma_{i+1}$ für $0, \dots, k-1$ 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 $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots$ von Konfigurationen mit $\gamma_0 = \langle S, \sigma \rangle$ mit $\gamma_i \Rightarrow \gamma_{i+1}$ für $i \geq 0$.

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 .

$$\begin{array}{c} \langle (z := x; x := y); y = z; , \sigma \rangle \\ \xRightarrow{[\text{seq}_{\text{sos}}^1]} \langle x := y; y := z, \sigma[z \mapsto 5] \rangle \end{array} \quad (\text{i})$$

$$\xRightarrow{[\text{seq}_{\text{sos}}^2]} \langle y := z, \sigma[z \mapsto 5][y \mapsto 7] \rangle \quad (\text{ii})$$

$$\xRightarrow{[\text{zuw}_{\text{sos}}]} \sigma[z \mapsto 5][x \mapsto 7][y \mapsto 5]$$

zu (i):

$$\begin{array}{c} [\text{seq}_{\text{sos}}^1] \frac{[\text{seq}_{\text{sos}}^2] \frac{[\text{zuw}_{\text{sos}}] \overline{\langle z := x, \sigma \rangle \Rightarrow \sigma[z \mapsto 5]}}{\langle z := x; x := y, \sigma \rangle \Rightarrow \langle x := y, \sigma[z \mapsto 5] \rangle}}{\langle (z := x; x := y); y := z, \sigma \rangle \Rightarrow \langle x := y; y := z, \sigma[z \mapsto 5] \rangle}} \end{array}$$

zu (ii):

$$\begin{array}{c} [\text{seq}_{\text{sos}}^2] \frac{[\text{zuw}_{\text{sos}}] \overline{\langle x := y, \sigma[z \mapsto 5] \rangle \Rightarrow \langle y := z, \sigma[z \mapsto 5][x \mapsto 7] \rangle}}{\langle x := y; y := z, \sigma[z \mapsto 5] \rangle \Rightarrow \langle y := z, \sigma[z \mapsto 5][x \mapsto 7] \rangle}} \end{array}$$

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 $[\text{seq}_{\text{sos}}^2]$ angewendet 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 $k \geq 3$

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

(a) $[\text{seq}_{\text{sos}}^1]$ 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 $[\text{seq}_{\text{sos}}^1]$ 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 $\langle S'_1, \sigma \rangle \Rightarrow^{k'_1} \sigma'$ also $\langle S_1, \sigma \rangle \Rightarrow^{k_1+1} \sigma'$.

Also gilt die Aussage für $k_1 + 1, k_2 = k'_2, \sigma'$.

(b) $[\text{seq}_{\text{sos}}^2]$ 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 gibt es 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 *semantische äquivalent* gdw. folgendes für allen 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 unendliche 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 semantische äquivalent.

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

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

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

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

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

Wir wissen $\langle S, \sigma \rangle \rightarrow \sigma'$, d. h. es existiert ein endliche Ableitungsbaum T dafür. Mache Induktion nach der Tiefe von T .

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

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

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

$$\overline{\langle \text{while } b \text{ do } S', \sigma \rangle \rightarrow \underbrace{\sigma'}_{\sigma}}$$

Jetzt gilt

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

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

Induktionsschritt:

- (a) $[\text{while}_{\text{ns}}^w]$
- (b) $[\text{if}_{\text{ns}}^*]$
- (c) $[\text{seq}_{\text{ns}}]$

Wir machen exemplarisch (a).

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

$$\frac{\frac{T_1}{\langle S', \sigma \rangle \rightarrow \sigma''} \quad \frac{T_2}{\langle \text{while } b \text{ do } S', \sigma'' \rangle \rightarrow \sigma'}}{\langle \text{while } b \text{ do } S', \sigma \rangle \rightarrow \sigma'}$$

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

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

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

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

gilt.

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

(b) “ \Leftarrow ”: Übung

□

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.

$$S \rightarrow \dots \mid \text{abort}$$

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:

$S = \text{if } \neg(x \geq 1) \text{ then abort else } (y := 1; \text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1))$

Wie sieht es bei der NS aus?

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

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

Dabei existiert T nicht! Daraus folgt

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

(für unseren konkreten Zustand σ).

S ist **semantisch äquivalent** zu

$S' = \text{if } \neg(x \geq 1) \text{ then } (\text{while true do skip}) \text{ else } (y := 1; \text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1))$

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

Wie sieht es bei der SOS aus?

$$\langle S, \sigma \rangle \xRightarrow{[\text{if}_{\text{sos}}^w]} \langle \text{abort}, \sigma \rangle$$

Die rechte Konfiguration ist eine steckengebliebene Konfiguration.

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

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.

$$\begin{aligned} S &\rightarrow \dots \mid \text{begin } D_V \ S \ \text{end} \\ D_V &\rightarrow \text{var Var} := A; \ D_V \mid \varepsilon \end{aligned}$$

D_V entspricht der syntaktischen Kategorie Dec_V .

Beispiel. Betrachte

```

1 begin
2   var y := 1;
3   x := 1;
4   begin
5     var x := 2;
6     y := x + 1  (* (1) 2 + 1 *)
7   end
8   y := y + x  (* (2) 3 + 1 *)
9 end

```

In (1) und (2) überdeckt das lokale x das globale x in diesem Block.

4.8.3 Formalisierung lokaler Blöcke in der NS

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

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

Wir führen eine neue Relation “ \rightarrow_D ” ein. Sie hat die Form

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

wobei σ' ein lokaler Zustand ist.

4.8.4 Schlussregeln für D_V , \rightarrow_D und Blöcke

$$\overline{\langle \varepsilon, \sigma \rangle \rightarrow_D \sigma}$$

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

$$\frac{\langle D_V, \sigma \rangle \rightarrow_D \sigma'', \langle S, \sigma'' \rangle \rightarrow \sigma'}{\langle \text{begin } D_V \ S \ \text{end}, \sigma \rangle \rightarrow \sigma'[DV(D_V) \mapsto \sigma]}$$

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 lokale deklarierten Variablen werden auf ihren ursprünglichen Wert zurückgesetzt.

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:

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

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

```
1 begin
2   var x := 0;
3   proc p is x := x * 2;
4   proc q is call p;
5
6   begin
7     var x := 5;
8     proc p is x := x + 1;
9     call q;
10    y := x
11  end
12 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 dynamische Gültigkeitsbereichen

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

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

formalisiert. Die Übergangsrelation hat nun die Form

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

mit $\text{env}_p \in \text{Env}_p$ (also eine Funktion), $S \in \text{Stm}$ und $\sigma, \sigma' \in \text{State}$. 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 dynamischen Gültigkeitsbereichen

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

Beispiel. $[\text{seq}_{\text{ns}}]$

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

Wichtig / interessant sind nun $[\text{block}_{\text{ns}}]$ und $[\text{call}_{\text{ns}}]$ (neu):

(a) $[\text{block}_{\text{ns}}]$

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

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

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

(b) $[\text{call}_{\text{ns}}]$

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

wobei $\text{env}_p(p) = S$, 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.

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.

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

sind denotationell definiert.

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

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

5.1 Direkte Definition der semantischen Funktion

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

Die grundlegenden Definitionen bleiben gleich:

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

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

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

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

Definition 5.2 (*cond*).

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

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

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

Beobachtung: $\text{while } b \text{ do } S$ und $\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}$ sollen semantisch äquivalent sein.

Also eigentlich wollen wir schreiben:

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

Das verletzt aber die Zusammengesetztheit.

Wenn wir diese Gleichung betrachten, erkennen wir

Bemerkung (Erkenntnis). $\mathcal{S}_{\text{ds}}[\text{while } b \text{ do } S]$ 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 Funktional

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

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

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

durch

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

Ein *Fixpunkt* von F ist ein $f^* : \text{State} \rightarrow \text{State}$ 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*

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

der einem Funktional F einen Fixpunkt f^* zuordnet.

Dann können wir schreiben

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

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 Funktional.

Frage. Was können wir über $\text{FIX}(f)$ sagen?

- (a) Existiert für *jedes* Funktional $G : (\text{State} \rightarrow \text{State}) \rightarrow (\text{State} \rightarrow \text{State})$ 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 $f = F(f)$?

Fixpunkt-Iteration: 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).

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

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 Zustandsüberföhrungsfunktion 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 Zustandsüberföhrungsfunktion, gelb), eine Funktion (Kandidat für Fixpunkt, blau) und das f -Funktional $F(f)$ (grün).

F erzeugt also eine neue Zustandsüberföhrungsfunktion, 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 Findes eines Fixpunktes bedeutet also, ein f (blau) zu finden, sodass nach Anwendung von $F(f)$ (grün) die gleichen Pfeile entstehen wie in f .

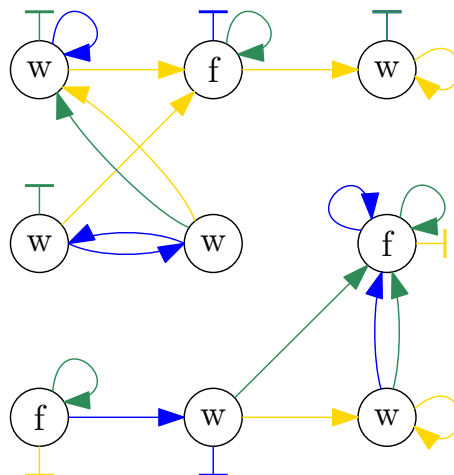
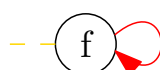


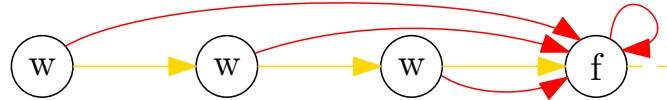
Abbildung 2: Visualisierung der Funktionsverkettung durch F

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.

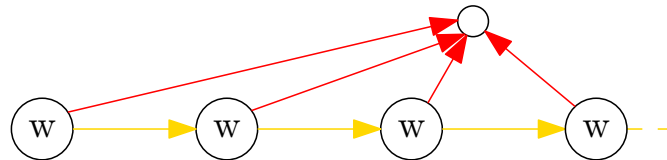


- (c) Es gibt Zustände mit Wahrheitswert w , die nicht nach endliche vielen Schritten einen Zustand mit Wahrheitswert f erreichen.

- (a) $w \rightarrow w \rightarrow w \rightarrow \dots$

```

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

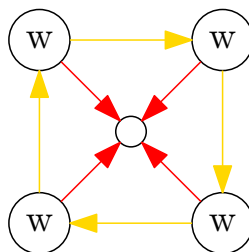


Hier müssen alle Pfeile zum selben Zustand zeigen, egal welcher genau das ist.

- (b) $w_1 \rightarrow w_2 \rightarrow w_2 \rightarrow w_1$

```

1 x := 0;
2 while x != -1 do
3   x := (x + 1) mod 10
  
```

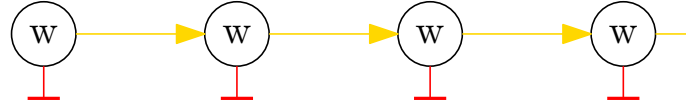


Hier müssen ebenfalls alle Pfeile zum selben Zustand zeigen, egal welcher genau das ist.

- (c) $w \rightarrow w \rightarrow w \rightarrow \perp$

```

1 x := 1;
2 while x < 20 do
3   x := x + 1;
4   if x = 10 then
5     while true do skip
6   else
7     ...
  
```



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 die richtigen Fixpunkt durch Fixpunktiteration. Starte mit der Zustandsüberföhrungsfunktion

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

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).

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

Aber wie ist der Grenzwert in diesem Kontext definiert?

Definition 5.6. Sei $\mathcal{F} = \{f \mid f : \text{State} \rightarrow \text{State}\}$ die Menge aller partiellen Zustandsüberföhrungsfunktionen. 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. $g_1, g_2, g_3, g_4 : \text{State} \rightarrow \text{State}$

$$\begin{aligned} g_1(\sigma) &= \sigma \quad \forall \sigma \in \text{State} \\ g_2(\sigma) &= \begin{cases} \sigma & \text{falls } \sigma(x) \geq 0 \\ \perp & \text{sonst} \end{cases} \\ g_3(\sigma) &= \begin{cases} \sigma & \text{falls } \sigma(x) \leq 0 \\ \perp & \text{sonst} \end{cases} \\ g_4(\sigma) &= \begin{cases} \sigma & \text{falls } \sigma(x) = 0 \\ \perp & \text{sonst} \end{cases} \end{aligned}$$

Behauptung:

$$\begin{aligned} g_4 &\sqsubseteq g_2 \sqsubseteq g_1 \\ g_4 &\sqsubseteq g_3 \sqsubseteq g_1 \\ g_4 &\sqsubseteq g_1 \end{aligned} \tag{*}$$

(*) 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

$$f \sqsubseteq g$$

Bemerkung (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_\perp .