Verifikation

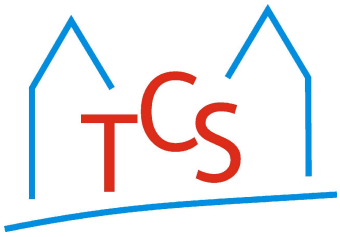
Unterlagen zum Lehrmodul CS4030

Prof. Dr. R. Reischuk

Institut für Theoretische Informatik

Universität zu Lübeck

WS 2009/10



Version 9.3.2010

Inhaltsverzeichnis

1 Einleitung

1.1 Überblick

Unter **Verifikation** versteht man die *Untersuchung eines Produktes oder Service daraufhin, ob die Anforderungen erfüllt werden.*
Dies kann geschehen durch Inspektion, Testen, Beweisen, ... Testen beispielsweise kann Fehler aufdecken, aber im allgemeinen nicht die Korrektheit nachweisen, es sei denn die Testmenge deckt alle Situationen ab, was auf Grund der vielen Möglichkeiten (in der Regel mindestens exponentielles Wachstum) in der Praxis kaum durchführbar ist.

Anwendungsszenarien:

- allgemeine Anwendungen: Funktionalität einer Signalsteuerung, eines Auto-Piloten, eines Bankautomaten oder die Sicherheit eines kryptografisches Protokolls;
- in der SW-Entwicklung: formale Verifikation der beiden Aspekte partielle Korrektheit und Terminierung (i.a. nicht entscheidbar), Garantien für das Laufzeit-Verhalten von Algorithmen oder Systemen;
- in der HW-Entwicklung: beim Schaltkreis-Design beispielsweise ein mehrstufiges Verfahren: Nachweis der funktionalen Korrektheit, der analogen Korrektheit (Störungen), Layout (Signallaufzeiten).

Spätestens seit dem Pentium-Bug ist formale Verifikation Standard in der HW-Entwicklung; in der SW-Entwicklung wird die Notwendigkeit immer mehr eingesehen.

Grundlage für eine Verifikation ist die formale Spezifikation in einem geeigneten Kalkül, beispielsweise endliche Automaten, Transitionssysteme, Petrinetze, hybride Automaten
Verwendung geeigneter Semantiken (operational, denotationell, axiomatisch) oder Logiken (Hoare-Logik, LTL, CTL)
Beim Beweis werden formale Methoden oder Techniken eingesetzt, insbesondere **Model Checking** (Exploration des Zustandsraumes) und **Inferenz** (formale Beweisführung).

Im Unterschied zur Verifikation wird bei der **Validierung** untersucht, ob der Anforderungskatalog geeignet ist für die Anwendung (in der Regel dynamisch):
Validierung: are we constructing the right product? versus
Verifikation: are we constructing the product right?

Semantiken gliedern sich in

- **operational:** ein Programm wird als eine Folge von Berechnungsschritten interpretiert mit Programmzuständen (Beispiel ALGOL60, PL1), eine Programmabstraktion ist dann Grundlage für Korrektheitsbeweise;
- **denotationell:** Konstruktion von mathematischen Objekten (Denotationen), die die Bedeutung von Programmausdrücken beschreiben; dies entspricht einem funk-

tionalen Ansatz, die die Transformation der Inputs in Outputs modelliert; Grundlagen hierzu wurden von Christopher Strachey, Dana Scott 1960 gelegt, wichtig dabei sind Komposition, partielle Ordnungen und Fixpunkte für Rekursion;

- **axiomatisch:** ein noch weitere Abstraktion durch logische Aussagen über Programmzustände, Variablen, ... (Hoare-Kalkül, Dijkstras wp-Kalkül)

1.2 Empfohlene Literatur

H. Nielson, F. Nielson, Semantics With Applications - A Formal Introduction, J. Wiley, 1992

J. Monin, Understanding Formal Methods, Springer, 2003

C. Baier, P. Katoen, Principles of Model Checking, MIT Press, 2008

J. van Leeuwen, Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics, Elsevier, 1990

E. Best, Semantik, Theorie seqentieller und paralleler Programmierung, Vieweg, 1995

F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge UP, 1998

E. Olderog, B. Steffen, Correct System Design, Springer LNCS 1710, 1998

E. Börger (Ed.), Architecture Design and Validation Methods, Springer 2000

M. Felleisen, R. Findler, M. Flatt, S. Krishnamurthi, How to Design Programs, MIT Press, 2001

G. Winskel, The Formal Semantics of Programming Languages, MIT Press, 1993

C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, Prentice Hall, 2002

1.3 Einführendes Beispiel

Oftmals weicht das Verhalten von Programmen von den Erwartungen des Programmierers ab. Durch Testeingaben können Fehler gefunden werden, deren Abwesenheit kann auf diese Weise aber nur schwer garantiert werden. Betrachten wir folgendes Beispiel einer Funktion $f : \mathbb{Z} \to \mathbb{Z}$, für die wir entscheiden wollen, ob sie eine Nullstelle besitzt. Diese Aufgabe soll durch folgendes verteiltes Programm bestehend aus den Prozessen Q_1 und Q_2 mit Hilfe der Variable `found` gelöst werden.

```
process Q1 :
found := false;
x := 0;
while ¬found do
  x:=x+1;
  found:=[f(x)=0]
od
```

```
process Q2 :
found := false;
y := 1;
while ¬found do
  y:=y-1;
  found:=[f(y)=0)]
od
```

Ist das Programm $Q = Q_1||Q_2$ korrekt? Die Korrektheit bei parallel ausgeführten Programmen nachzuweisen, ist noch einmal deutlich schwieriger als bei sequentiellen, da die Synchronisationsproblematik den Raum der möglichen Programmzustände massiv vergrößert und weitere Probleme wie Deadlocks oder Fairness hinzukommen.

1.4 Unterschiedliche Semantiken

Eine **operationale** Erklärung der Bedeutung eines Programmkonstruktes erläutert, wie dieses auszuführen ist. Betrachten wir ein einfaches Beispiel, welches nur aus der Komposition von Zuweisungen besteht.

- Eine Folge von Anweisungen getrennt durch das Zeichen ";" wird der Reihe nach von links nach rechts ausgeführt.
- Eine Zuweisung $x := y$ wird ausgeführt, indem der Variablen auf der linken Seite der Wert der Variablen auf der rechten Seite zugewiesen wird.

$$
\begin{aligned}
< z := x;\ x := y;\ y := z, \quad & [x \mapsto 5,\ y \mapsto 7, z \mapsto 0] > \quad \Longrightarrow \\
< x := y;\ y := z, \quad & [x \mapsto 5,\ y \mapsto 7, z \mapsto 5] > \quad \Longrightarrow \\
< y := z, \quad & [x \mapsto 7,\ y \mapsto 7, z \mapsto 5] > \quad \Longrightarrow \\
< , \quad & [x \mapsto 7,\ y \mapsto 5, z \mapsto 5] >
\end{aligned}
$$

Die Herleitung ist bereits eine Abstraktion, da wir Details wie Register und Adressen für Variable nicht berücksichtigen.

Bei einer **denotationellen Semantik** werden die Auswirkungen einer Programmausführung beschrieben, und zwar mit Hilfe mathematischer Funktionen.

- Eine Folge von Anweisungen entspricht der Komposition der entsprechenen Funktionen.
- Eine Zuweisung $x := y$ erzeugt einen neuen Zustand, der sich vom alten dadurch unterscheidet, daß die x-Variable nun den Wert der y-Variable besitzt. Dies beschreiben wir durch die Funktion $S[x := y]$.

Somit ergibt sich für das obige Programm

$$S[z := x;\ x := y;\ y := z] \ = \ S[y := z] \circ S[x := y] \circ S[z := x] \ ,$$

wobei man beachte, daß die Komposition von Funktionen üblicherweise von rechts nach links notiert wird. Wendet man nun diese Funktion auf den Anfangszustand an, so ergibt sich

$$\begin{aligned}
& S[z := x;\ x := y;\ y := z]\ ([x \mapsto 5,\ y \mapsto 7, z \mapsto 0]) \\
&= \ (S[y := z] \circ S[x := y] \circ S[z := x])\ ([x \mapsto 5,\ y \mapsto 7, z \mapsto 0]) \\
&= \ (S[y := z]\ (S[x := y]\ (S[z := x]\ ([x \mapsto 5,\ y \mapsto 7, z \mapsto 0])))) \\
&= \ (S[y := z]\ (S[x := y]\ ([x \mapsto 5,\ y \mapsto 7, z \mapsto 5])) \\
&= \ (S[y := z]\ ([x \mapsto 7,\ y \mapsto 7, z \mapsto 5]) \\
&= \ [x \mapsto 7,\ y \mapsto 5, z \mapsto 5]
\end{aligned}$$

Die Abstraktionsebene ist in diesem Fall höher, da wir uns nicht darum kümmern, wie Programmanweisungen ausgeführt werden. Eine derartige Semantik kann man erweitern, um zusätzliche Eigenschaften zu überprüfen, beispielsweise ob

- alle Variablen initialisiert sind, bevor sie benutzt werden,
- ob bestimmte Ausdrücke vereinfacht werden können, zum Beispiel einen konstanten Wert haben – dann könnte man diese ersetzen,
- ob alle Teile eines Programmes tatsächlich erreicht werden können.

Möchte man nur bestimmte Eigenschaften eines Programmes überprüfen, so spricht man von partieller Korrektheit. Dies wird durch eine Vorbedingung Φ und eine Nachbedingung Ψ ausgedrückt.

Definition 1.1 Ein Programm Π heißt **partiell korrekt bezüglich (Φ, Ψ)**, wenn gilt: Gilt die Vorbedingung Φ im Anfangszustand beim Start von Π und terminiert das Programm, dann ist im Endzustand die Nachbedingung Ψ erfüllt.

Im obigen Beispiel könnte dies etwa lauten

$$\{x = a \ \wedge \ y = b\} \quad z := x;\ x := y;\ y := z \quad \{y = a \ \wedge \ x = b\}$$

mit Vorbedingung $\Phi \ = \ x = a \ \wedge \ y = b$ und Nachbedingung $\Psi \ = \ y = a \ \wedge \ x = b$.

Eine **axiomatische Semantik** liefert ein logisches System, um partielle Korrektheit nachzuweisen. Mit

$$\begin{aligned}\Phi_0 &:= x = a \ \wedge \ y = b\\ \Phi_1 &:= z = a \ \wedge \ y = b\\ \Phi_2 &:= z = a \ \wedge \ x = b\\ \Phi_3 &:= y = a \ \wedge \ x = b\end{aligned}$$

läßt sich in einem einfachen Kalkül folgende Deduktion führen:

$$\begin{array}{rcll} \{\Phi_0\}\ z := x\ \{\Phi_1\} & \wedge & \{\Phi_1\}\ x := y\ \{\Phi_2\} & \Longrightarrow \quad \{\Phi_0\}\ z := x;\ x := y\ \{\Phi_2\} \\ \{\Phi_0\}\ z := x;\ x := y\ \{\Phi_2\} & \wedge & \{\Phi_2\}\ y := z\ \{\Phi_3\} & \Longrightarrow \quad \{\Phi_0\}\ z := x;\ x := y;\ y := z\ \{\Phi_3\} \end{array}$$

Wir werden sehen, daß je nach Aufgabenstellung diese verschiedenen Semantiken ihre Vor- und Nachteile besitzen.

1.5 Die Sprache `WHILE`

Dazu betrachten wir als grundlegendes Beispiel die einfache Programmiersprache `WHILE`.

Syntax von `WHILE`:
Numerale `NUM`, repräsentiert durch die Notation m (Meta-Variable),
Variable `VAR`, repräsentiert durch Notation x,
arithmetische Ausdrücke `AEXP`, repräsentiert durch a,
Boolesche Ausdrücke `BEXP`, repräsentiert durch b,
Anweisungen `STN`, repräsentiert durch S.

Meta-Variable können mit zusätzlichen Kennzeichen wie m' oder Subscripts m_1, m_2 versehen werden. Syntaktische Details werden wir hier nicht genauer betrachten. Als Struktur von Numeralen kann man beispielsweise Strings über Ziffern wählen, von Variablen Strings über Buchstaben und Ziffern, die mit einem Buchstaben beginnen. Ausdrücke und Anweisungen ergeben sich wie folgt.

$$\begin{aligned} a &== m \mid x \mid a_1 + a_2 \mid a_1 \cdot a_2 \mid a_1 - a_2\\ b &== \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \ \wedge \ b_2\\ S &== x := a \mid \texttt{skip} \mid S_1 \ ; \ S_2 \mid \texttt{if}\ b\ \texttt{then}\ \ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ \ b\ \texttt{do}\ \ S' \end{aligned}$$

Beispiel `WHILE`-Programm für die Fakultätsfunktion:

$$\begin{array}{l} y := 1; \\ \texttt{while}\ \ \neg(x = 1)\ \ \texttt{do}\ \ (y := y \cdot x;\ x := x - 1) \end{array}$$

Die Semantik dieser Sprache werden wir mit Hilfe sogenannter **semantischer Funktionen** für jede syntaktische Kategorie festlegen.

1.6 Die Semantik von Ausdrücken

Als erstes betrachten wir Numerale und nehmen an, daß diese im Binärsystem notiert werden, d.h.

$$m \;==\; 0 \mid 1 \mid m'0 \mid m'1 \;.$$

Wir definieren eine Funktion

$$\mathcal{N} \;:\; \texttt{NUM} \to \mathbb{Z} \;,$$

die semantische Funktion der Numerale. Für $m \in \texttt{NUM}$ bezeichne $\mathcal{N}[[m]]$ die Zahl, die durch die semantische Funktion spezifiert wird. Wir erhalten

$$\begin{aligned} N[[0]] &:= 0 \\ N[[1]] &:= 1 \\ N[[m\ 0]] &:= 2 \cdot \mathcal{N}[[m]] \\ N[[m\ 1]] &:= 2 \cdot \mathcal{N}[[m]] + 1 \end{aligned}$$

Lemma 1.1 *Durch die obige Definition wird eine totale Funktion $\mathcal{N}$ auf* `NUM` *definiert.*

Beweis: Induktiv über die Syntax der Numerale.

Generell läßt sich diese Vorgehensweise wie folgt beschreiben:

Kompositionale Definition:

- jede syntaktische Kategorie wird durch eine abstrakte Syntax mit Basiselementen und Kompositionsoperatoren definiert. Dabei besitzen diese Operatoren eine eindeutige Zerlegung.
- Die Semantik wird durch ein kompositionales Konstrukt festgelegt mit Werten für die Basiselemente und Konstrukturen für die Operatoren.

Beweise lassen sich dann mit Hilfe **strukturelle Induktion** führen:

- zeige, daß eine Eigenschaft für die Basiselemente der syntaktischen Katrgorie gilt,
- zeige, daß die Eigenschaft bei jedem Komposistionsoperator erhalten bleibt, d.h. als Induktionshypothese wird angenommen, daß diese für jeden Teilausdruck gilt und dann im Induktionsschritt nachgewiesen, daß dies auch für den Gesamtausdruck gilt.

Semantische Funktionen

Im folgenden bezeichne `op` eine der arithmetischen Operationen $+, -, \cdot$ und `comp` einen der Vergleiche $=$ oder $\leq$.

Für Variable führen wir den Begriff **Zustand** ein. Genauer gesprochen werden Funktionen

$$\sigma \ : \ \mathtt{VAR} \to \mathbb{Z}$$

betrachtet, die jeder Variablen einen Wert zuordnen. Die Menge dieser Abbildungen bezeichnen wir mit

$$\mathtt{STATE} \ := \ \{\sigma \mid \sigma \ : \ \mathtt{VAR} \to \mathbb{Z}\} \ .$$

Die Semantik arithmetischer Ausdrücke wird zustandsabhängig definiert. Dies wird beschrieben durch die Funktion

$$\mathcal{A} \ : \ \mathtt{AEXP} \ \times \ \mathtt{STATE} \to \mathbb{Z} \ .$$

Alternativ kann man die Auswertung auch als ein zweistufiges Verfahren ansehen und $\mathcal{A}$ interpretieren in der Form

$$\mathcal{A} \ : \ \mathtt{AEXP} \to (\mathtt{STATE} \to \mathbb{Z}) \ .$$

Bespielsweise ergibt $\mathcal{A}[[x+1]]$ eine neue Zustandsfunktion und $\mathcal{A}[[x+1]](\sigma)$ den Wert des arithmetischen Ausdrucks $x+1$ im Zustand σ. Wir definieren

$$\begin{aligned}
\mathcal{A}[[n]]\sigma(x) &:= \mathcal{N}(n) \ , \\
\mathcal{A}[[x]]\sigma(x) &:= \sigma(x) \ , \\
\mathcal{A}[[a_1 \, \mathtt{op} \, a_2]]\sigma(x) &:= \mathcal{A}[[a_1]]\sigma(x) \, \mathtt{op} \, \mathcal{A}[[a_2]]\sigma(x) \ .
\end{aligned}$$

Beispiel

Für die Semantik Boolescher Ausdrücke dient die Funktion

$$\mathcal{B} \ : \ \mathtt{BEXP} \ \times \ \mathtt{STATE} \to \mathtt{TF} \ ,$$

wobei $\mathtt{TF} := \{\mathtt{tt}, \mathtt{ff}\}$ die Booleschen Wahrheitswerte `true` und `false` bezeichne. Wir definieren und einen

$$\begin{aligned}
\mathcal{B}[[\ \mathtt{true}\ ]]\sigma(x) &:= \mathtt{tt} \ , \\
\mathcal{B}[[\ \mathtt{false}\ ]]\sigma(x) &:= \mathtt{ff} \ , \\
\mathcal{B}[[a_1 \, \mathtt{comp} \, a_2]]\sigma(x) &:= \begin{cases} \mathtt{tt} & \text{falls } \mathcal{A}[[a_1]]\sigma \ \mathtt{comp} \ \mathcal{A}[[a_2]]\sigma, \\ \mathtt{ff} & \text{sonst,} \end{cases} \\
\mathcal{B}[[\neg b]]\sigma(x) &:= \begin{cases} \mathtt{ff} & \text{falls } \mathcal{B}[[b]]\sigma = \mathtt{tt}, \\ \mathtt{tt} & \text{sonst,} \end{cases} \\
\mathcal{B}[[b_1 \ \wedge \ b_2]]\sigma(x) &:= \begin{cases} \mathtt{tt} & \text{falls } \mathcal{B}[[b_1]]\sigma = \mathtt{tt} \text{ und } \mathcal{B}[[b_2]]\sigma = \mathtt{tt}, \\ \mathtt{ff} & \text{sonst.} \end{cases}
\end{aligned}$$

Freie Variable

Die Menge $\mathtt{FV}(a)$ der (freien) Variablen eines arithmetischen Ausdrucks ist wie folgt definiert:

$$\begin{aligned} \mathtt{FV}(n) &:= \emptyset \ , \\ \mathtt{FV}(x) &:= \{x\} \ , \\ \mathtt{FV}(a_1 \,\mathtt{op}\, a_2) &:= \mathtt{FV}(a_1) \cup \mathtt{FV}(a_2) \ . \end{aligned}$$

Lemma 1.2 *Ist a ein arithmetischer Ausdruck und sind σ, σ' Zustände mit $\sigma(x) = \sigma'(x)$ für alle $x \in \mathtt{FV}(a)$, dann gilt $\mathcal{A}[[a]]\sigma = \mathcal{A}[[a]]\sigma'$.*

Substitution

Lemma 1.3

$$\mathcal{A}[[a\ [y \mapsto a_0]\ ]]\sigma \ = \ \mathcal{A}[[a]]\sigma[y \mapsto \mathcal{A}[[a_0]]\sigma] \ .$$

2 Operationale Semantik

Eine operationale Semantik erklärt Programme, indem sie spezifiziert, wie diese auszuführen sind. Dazu wird definiert, wie Zustände durch Anweisungen verändert werden. Man unterscheidet zwischen

- natürliche Semanik,
- strukturelle operationale Semanik.

Zur Beschreibung verwenden wir ein Transitionssystem mit den Elementen

$$\begin{array}{rl} \sigma & \text{ein Terminal, ein Endzustand,} \\ \langle S, \sigma \rangle & \text{Anweisung wird im Zustand } \sigma \text{ ausgeführt.} \end{array}$$

Ein Übergang wird in der Form $\langle S, \sigma \rangle \to \sigma'$ notiert.

2.1 Natürliche Semantik

Eine Regel hat die Form

$$\frac{\langle S_1, \sigma_1 \rangle \to \sigma_1', \ldots, \langle S_t, \sigma_t \rangle \to \sigma_t'}{\langle S, \sigma \rangle \to \sigma'} \qquad \text{falls [Bedingungen] .}$$

Die Ausdrücke über der Linie bilden die **Voraussetzungen**, die Transition unter der Linie die **Folgerung**. Zusätzlich können rechts noch eine Menge von **Bedingungen** spezifiziert werden. Die S_i bei den Voraussetzungen sind Bestandteile der Anweisung (des Programms) S. Regeln ohne Voraussetzungen werden auch als **Axiome** bezeichnet.

Genau genommen ist das erste Axiom ein Schema, da x, a und σ Meta-Variable sind. Beispiel für eine Instanz wäre das Axiom $\langle x := x + 1, \sigma_0 \rangle \to \sigma_0[x \mapsto 1]$, wobei σ_0 den Zustand bezeichne, bei dem alle Variablen den Wert 0 haben.

Beispiel 2.1 *Anwendung der Kompositions- und der* `if` *-Regeln.*

Für das `while` -Konstrukt gibt es eine Regel und ein Axiom (im Fall, daß die Schleifen-Bedingung nicht erfüllt ist).

Ableitungsbäume

Um für einen Zustand und eine Anweisung den Endzustand zu bestimmen, versucht man, einen Ableitungsbaum zu finden. Seine Blätter sind Axiome, seine Verzweigungen repräsentieren die Anwendung einer Regel. Für die Folgerung, die durch einen internen Knoten dargestellt wird, muß für jede Voraussetzung ein geeigneter Ableitungsbaum gefunden werden.

$[\texttt{ass}_{\texttt{ns}}]$	$\langle x := a, \sigma\rangle \to \sigma[x \mapsto \mathcal{A}[[a]]\sigma]$	
$[\texttt{skip}_{\texttt{ns}}]$	$\langle\ \texttt{skip}\ , \sigma\rangle \to \sigma$	
$[\texttt{comp}_{\texttt{ns}}]$	$\dfrac{\langle S_1, \sigma\rangle \to \sigma',\ \langle S_2, \sigma'\rangle \to \sigma''}{\langle S_1; S_2, \sigma\rangle \to \sigma''}$	
$[\texttt{if}^{\texttt{tt}}_{\texttt{ns}}]$	$\dfrac{\langle S_1, \sigma\rangle \to \sigma'}{\langle\ \texttt{if}\ \ b\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2, \sigma\rangle \to \sigma'}$	falls $\mathcal{B}[[b]]\sigma = \texttt{tt}$
$[\texttt{if}^{\texttt{ff}}_{\texttt{ns}}]$	$\dfrac{\langle S_2, \sigma\rangle \to \sigma'}{\langle\ \texttt{if}\ \ b\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2, \sigma\rangle \to \sigma'}$	falls $\mathcal{B}[[b]]\sigma = \texttt{ff}$
$[\texttt{while}^{\texttt{tt}}_{\texttt{ns}}]$	$\dfrac{\langle S, \sigma\rangle \to \sigma',\ \langle\ \texttt{while}\ \ b\ \ \texttt{do}\ \ S, \sigma'\rangle \to \sigma''}{\langle\ \texttt{while}\ \ b\ \ \texttt{do}\ \ S, \sigma\rangle \to \sigma''}$	falls $\mathcal{B}[[b]]s = \texttt{tt}$
$[\texttt{while}^{\texttt{ff}}_{\texttt{ns}}]$	$\langle\ \texttt{while}\ \ b\ \ \texttt{do}\ \ S,\ \sigma\rangle \to \sigma$	falls $\mathcal{B}[[b]]s = \texttt{ff}$

Tabelle 1: Natürliche Semantik für `WHILE`

Beispiel 2.2 *Programm für Fakultätsfunktion:*
$\Pi:\quad y := 1\,;\ \texttt{while}\ \neg(x = 1)\ \ \texttt{do}\ \ (y := y \cdot x;\ x := x - 1)$
Ist s *ein Zustand mit* $[x \mapsto 3]$ *dann ist zu zeigen:* $\langle \Pi, \sigma\rangle \to \sigma[y \mapsto 6]\ [x \mapsto 1]$
Beweis durch einen Ableitungsbaum

Definition 2.1 *Terminierung und Looping einer Anweisung:*
Es sei S *eine Anweisung und* σ *ein Zustand. Falls es einen Zustand* σ' *gibt, so daß* $\langle S, \sigma\rangle \to \sigma'$ *, sagen wir "*S **terminiert** *auf* σ *", andernfalls "*S **loopt** *auf* σ *".*

Definition 2.2 *Semantische Äquivalenz*
Zwei Anweisungen S_1, S_2 *heißen* **semantisch äquivalent***, falls für alle Zustände* σ, σ' *gilt:* $\langle S_1, \sigma\rangle \to \sigma' \iff \langle S_2, \sigma\rangle \to \sigma'$ *.*

Lemma 2.1 *Äquivalenz der* `while` *-Anweisung*
Π_1 := `while` b `do` S *und* Π_2 := `if` b `then` $(S;$ `while` b `do` $S)$ `else skip` *sind äquivalent.*

Beweis: Die Beweismethodik **Induktion über den Aufbau der Ableitungsbäume** funktioniert wie folgt:

- zeige, daß eine Eigenschaft für alle elementaren Ableitungsbäume gilt, d.h. für alle Axiome des Transitionssystems

- zeige, daß die Eigenschaft für alle zusammengesetzten Ableitungsbäume gilt:
  unter der Voraussetzung, daß die Eigenschaft für alle Voraussetzungen einer Regel erfüllt ist, ist nachzuweisen, daß sie auch für die Folgerung gilt, wenn die Bedingungen der Regel erfüllt sind.

Definition 2.3 *Determinismus*
Eine Semantik heißt **deterministisch**, *falls für alle Anweisungen* S *und Zustände* $\sigma, \sigma_1, \sigma_2$ *gilt:*

$$\langle S, \sigma\rangle \to \sigma_1 \quad \wedge \quad \langle S, \sigma\rangle \to \sigma_2 \qquad \Longrightarrow \qquad \sigma_1 = \sigma_2 \ .$$

Durch Induktion über die Ableitungsbäume, läßt sich auf einfache Weise zeigen:

Theorem 2.1 *Die natürliche Semantik ist deterministisch.*

Definition 2.4 *Die Menge der partiellen Funktionen* $f : \mathtt{STATE} \to \mathtt{STATE}$ *von der Menge* `STATE` *in sich bezeichnen wir mit* $\mathcal{F}_{\mathtt{STATE}}$.

Die natürliche Semantik von `WHILE` *wird beschrieben durch die semantische Funktion* $\mathcal{S}_{\mathtt{ns}}$, *welche Anweisungen auf Funktionen in* $\mathcal{F}_{\mathtt{STATE}}$ *abbildet, d.h.* $\mathcal{S}_{\mathtt{ns}} : \mathtt{STN} \to \mathcal{F}_{\mathtt{STATE}}$. *Entgegen der in der Mathematik üblichen Notation schreiben wir* $\mathcal{S}_{\mathtt{ns}}[[S]]$ *für Funktionswerte semantischer Funktion bei Argument* S . $\mathcal{S}_{\mathtt{ns}}[[S]]$ *definiert eine Funktion* $f \in \mathcal{F}_{\mathtt{STATE}}$ *auf der Menge der Zustände. Für einen Zustand* σ *notieren wir dann den Zustand* $f(\sigma) = \sigma'$ *in der Form* $\sigma' = \mathcal{S}_{\mathtt{ns}}[[S]]\sigma$ *ohne weitere Klammern. Im Falle der natürlichen Semantik ist er definiert durch*

$$\mathcal{S}_{\mathtt{ns}}[[S]]\sigma \ := \ \begin{cases} \sigma' & \textit{falls } \langle S, \sigma\rangle \to \sigma', \\ \bot & \textit{andernfalls.} \end{cases}$$

2.2 Strukturelle operationale Semantik

Anstelle einer Top-down-Sicht im Falle der natürlichen Semantik soll nunmehr die Ausführung von Anweisung bottom-up spezifiziert werden. Dazu zerlegen wir ein Anweisung schrittweise in ihre einzelnen Elemente.

Definition 2.5 *Einzelschritt-Transitionen*
Das Transitionssystem für die strukturell operationale Semantik wird beschrieben durch einzelne Schritte der Form

$$\langle S, \sigma\rangle \implies \gamma$$

wobei die Konfiguration γ *entweder von der Form* $\langle S', \sigma'\rangle$ *ist oder nur ein einzelner Zustand* σ' . *Im Fall sagen wir,* S *ist noch* **nicht vollständig ausgeführt** *sondern nur der 1. Schritt und es verbleibt die Restanweisung* S' . *Im anderen Fall ist* S **komplett ausgeführt** *und das Ergebnis ist eine (terminale) Konfiguration* σ' .

$[\texttt{ass}_{\texttt{sos}}]$	$\langle x := a, \sigma\rangle \implies \sigma[x \mapsto \mathcal{A}[[a]]s]$
$[\texttt{skip}_{\texttt{sos}}]$	$\langle\ \texttt{skip}\ , \sigma\rangle \implies \sigma$
$[\texttt{comp}^1_{\texttt{sos}}]$	$\dfrac{\langle S_1, \sigma\rangle \implies \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \implies \langle S_1'; S_2, \sigma'\rangle}$
$[\texttt{comp}^2_{\texttt{sos}}]$	$\dfrac{\langle S_1, \sigma\rangle \implies \sigma'}{\langle S_1; S_2, \sigma\rangle \implies \langle S_2, \sigma'\rangle}$
$[\texttt{if}^{\texttt{tt}}_{\texttt{sos}}]$	$\langle\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \implies \langle S_1, \sigma\rangle\ \texttt{if}\ \mathcal{B}[[b]]s = \texttt{tt}$
$[\texttt{if}^{\texttt{ff}}_{\texttt{sos}}]$	$\langle\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \implies \langle S_2, \sigma\rangle\ \texttt{if}\ \mathcal{B}[[b]]s = \texttt{ff}$
$[\texttt{while}_{\texttt{sos}}]$	$\langle\ \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \implies \langle\ \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \ \texttt{skip}\ , \sigma\rangle$

Tabelle 2: Strukturelle operationale Semantik für `WHILE`

Ist es möglich, eine Konfiguration $\gamma_0 = \langle S_0, \sigma_0\rangle$ durch eine Folge von Ableitungen der Länge k in eine Konfiguration γ_k zu überführen, so notieren wir dies durch $\gamma_0 \Longrightarrow^k \gamma_k$. $\Longrightarrow^*$ bezeichne den transitiven Abschluß der Relation $\implies$.

Terminierung und Looping sind bei einer operationalen Semantik etwas aufwendiger zu definieren.

Definition 2.6 *Eine Konfiguration $\langle S, \sigma\rangle$* **hängt**, *falls es kein γ gibt mit $\langle S, \sigma\rangle \implies \gamma$. Eine Folge von Transitionen*

$$\langle S_0, \sigma_0\rangle \implies \langle S_1, \sigma_1\rangle \ldots \implies \langle S_{\tau-1}, \sigma_{\tau-1}\rangle \implies \sigma_\tau\ ,$$

die in einer terminalen Konfiguration σ_τ endet, sowie eine Folge

$$\langle S_0, \sigma_0\rangle \implies \langle S_1, \sigma_1\rangle \ldots \implies \langle S_\tau, \sigma_\tau\rangle\ ,$$

die in einer hängenden Konfiguration $\langle S_\tau, \sigma_\tau\rangle$ endet, oder eine unendliche Folge

$$\langle S_0, \sigma_0\rangle \implies \langle S_1, \sigma_1\rangle \implies \langle S_2, \sigma_2\rangle \implies \ldots\ ,$$

nennen wir eine **Ableitungssequenz**.
Eine Anweisung S **terminiert auf einem Zustand σ**, *falls $\langle S, \sigma\rangle$ eine endliche Ableitungssequenz besitzt. Sie terminiert* **erfolgreich**, *wenn dabei eine eine terminale Konfiguration erreicht wird, d.h. $\langle S, \sigma\rangle \Longrightarrow^* \sigma'$ für einen Zustand σ'.*
Eine Anweisung S **loopt** *auf einem Zustand σ, falls $\langle S, \sigma\rangle$ eine unendliche Ableitungssequenz besitzt.*

Lemma 2.2 *Gilt* $\langle S_1; S_2,\ \sigma\rangle \Longrightarrow^k \sigma'$ *, dann gibt es einen Zustand* σ'' *und natürliche Zahlen* k_1, k_2 *mit* $k = k_1 + k_2$ *, so daß* $\langle S_1, \sigma\rangle \Longrightarrow^{k_1} \sigma''$ *und* $\langle S_2, \sigma''\rangle \Longrightarrow^{k_2} \sigma'$ *.*

Beweis: Induktion über die Länge der Ableitungssequenzen ∎

Definition 2.7 *Semantische Äquivalenz*
Zwei Anweisungen S_1, S_2 *heißen* **semantisch äquivalent***, falls für alle Zustände* σ *:*

- *für alle terminalen oder hängenden Konfigurationen* γ *gilt:* $\langle S_1, \sigma\rangle \Longrightarrow^* \gamma$ *genau dann, wenn auch* $\langle S_2, \sigma\rangle \Longrightarrow^* \gamma$ *,*
- $\langle S_1, \sigma\rangle$ *besitzt eine unendliche Ableitungssequenz genau dann, wenn dies auch für* $\langle S_2, \sigma\rangle$ *gilt.*

Die semantische Funktion $\mathcal{S}_{\mathtt{sos}} : \mathtt{STN} \to \mathcal{F}_{\mathtt{STATE}}$ *bildet ähnlich wie im Fall der natürlichen Semantik Anweisungen auf partielle Funktionen auf der Menge der Zustände ab. Sie wird definiert duch*

$$\mathcal{S}_{\mathtt{sos}}[[S]]\sigma \ := \ \begin{cases} \sigma' & \text{falls } \langle S, \sigma\rangle \Longrightarrow^* \sigma', \\ \bot & \text{andernfalls.} \end{cases}$$

2.3 Äquivalenz

Theorem 2.2 *Für jede Anweisung* S *in* `WHILE` *gilt* $\mathcal{S}_{\mathtt{ns}}[[S]] = \mathcal{S}_{\mathtt{sos}}[[S]]$ *, d.h. terminiert* S *für eine der Semantiken, dann auch für die andere und die Zustände sind identisch, loopt* S *dagegen in einer der Semantiken, dann gilt das auch für die andere Semantik.*

Beweis: Durch Induktion über die Ableitungsbäume bzw. Länge der Ableitungen. Zeige:
1.) $\langle S, \sigma\rangle \to \sigma'$ impliziert $\langle S, \sigma\rangle \Longrightarrow^* \sigma'$,
2.) $\langle S, \sigma\rangle \Longrightarrow^k \sigma'$ impliziert $\langle S, \sigma\rangle \to \sigma'$. ∎

2.4 Erweiterungen von `WHILE`

Als nächstes wollen wir Erweiterungen der Programmiersprache `WHILE` durch gängige Programmierkonstrukte betrachten und untersuchen, wie sich dies auf die beiden Semantiken auswirkt. Hierbei werden deutlich Unterschiede zwischen den beiden Semantiken sichtbar.

2.4.1 Abbruch

Der Syntax von `WHILE` wird noch die elementare Anweisung `abort` hinzugefügt, die die Programmausführung stoppt. Für jedes Tupel $\langle$ `abort` $, \sigma\rangle$ soll `abort` hängen.

Strukturell operational ist `abort` weder semantisch äquivalent zu `skip` , da dort die Transition $\langle$ `skip` $, \sigma \rangle \implies \sigma$ möglich ist, noch zu

$$S_\infty := \texttt{while true do skip} ,$$

weil damit ein Programm in eine unendliche Schleife gerät:

$$\langle \texttt{while true do skip} , \sigma \rangle \Longrightarrow^* \langle \texttt{while true do skip} , \sigma \rangle .$$

Die natürliche Semantik kann zwar auch zwischen `abort` und `skip` unterscheiden, nicht jedoch zwischen `abort` und S_∞ , da diese beiden Anweisungen semantisch äquivalent sind. Damit können ein Hängen und eine unendliche Schleife durch diese Semantik nicht differenziert werden.

Die strukturelle operationale Semantik besitzt dagegen im ersten Fall keine Ableitungssequenz, im zweiten dagegen eine unendlicher Länge.

Erweitert man die Zustandsmenge durch einen speziellen Fehlerzustand und modelliert ein `abort` durch den Übergang in diesen Fehlerzustand, dann ist auch die natürliche Semantik in der Lage, zwischen `abort` und S_∞ zu unterscheiden.

2.4.2 Nichtdeterminismus

Wir erweitern nun `WHILE` durch den Operator `or` , d.h. eine Anweisung kann zusammengesetzt werden in der Form S_1 `or` S_2 .

Die natürliche Semantik unterdrückt ein unendliches Looping soweit möglich, die strukturell operationale Semantik dagegen nicht.

Aufgabe 2.1 *Die Anweisung* `random`(x) *setzt* x *auf eine beliebige natürliche Zahl. Frage: wird dadurch die Sprache* `WHILE` *erweitert? Man untersuche dies für beide Semantiken.*

2.4.3 Parallelismus

Als nächstes wird `WHILE` durch den Operator `par` vergrößert: S_1 `par` S_2 soll bedeuten, daß die beiden Anweisungen S_1 und S_2 in beliebiger Überlappung parallel ausgeführt werden können .

Die natürliche Semantik kann dies nicht ausdrücken, die strukturell operationale dagegen schon.

Aufgabe 2.2 *Die zusätzliche Anweisung:* `protect` S `end` *soll bedeuten, daß* S *als eine atomare Einheit ausgeführt werden soll. Man erweitere die beiden Semantiken entsprechend.*

2.4.4 Blöcke und Prozeduren

`WHILE` soll nun um die Deklaration von Variablen und Prozeduren erweitert werden. Wir wollen außerdem das Konzept von Umgebungen für Variable und Prozeduren einführen. Im ersten Schritt soll `WHILE` zu der Sprache `BLOCK` verallgemeinert werden, indem die Syntax um das Konstrukt

$$\texttt{begin}\ D_V\ S\ \texttt{end}$$

erweitert wird, wobei D_V eine Metavariable ist für die neue syntaktische Kategorie $\texttt{DEC}_V$ von Variablendeklarationen. Dessen Syntax ist spezifiziert als

$$D_V \ == \ \texttt{var}\ \ x := a;\ D_V \mid \lambda\ ,$$

wobei λ eine leere Deklaration beschreibt. Damit soll ausgedrückt werden, daß die Variablen, die innerhalb des durch `begin` und `end` definierten Blockes deklariert werden, lokal sind.

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

Die in einem Block deklarierten Variable definieren wir wie folgt:

$$\begin{aligned} \texttt{DV}(\ \texttt{var}\ x := a;\ D_V) &:= \{x\} \cup \texttt{DV}(D_V)\ , \\ \texttt{DV}(\lambda) &:= \emptyset\ . \end{aligned}$$

Das Transistionssystem wird erweitert um die Übergänge

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

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

Tabelle 3: Natürliche Semantik für Blöcke

Der Substitutionsoperator wird erweitert zu

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

Dadurch werden lokale Variable auf ihren ursprünglichen Wert zurückgesetzt, wenn der Block wieder verlassen wird.

Für die strukturell operationale Semantik ist eine entsprechende Spezifikation schwieriger. Gewöhnliche Zustände werden durch Laufzeit-Stacks erweitert.

$$[\texttt{var}_{\texttt{ns}}] \quad \frac{\langle D_V, \sigma[x \mapsto \mathcal{A}[[a]]s]\rangle \to_D \sigma'}{\langle\ \texttt{var}\ x := a;\ D_V,\ \sigma\rangle \to_D \sigma'}$$

$$[\texttt{none}_{\texttt{ns}}] \quad \langle \lambda, \sigma\rangle \to_D \sigma$$

Tabelle 4: Natürliche Semantik für Variablen-Deklaration

2.5 Die Programmiersprache PROC

Für Prozeduren wird die Syntax von BLOCK erweitert um

$$D_P \ == \ \texttt{proc}\ p\ \texttt{is}\ S;\ D_P \mid \lambda\ ,$$

wobei p eine Metavariable ist für die syntaktische Kategorie Pname von Prozedur-Namen. D_P ist eine Metavariable für die syntaktische Kategorie $\texttt{DEC}_P$ von Prozedur-Deklarationen.

Wir beschreiben 3 mögliche Semantiken für PROC, die sich durch den Auswirkungsbereich von Variablen und Prozeduren unterscheiden,

- Wirkungsbereich dynamisch für beide,
- dynamisch für Variable, aber statisch für Prozeduren,
- statisch für beide.

und erläutern dies am nachfolgenden Beispiel.

```
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
```

Im ersten Fall ruft im inneren Block die Prozedur $q =$ `call` p die im Inneren dieses Blockes definierte Prozedur $p = (x := x + 1)$ auf. Die lokale Variable x erhält durch diese Zuweisung der Wert $5 + 1 = 6$. Dieser Wert wird nachfolgend der Variablen y zugeweisen.

Im zweiten Fall ist q statisch durch `call` $p = (x := x \cdot 2)$ festgelegt. Die lokale Variable x und später auch y erhalten dann den Wert $5 \cdot 2 = 10$.

Im dritten Fall verwendet $p = (x := x \cdot 2)$ die statische Variable x, die in diesem Fall außerhalb des inneren Blockes mit 0 initialisiert wurde. Innerhalb von p erhält diese Variable somit den Wert $0 \cdot 2 = 0$. Bei der darauffolgenden Zuweisung $y := x$ wird jedoch die lokale Variable x mit dem aktuellen Wert 5 verwendet, d.h. auch y erhält den Wert 5.

2.5.1 Dynamische Wirkungsbereiche von Variablen und Prozeduren

$$[\mathtt{ass}_{\mathtt{ns}}] \quad \mathtt{env}_P \vdash \langle x := a, \sigma\rangle \to \sigma[x \mapsto \mathcal{A}[[a]]\sigma]$$

$$[\mathtt{skip}_{\mathtt{ns}}] \quad \mathtt{env}_P \vdash \langle\ \mathtt{skip}\ , \sigma\rangle \to \sigma$$

$$[\mathtt{comp}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_P \vdash \langle S_1, \sigma\rangle \to \sigma', \mathtt{env}_P \vdash \langle S_2, \sigma'\rangle \to \sigma''}{\mathtt{env}_P \vdash \langle S_1; S_2, \sigma\rangle \to \sigma''}$$

$$[\mathtt{if}^{\mathtt{tt}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_P \vdash \langle S_1, \sigma\rangle \to \sigma'}{\mathtt{env}_P \vdash \langle if\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \sigma\rangle \to \sigma'} \quad \text{falls } \mathcal{B}[[b]]\sigma = \mathtt{tt}$$

$$[\mathtt{if}^{\mathtt{ff}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_P \vdash \langle S_2, \sigma\rangle \to \sigma'}{\mathtt{env}_P \vdash \langle\ \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \sigma\rangle \to \sigma'} \quad \text{falls } \mathcal{B}[[b]]\sigma = \mathtt{ff}$$

$$[\mathtt{while}^{\mathtt{tt}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_P \vdash \langle S, \sigma\rangle \to \sigma', \mathtt{env}_P \vdash \langle\ \mathtt{while}\ b\ \mathtt{do}\ S, \sigma'\rangle \to \sigma''}{\mathtt{env}_P \vdash \langle\ \mathtt{while}\ b\ \mathtt{do}\ S, \sigma\rangle \to \sigma''} \quad \text{falls } \mathcal{B}[[b]]s = \mathtt{tt}$$

$$[\mathtt{while}^{\mathtt{ff}}_{\mathtt{ns}}] \quad \mathtt{env}_P \vdash \langle\ \mathtt{while}\ b\ \mathtt{do}\ S, \sigma\rangle \to \sigma \quad \text{falls } \mathcal{B}[[b]]s = \mathtt{ff}$$

$$[\mathtt{block}_{\mathtt{ns}}] \quad \frac{\langle\ D_V, \sigma\rangle\ \to_D\ \sigma',\ \mathtt{upd}_P(D_P,\ \mathtt{env}_P)\ \vdash\ \langle\ S,\ \sigma'\ \rangle\ \to\ \sigma''}{\mathtt{env}_P\ \vdash\ \langle\ \mathtt{begin}\ \ D_V\ D_P\ S\ \ \mathtt{end}\ ,\ \sigma\rangle\ \to\ \sigma''[DV(D_V) \to\ \sigma]}$$

$$[\mathtt{call}^{\mathtt{rec}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_P\ \vdash\ \langle S,\ \sigma\rangle \to \sigma'}{\mathtt{env}_P\ \vdash\ \langle\ \mathtt{call}\ p,\ \sigma\rangle\ \to \sigma'} \quad \text{wobei } \mathtt{env}_P\ p = S$$

Tabelle 5: Natürliche Semantik für PROC mit dynamischen Wirkungsbereichen

Um eine Anweisung `call` p auszuführen, wird der Rumpf von p ausgeführt. Die Namenszuweisung wird durch ein Prozedur-Environment beschrieben. Dazu definieren wir $\mathtt{env}_P$, welches die Anweisung im Rumpf einer Prozedur zurückgibt. $\mathtt{env}_P$ ist ein Element von $\mathtt{ENV}_P$, der Menge der Abbildungen von `Pname` nach `STN`.

Um die natürliche Semantik für PROC zu spezifizieren, werden neue Transitionen der Form

$$\mathtt{env}_P \vdash \langle S, \sigma\rangle \to \sigma'$$

eingeführt. Dies soll ausdrücken, daß man in der Umgebung einer Prozedur auf den Rumpf der Prozedur zugreifen kann.

Als neues Konstrukt für die Regel $\texttt{begin}\ \ D_V\ D_P\ S\ \ \texttt{end}$ wird $\texttt{upd}_P(D_P,\ \texttt{env}_P)$ eingeführt, spezifiziert durch:

$$\texttt{upd}_P(\ \texttt{proc}\ \ p\ \texttt{is}\ S;\ D_P, \texttt{env}_P)\ =\ \texttt{upd}_P(D_P,\ \texttt{env}_P[p \mapsto S])\ ,$$
$$\texttt{upd}_P(\lambda, \texttt{env}_P)\ =\ \texttt{env}_P\ .$$

2.5.2 Statische Wirkungsbereiche von Prozeduren

$$[\texttt{call}_{\texttt{ns}}] \quad \frac{\texttt{env}'_P\ \vdash\ \langle S, \sigma\rangle \to \sigma'}{\texttt{env}_P\ \vdash\ \langle\ \texttt{call}\ \ p,\ \sigma\rangle \to \sigma'} \quad \text{wobei } \texttt{env}_P\ p\ =\ (S, \texttt{env}'_P)$$

$$[\texttt{call}^{\texttt{rec}}_{\texttt{ns}}] \quad \frac{\texttt{env}'_P[p \mapsto (S, \texttt{env}'_P)]\ \vdash\ \langle S,\ \sigma\rangle \to s'}{\texttt{env}_P\ \vdash\ \langle\ \texttt{call}\ \ p,\ \sigma\rangle \to \sigma'} \quad \text{wobei } \texttt{env}_P\ p\ = (S,\ \texttt{env}'_P)$$

Tabelle 6: Prozedur-Aufrufe bei statischischen Wirkungsbereichen von Prozeduren

2.5.3 Statische Wirkungsbereiche von Variablen

Zusätzlich Umgebungen für Variable: $\texttt{ENV}_V$, Loc, Store

$$[\texttt{var}_{\texttt{ns}}] \quad \frac{\langle D_V,\ \texttt{env}_V[x \mapsto \ell], \texttt{sto}[\ell \mapsto v][\texttt{next} \mapsto \texttt{new}\ \ell]\rangle \to_D (\texttt{env}'_V, \texttt{sto}')}{\langle\ \texttt{var}\ \ x := a;\ D_V,\ \texttt{env}_V,\ \texttt{sto}\rangle \to_D (\texttt{env}'_V, \texttt{sto}')}$$
wobei $v = \mathcal{A}[[a]](\texttt{sto} \circ \texttt{env}_V)$ und $\ell = \texttt{sto}\ \texttt{next}$

$$[\texttt{none}_{\texttt{ns}}] \quad \langle \lambda,\ \texttt{env}_V, \texttt{sto}\rangle \to_D (\texttt{env}_V, \texttt{sto})$$

Tabelle 7: Deklaration von Variablen mit Lokationen

$[\mathtt{ass}_{\mathtt{ns}}]$ $\quad \mathtt{env}_V, \mathtt{env}_P \vdash \langle x := a, \mathtt{sto}\rangle \to \mathtt{sto}[\ell \to v]$
wobei $\ell = \mathtt{env}_V\ x$ und $v = \mathcal{A}[[a]](\mathtt{sto} \circ \mathtt{env}_V)$

$[\mathtt{skip}_{\mathtt{ns}}]$ $\quad \mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{skip}\ , \mathtt{sto}\rangle \to \mathtt{sto}$

$$[\mathtt{comp}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_V, \mathtt{env}_P \vdash \langle S_1, \mathtt{sto}\rangle \to \mathtt{sto}', \mathtt{env}_V, \mathtt{env}_P \vdash \langle S_2, \mathtt{sto}'\rangle \to \mathtt{sto}''}{\mathtt{env}_V, \mathtt{env}_P \vdash \langle S_1; S_2, \mathtt{sto}\rangle \to \mathtt{sto}''}$$

$$[\mathtt{if}^{\mathtt{tt}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_V, \mathtt{env}_P \vdash \langle S_1, \mathtt{sto}\rangle \to \mathtt{sto}'}{\mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{if}\ \ b\ \ \mathtt{then}\ \ S_1\ \ \mathtt{else}\ \ S_2, \mathtt{sto}\rangle \to \mathtt{sto}'} \quad \text{falls}\ \ \mathcal{B}[[b]](\mathtt{sto} \circ \mathtt{env}_V) = \mathtt{tt}$$

$$[\mathtt{if}^{\mathtt{ff}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_V, \mathtt{env}_P \vdash \langle S_2, \mathtt{sto}\rangle \to \mathtt{sto}'}{\mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{if}\ \ b\ \ \mathtt{then}\ \ S_1\ \ \mathtt{else}\ \ S_2, \mathtt{sto}\rangle \to \mathtt{sto}'} \quad \text{falls}\ \ \mathcal{B}[[b]](\mathtt{sto} \circ \mathtt{env}_V) = \mathtt{ff}$$

$$[\mathtt{while}^{\mathtt{tt}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}_V, \mathtt{env}_P \vdash \langle S, \mathtt{sto}\rangle \to \mathtt{sto}', \mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S, \mathtt{sto}'\rangle \to \mathtt{sto}''}{\mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S, \mathtt{sto}\rangle \to \mathtt{sto}''} \quad \text{falls}\ \ \mathcal{B}[[b]](\mathtt{sto} \circ \mathtt{env}_V) = \mathtt{tt}$$

$[\mathtt{while}^{\mathtt{ff}}_{\mathtt{ns}}]$ $\quad \mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S, \mathtt{sto}\rangle \to \mathtt{sto}$ $\quad$ falls $\ \mathcal{B}[[b]](\mathtt{sto} \circ \mathtt{env}_V) = \mathtt{ff}$

$$[\mathtt{block}_{\mathtt{ns}}] \quad \frac{\langle D_V, \mathtt{env}_V, \mathtt{sto}\rangle \to_D (\mathtt{env}'_V, \mathtt{sto}'), \quad \mathtt{env}'_V, \mathtt{env}'_P \vdash \langle S, \mathtt{sto}'\rangle \to \mathtt{sto}''}{\mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{begin}\ D_V\ D_P\ S\ \mathtt{end}\ , \mathtt{sto}\rangle \to \mathtt{sto}''}$$
wobei $\mathtt{env}'_P = \mathtt{upd}_P(D_P, \mathtt{env}'_V, \mathtt{env}_P)$

$$[\mathtt{call}_{\mathtt{ns}}] \quad \frac{\mathtt{env}'_V, \mathtt{env}'_P \vdash \langle S, \mathtt{sto}\rangle \to \mathtt{sto}'}{\mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{call}\ \ p, \mathtt{sto}\rangle \to \mathtt{sto}'}$$
wobei $\mathtt{env}_P p = (S, \mathtt{env}'_V, \mathtt{env}'_P)$

$$[\mathtt{call}^{\mathtt{rec}}_{\mathtt{ns}}] \quad \frac{\mathtt{env}'_V, \mathtt{env}'_P[p \to (S, \mathtt{env}'_V, \mathtt{env}'_P)] \vdash \langle S, \mathtt{sto}\rangle \to \mathtt{sto}'}{\mathtt{env}_V, \mathtt{env}_P \vdash \langle\ \mathtt{call}\ \ p, \mathtt{sto}\rangle \to \mathtt{sto}'}$$
wobei $\mathtt{env}_P p = (S, \mathtt{env}'_V, \mathtt{env}'_P)$

Tabelle 8: Natürliche Semantik für PROC mit statischen Wirkungsbereichen

3 Beweisbar korrekte Implementationen

Wir wollen nun ein `WHILE`-Programm in Assembler-Code übersetzen. Dazu definieren wir eine abstrakte Maschine mit einer operationalen Semantik. Die Korrektheit der Übersetzung wird dann dadurch gezeigt werden, daß

- die Übersetzung eines Programmes S in Code C sowie
- die anschließende Ausführung von C auf der abstrakten Maschine

zum gleichen Ergebnis führt wie die semantischen Funktionen $\mathcal{S}_{\mathtt{ns}}$ und $\mathcal{S}_{\mathtt{sos}}$.

3.1 Abstrakte Maschinen

Definition 3.1 *Die abstrakte Maschine* `AM` *wird beschrieben durch Konfigurationen* $\langle C, \Gamma, \sigma\rangle$ *mit*

- C *eine Sequenz von Instruktionen, die auszuführen sind,*
- $\Gamma \in \mathtt{STACK} := (\mathbb{Z} \cup \mathtt{TF})^*$ *der Inhalt des Operationsstacks,*
- $\sigma : \mathtt{VAR} \to \mathbb{Z}$ *der Speicher, der die Werte der Variablen hält (formal analog zu einem Element in* `STATE`*).*

Eine Konfiguration $\langle C, \Gamma, \sigma\rangle$ ist somit ein Element aus $\mathtt{CODE} \times \mathtt{STACK} \times \mathtt{STATE}$. Wenn C leer ist, nennen wir eine Konfiguration **terminal**. Übergänge zwischen Konfigurationen werden spezifiziert durch ein Transistionssystem

$$\langle C, \Gamma, \sigma\rangle \ \triangleright\ \langle C', \Gamma', \sigma'\rangle \ .$$

Die Instruktionen von `AM` sind in der Tabelle 9 aufgelistet. Man sieht, daß in jeder Konfiguration maximal eine Transition möglich ist, d.h. die so definierte Semantik für AM ist deterministisch.

Es bezeichne `CODE` die syntaktische Kategorie, die Folgen von Instruktionen beschreibt, und $\triangleright^k$ bzw. $\triangleright^*$ die k-Schritt-Transition bzw. den transitiven Abschluß von $\triangleright$. Ähnlich wie im vorigen Kapitel kann man leicht zeigen:

Lemma 3.1

Falls $\langle c_1, \Gamma_1, \sigma\rangle \ \triangleright^k\ \langle c', \Gamma', \sigma'\rangle$ *dann auch* $\langle c_1 : c_2, \Gamma_1 : \Gamma_2, \sigma\rangle \ \triangleright^k\ \langle c' : c_2, \Gamma' : \Gamma_2, \sigma'\rangle$.

Definition 3.2 *Die* **Exekutions-Funktion** $\mathcal{M}$ *einer abstrakten Maschine* `AM` *ist eine Abbildung* $\mathcal{M} : \mathtt{CODE} \to \mathcal{F}_{\mathtt{STATE}}$*, die für eine Instruktionssequenz* C *die Modifikation des Zustandes* σ *beschreibt, d.h.*

$$\mathcal{M}[[C]]\sigma \ := \ \begin{cases} \sigma' & \textit{falls } \langle C, \lambda, \sigma\rangle \ \triangleright\ \langle \lambda, \Gamma, \sigma'\rangle, \\ \bot & \textit{andernfalls.} \end{cases}$$

Eine **Berechnung von AM** *ist entweder eine endliche Folge von Konfigurationen*

$$K_0, K_1, \ldots, K_t \quad \text{mit} \quad K_i \triangleright K_{i+1} \;\; \textit{für alle} \;\; i \in [0..t-1]\,,$$

wobei K_t keine Nachfolger besitzt, oder eine unendliche Folge $K_0, K_1, K_2 \ldots$. Wie vorher definieren wir die Begriffe **terminal** *und* **looping** *für Berechnungen von AM.*

$\langle \texttt{PUSH} - n : c,\ \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ \mathcal{N}[[n]] : \Gamma,\ \sigma\rangle$	
$\langle \texttt{ADD} : c,\ z_1 : z_2 : \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ (z_1 + z_2) : \Gamma,\ \sigma\rangle$	falls $z_1,\ z_2 \in \mathbb{Z}$
$\langle \texttt{MULT} : c,\ z_1 : z_2 : \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ (z_1 \cdot z_2) : \Gamma,\ \sigma\rangle$	falls $z_1,\ z_2 \in \mathbb{Z}$
$\langle \texttt{SUB} : c,\ z_1 : z_2 : \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ (z_1 - z_2) : \Gamma,\ \sigma\rangle$	falls $z_1,\ z_2 \in \mathbb{Z}$
$\langle \texttt{TRUE} : c,\ \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ \texttt{tt} : \Gamma,\ \sigma\rangle$	
$\langle \texttt{FALSE} : c,\ \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ \texttt{ff} : \Gamma,\ \sigma\rangle$	
$\langle \texttt{EQ} : c,\ z_1 : z_2 : \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ (z_1 = z_2) : \Gamma,\ \sigma\rangle$	falls $z_1,\ z_2 \in \mathbb{Z}$
$\langle \texttt{LE} : c,\ z_1 : z_2 : \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ (z_1 \leq z_2) : \Gamma,\ \sigma\rangle$	falls $z_1,\ z_2 \in \mathbb{Z}$
$\langle \texttt{AND} : c,\ t_1 : t_2 : \Gamma,\ \sigma\rangle$	$\triangleright$	$\begin{cases} \langle c,\ \texttt{tt} : \Gamma,\ \sigma\rangle & \text{falls } t_1 = \texttt{tt} \text{ und } t_2 = \texttt{tt} \\ \langle c,\ \texttt{ff} : \Gamma,\ \sigma\rangle & \text{falls } t_1 = \texttt{ff} \text{ oder } t_2 = \texttt{ff} \end{cases}$	
$\langle \texttt{NEG} : c,\ t : \Gamma,\ \sigma\rangle$	$\triangleright$	$\begin{cases} \langle c,\ \texttt{ff} : \Gamma,\ \sigma\rangle & \text{falls } t = \texttt{tt} \\ \langle c,\ \texttt{tt} : \Gamma,\ \sigma\rangle & \text{falls } t = \texttt{ff} \end{cases}$	
$\langle \texttt{FETCH} - x : c,\ \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ (\sigma\ x) : \Gamma,\ \sigma\rangle$	
$\langle \texttt{STORE} - x : c,\ z : \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ \Gamma,\ \sigma[x \to z]\rangle$	falls $z \in \mathbb{Z}$
$\langle \texttt{NOOP} : c,\ \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c,\ \Gamma,\ \sigma\rangle$	
$\langle \texttt{BRANCH}(c_1,\ c_2) : c,\ t : \Gamma,\ \sigma\rangle$	$\triangleright$	$\begin{cases} \langle c_1 : c,\ \Gamma,\ \sigma\rangle & \text{falls } t = \texttt{tt} \\ \langle c_2 : c,\ \Gamma,\ \sigma\rangle & \text{falls } t = \texttt{ff} \end{cases}$	
$\langle \texttt{LOOP}(c_1,\ c_2) : c,\ \Gamma,\ \sigma\rangle$	$\triangleright$	$\langle c_1 : \texttt{BRANCH}(c_2 : \texttt{LOOP}(c_1,\ c_2),\ \texttt{NOOP}) : c,\ \Gamma,\ \sigma\rangle$	

Tabelle 9: Operationale Semantik für AM

Zur Analyse dieser abstrakten Maschinen verwenden wir analog zur operationalen Semantik als Beweistechnik Induktion über die Länge von Berechnungen. Abstrakte Maschinen modellieren reale Hardwarearchitekturen noch nicht hundertprozentig. Dazu sind noch weitere Vereinfachungen notwendig, auf die wir jedoch zunächst verzichten wollen, um die Beweisführung zu erleichtern.

1. Die abstrakte Maschine AM_1 referenziert Variable nicht durch Namen, sondern durch Adressen. Konfigurationen haben die Form $\langle C, \Gamma, \mu \rangle$, wobei μ eine Folge von Werten über $\mathbb{Z}$ ist. Anstelle von Operationen $\texttt{FETCH} - x$ und $\texttt{STORE} - x$ für eine Variable mit Namen x gibt es die Instruktionen $\texttt{GET} - n$ und $\texttt{PUT} - n$, die auf das n-te Elemente in μ zugreifen.

2. Die abstrakte Maschine AM_2 verzichtet auf `BRANCH`- und `LOOP`-Instruktionen und verwendet stattdessen Labels und einen Programm-Counter.

3. Die abstrakte Maschine AM_3 schließlich verwendet für Sprünge nur absolute Adressen.

3.2 Übersetzung in Maschinencode

Als nächstes müssen wir festlegen, wie `WHILE`-Programme in Code für AM übersetzt werden. Die folgenden Tabellen beschreiben dies.

$$
\begin{array}{lcl}
\mathcal{CA}[[n]] & = & \texttt{PUSH} - n \\
\mathcal{CA}[[x]] & = & \texttt{FETCH} - x \\
\mathcal{CA}[[a_1 + a_2]] & = & \mathcal{A}[[a_2]] : \mathcal{A}[[a_1]] : \texttt{ADD} \\
\mathcal{CA}[[a_1 \cdot a_2]] & = & \mathcal{A}[[a_2]] : \mathcal{A}[[a_1]] : \texttt{MULT} \\
\mathcal{CA}[[a_1 - a_2]] & = & \mathcal{A}[[a_2]] : \mathcal{A}[[a_1]] : \texttt{SUB} \\
\mathcal{CB}[[\texttt{ true }]] & = & \texttt{TRUE} \\
\mathcal{CB}[[\texttt{ false }]] & = & \texttt{FALSE} \\
\mathcal{CB}[[a_1 = a_2]] & = & \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]] : \texttt{EQ} \\
\mathcal{CB}[[a_1 \leq a_2]] & = & \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]] : \texttt{LE} \\
\mathcal{CB}[[\neg b]] & = & \mathcal{CB}[[b]] : \texttt{NEG} \\
\mathcal{CB}[[b_1 \wedge b_2]] & = & \mathcal{CB}[[b_2]] : \mathcal{CB}[[b_1]] : \texttt{AND}
\end{array}
$$

Tabelle 10: Übersetzung von Ausdrücken

Definition 3.3 *Die semantische Funktion* $\mathcal{S}_{am} : \texttt{STN} \to \mathcal{F}_{\texttt{STATE}}$ *ergibt sich durch Übersetzung eines* `WHILE`*-Programmes zunächst in Code für* `AM` *und anschließender Ausführung dieses Codes auf der Maschine, mit anderen Worten*

$$\mathcal{S}_{am}[[S]] \;=\; (\mathcal{M} \circ \mathcal{CS})[[S]] \;.$$

$$
\begin{array}{lcl}
\mathcal{CS}[[x := a]] & = & \mathcal{CA}[[a]] : \texttt{STORE} - x \\
\mathcal{CS}[[\texttt{ skip }]] & = & \texttt{NOOP} \\
\mathcal{CS}[[S_1; S_2]] & = & \mathcal{CS}[[S_1]] : \mathcal{CS}[[S_2]] \\
\mathcal{CS}[[\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]] & = & \mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[S_1]], \mathcal{CS}[[S_s]]) \\
\mathcal{CS}[[\texttt{while } b \texttt{ do } S]] & = & \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]])
\end{array}
$$

Tabelle 11: Übersetzung von WHILE-Ausdrücken

3.3 Semantische Äquivalenz der abstrakten Maschine

Die Korrektheit wird durch Betrachtung der einzelnen elementaren Anweisungen und Induktion über die Länge der Anweisungen nachgewiesen.

Theorem 3.1 *Für jedes* WHILE*-Statement* S *gilt:* $\mathcal{S}_{ns}[[S]] = \mathcal{S}_{am}[[S]]$.

Beweis: Es gilt zu beweisen:

- Wenn in einem gegebenen Zustand σ die Ausführung einer Anweisung S in einer der Semantiken terminiert, dann auch in der anderen und die Ergebniszustände sind identisch.
- Falls die Ausführung von S dagegen in einer der Semantiken loopt, dann auch in der anderen.

Als erstes betrachten wir Transitionen in der natürlichen Semantik

Lemma 3.2 *Für jede Anweisung* S *von* WHILE *und Zustände* σ, σ' *gilt:*

$$\textit{falls } \langle S, \sigma\rangle \to \sigma' \quad \textit{dann auch} \quad \langle \mathcal{CS}[[S]], \lambda, \sigma\rangle \ \triangleright^* \ \langle \lambda, \lambda, \sigma'\rangle \ .$$

Beweis: Induktion über den Aufbau der Ableitungsbäume für $\langle S, \sigma\rangle \to \sigma'$.

Fall $[\texttt{ass}_{\texttt{ns}}]$ Annahme $\langle x := a, \sigma\rangle \to \sigma'$, wobei $\sigma' = \sigma[x \mapsto \mathcal{A}[[a]]\sigma]$.
Es gilt $\mathcal{CS}[[x := a]] = \mathcal{CA}[[a]] : \texttt{STORE} - \texttt{x}$ und $\langle \mathcal{CA}[[a]], \lambda, \sigma\rangle \ \triangleright^* \ \langle \lambda, \mathcal{A}[[a]]\sigma, \sigma\rangle$.
Des weiteren können wir unter Verwendung von Lemma 3.1 schließen

$$
\begin{array}{rcl}
\langle \mathcal{CA}[[a]] : \texttt{STORE} - \texttt{x}, \lambda, \sigma\rangle & \triangleright^* & \langle \texttt{STORE} - \texttt{x}, (\mathcal{A}[[\texttt{a}]]\sigma), \sigma\rangle \\
& \triangleright & \langle \lambda, \lambda, \sigma[x \mapsto \mathcal{A}[[a]]\sigma]\rangle \ .
\end{array}
$$

Fall $[\texttt{skip}_{\texttt{ns}}]$ offensichtlich.

Fall $[\texttt{comp}_{\texttt{ns}}]$ Annahme $\langle S_1; S_2, \sigma\rangle \to \sigma''$.

Es gilt $\mathcal{CS}[[S_1; S_2]] \;=\; \mathcal{CS}[[S_1]] : \mathcal{CS}[[S_2]]$.
Die Induktionsvoraussetzung angewandt auf $\langle S_1, \sigma\rangle \to \sigma'$ und $\langle S_2, \sigma'\rangle \to \sigma''$ ergibt $\langle \mathcal{CS}[[S_1]], \lambda, \sigma\rangle \;\triangleright^*\; \langle \lambda, \lambda, \sigma'\rangle$ und $\langle \mathcal{CS}[[S_2]], \lambda, \sigma'\rangle \;\triangleright^*\; \langle \lambda, \lambda, \sigma''\rangle$.
Mit Lemma 3.1 folgt $\langle \mathcal{CS}[[S_1]] : \mathcal{CS}[[S_2]], \lambda, \sigma\rangle \;\triangleright^*\; \langle \mathcal{CS}[[S_2]], \lambda, \sigma'\rangle \;\triangleright^*\; \langle \lambda, \lambda, \sigma''\rangle$.

Fall $[\texttt{if}^{\texttt{tt}}_{\texttt{ns}}]$ Angenommen $\langle$ `if` b `then` S_1 `else` $S_2, \sigma\rangle \to \sigma'$
und es gelte $\mathcal{B}[[b]]\sigma = \texttt{tt}$ sowie $\langle S_1, \sigma\rangle \to \sigma'$.
Dann folgt $\mathcal{CS}[[$ `if` b `then` S_1 `else` $S_2]] \;=\; \mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}_1]], \mathcal{CS}[[\texttt{S}_2]])$.
Dies ergibt

$$\begin{aligned}
&\langle \mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}_1]], \mathcal{CS}[[\texttt{S}_2]]),\ \lambda,\ \sigma\rangle \\
&\quad \triangleright^* \langle \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}_1]], \mathcal{CS}[[\texttt{S}_2]]),\ (\mathcal{B}[[\texttt{b}]]\sigma),\ \sigma\rangle \\
&\quad \triangleright \langle \mathcal{CS}[[S_1]],\ \lambda,\ \sigma\rangle \\
&\quad \triangleright^* \langle \lambda, \lambda, \sigma'\rangle\,.
\end{aligned}$$

Der zweite Schritt folgt aus der Definition von `BRANCH` im Fall, daß das oberste Element des Stacks `tt` ist – der Wert von $\mathcal{B}[[b]]\sigma$.
Der dritte Schritt folgt aus der Induktionshypothese für $\langle S_1, \sigma\rangle \to \sigma'$.

Fall $[\texttt{if}^{\texttt{ff}}_{\texttt{ns}}]$ analog.

Fall $[\texttt{while}^{\texttt{tt}}_{\texttt{ns}}]$ Angenommen $\langle$ `while` b `do` $S, \sigma\rangle \to \sigma''$
und es gelte $\mathcal{B}[[b]]\sigma = \texttt{tt}$ sowie $\langle S, \sigma\rangle \to \sigma'$ und $\langle$ `while` b `do` $S, \sigma'\rangle \to \sigma''$.

Dann folgt $\mathcal{CS}[[$ `while` b `do` $S]] = \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]])$ und wir erhalten

$$\begin{aligned}
&\langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma\rangle \\
&\quad \triangleright \langle \mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), \lambda, \sigma\rangle \\
&\quad \triangleright^* \langle \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), (\mathcal{B}[[\texttt{b}]]\sigma), \sigma\rangle \\
&\quad \triangleright \langle \mathcal{CS}[[S]] : \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma\rangle\,.
\end{aligned}$$

Wir können die Induktionshypothese auf $\langle S, \sigma\rangle \to \sigma'$ und $\langle$ `while` b `do` $S, \sigma'\rangle \to \sigma''$ anwenden und erhalten

$$\langle \mathcal{CS}[[S]], \lambda, \sigma\rangle \;\triangleright^*\; \langle \lambda, \lambda, \sigma'\rangle \quad \text{sowie} \quad \langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda,\ \sigma'\rangle \;\triangleright^*\; \langle \lambda, \lambda, \sigma''\rangle\,.$$

Mit Lemma 3.1 ergibt sich

$$\begin{aligned}
&\langle \mathcal{CS}[[S]] : \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma\rangle \\
&\quad \triangleright^* \langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma'\rangle \\
&\quad \triangleright^* \langle \lambda, \lambda, \sigma''\rangle\,.
\end{aligned}$$

Fall $[\texttt{while}^{\texttt{ff}}_{\texttt{ns}}]$ Angenommen $\langle$ `while` b `do` $S, \sigma\rangle \to \sigma'$ gilt mit $\mathcal{B}[[b]]\sigma = \texttt{ff}$.

Dann können wir schließen also $\sigma = \sigma'$ und

$$\begin{aligned}
&\langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma\rangle \\
&\quad \triangleright \langle \mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), \lambda, \sigma\rangle
\end{aligned}$$

$$\begin{aligned}
&\triangleright^* \langle \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), (\mathcal{B}[[\texttt{b}]]\sigma), \sigma\rangle \\
&\triangleright \langle \texttt{NOOP}, \lambda, \sigma\rangle \\
&\triangleright \langle \lambda, \lambda, \sigma\rangle
\end{aligned}$$

Damit ist die Behauptung bewiesen. ∎

Lemma 3.3 *Für jede Anweisung S von* `WHILE` *und Zustände σ, σ' gilt:*

$$\textit{falls } \langle \mathcal{CS}[[S]], \lambda, \sigma\rangle \ \triangleright^k \ \langle \lambda, \Gamma, \sigma'\rangle \quad \textit{dann auch} \quad \langle S, \sigma\rangle \to \sigma' \quad \text{und} \quad \Gamma = \lambda \ .$$

Mit anderen Worten, falls die Ausführung des Codes von S in einem Zustand σ terminiert, dann terminiert die natürliche Semantik von S von σ aus in einem Zustand, der mit dem Zustand der terminalen Konfiguration übereinstimmt.

Beweis: Wir führen Induktion über k.
Der Fall $k = 0$ kann nicht eintreten, da $\mathcal{CS}[[S]] \neq \lambda$.
Die Behauptung gelte für alle $k \leq k_0$.

Fall $x := a$
Es gilt $\mathcal{CS}[[x := a]] = \mathcal{CA}[[a]] : \texttt{STORE}{-}x$.
Annahme $\langle \mathcal{CA}[[a]] : \texttt{STORE}{-}x, \lambda, \sigma\rangle \ \triangleright^{k_0+1} \ \langle \lambda, \Gamma, \sigma'\rangle$.
Man kann die Transitionsfolge zerlegen mit Hilfe einer Konfiguration $\langle \lambda, \Gamma'', \sigma''\rangle$ zu

$$\langle \mathcal{CS}[[a]], \lambda, \sigma\rangle \ \triangleright^{k_1} \ \langle \lambda, \Gamma'', \sigma''\rangle \quad \text{und} \quad \langle \texttt{STORE}{-}x, \Gamma'', \sigma''\rangle \ \triangleright^{k_2} \ \langle \lambda, \Gamma, \sigma'\rangle \ ,$$

wobei $k_1 + k_2 = k_0 + 1$ und $\Gamma'' = \mathcal{A}[[a]]\sigma$ und $\sigma'' = \sigma$. Nach Definition gilt $\sigma' = \sigma[x \mapsto \mathcal{A}[[a]]\sigma]$ und $\Gamma = \lambda$. Aus der Definition von $[\texttt{ass}_{\texttt{ns}}]$ folgt $\langle x := a, \sigma\rangle \to \sigma'$.

Fall `skip` offensichtlich.

Fall $S_1; S_2$
Annahme $\langle \mathcal{CS}[[S_1]] : \mathcal{CS}[[S_2]], \ \lambda, \sigma\rangle \ \triangleright^{k_0+1} \ \langle \lambda, \Gamma, \sigma''\rangle$.
Dann gibt es eine Konfiguration der form $\langle \lambda, \Gamma', \sigma'\rangle$, so daß

$$\langle \mathcal{CS}[[S_1]], \lambda, \sigma\rangle \ \triangleright^{k_1} \ \langle \lambda, \Gamma', \sigma'\rangle \quad \text{und} \quad \langle \mathcal{CS}[[S_2]], \Gamma'\sigma'\rangle \ \triangleright^{k_2} \ \langle \lambda, \Gamma, \sigma''\rangle \ ,$$

wobei $k_1 + k_2 = k_0 + 1$. Nach Induktionsvoraussetzung gilt $\langle S_1, \sigma\rangle \to \sigma''$ und $\Gamma' = \lambda$. Daher folgt $\langle \mathcal{CS}[[S_2]], \lambda, \sigma'\rangle \ \triangleright^{k_2} \ \langle \lambda, \Gamma, \sigma''\rangle$ und somit $\langle S_2, \sigma'\rangle \to \sigma''$ and $\Gamma = \lambda$.
Mit Hilfe von $[\texttt{comp}_{\texttt{ns}}]$ erhalten wir $\langle S_1; S_2, \sigma\rangle \to \sigma''$.

Fall `if` b `then` S_1 `else` S_2
Als Code wird generiert $\mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}_1]], \mathcal{CS}[[\texttt{S}_2]])$.

Annahme $\langle \mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}_1]], \mathcal{CS}[[\texttt{S}_2]]), \lambda, \sigma\rangle \ \triangleright^{\texttt{k}_0+1} \ \langle \lambda, \Gamma, \sigma'\rangle$.
Dann gibt es eine Konfiguration der Form $\langle \lambda, \Gamma'', \sigma''\rangle$ und $k_1 + k_2 = k_0 + 1$, so daß

$\langle \mathcal{CB}[[b]], \lambda, \sigma\rangle \ \triangleright^{k_1} \ \langle \lambda, \Gamma'', \sigma''\rangle$ und $\langle \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}_1]], \mathcal{CS}[[\texttt{S}_2]]), \Gamma'', \sigma''\rangle \ \triangleright^{\texttt{k}_2} \ \langle \lambda, \Gamma, \sigma'\rangle$.

Es muß gelten $\Gamma'' = \mathcal{B}[[b]]\sigma$ and $\sigma'' = \sigma$.
Falls $\mathcal{B}[[b]]\sigma = \texttt{tt}$ gibt es eine Konfiguration $\mathcal{CS}[[S_1]], \lambda, \sigma\rangle$ mit

$$(\mathcal{CS}[[S_1]], \lambda, \sigma\rangle \ \triangleright^{k_2-1} \ \langle \lambda, \Gamma, \sigma'\rangle \ .$$

Die Induktionshypothese liefert $\langle S_1, \sigma\rangle \to \sigma'$ and $\Gamma = \lambda$.
Die Regel $[\texttt{if}^{\texttt{tt}}_{\texttt{ns}}]$ ergibt dann $\langle$ `if` b `then` $\S_1$ `else` $S_2, \sigma\rangle \to \sigma'$.
Der Fall $\mathcal{B}[[b]]\sigma = \texttt{ff}$ ist analog.

Fall `while` b `do` S
Dies wird übersetzt in $\texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]])$, wir können daher annehmen

$$\langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma\rangle \ \triangleright^{k_0+1} \ \langle \lambda, \Gamma, \sigma''\rangle \ .$$

Nach Definition kann dies ersetzt werden durch

$\langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma\rangle$
$\quad \triangleright \langle \mathcal{CB}[[b]] : \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), \lambda, \sigma\rangle$
$\quad \triangleright^{k_0} \langle \lambda, \Gamma, \sigma''\rangle$.

Es gibt eine Konfiguration $\langle \lambda, \Gamma', \sigma'\rangle$ mit $\langle \mathcal{CB}[[b]], \lambda, \sigma\rangle \ \triangleright^{k_1} \ \langle \lambda, \Gamma', \sigma'$ und

$$\langle \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), \Gamma', \sigma'\rangle \ \triangleright^{\texttt{k}_2} \ \langle \lambda, \Gamma, \sigma''\rangle$$

mit $k_1 + k_2 = k_0$. Weiter können wir folgern $\Gamma' = \mathcal{B}[[b]]\sigma$ and $\sigma' = \sigma$.
Falls nun $\mathcal{B}[[b]]\sigma = \texttt{ff}$, folgt

$\langle \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), \mathcal{B}[[\texttt{b}]]\sigma, \sigma\rangle$
$\quad \triangleright \langle \texttt{NOOP}, \lambda, \sigma\rangle$
$\quad \triangleright \langle \lambda, \lambda, \sigma\rangle$,

d.h. $\Gamma = \lambda$ and $\sigma = \sigma''$.
Mit Hilfe der Regel $[\texttt{while}^{\texttt{ff}}_{\texttt{ns}}]$ erhalten wir $\langle$ `while` b `do` $S, \sigma\rangle \to \sigma''$ wie behauptet.
Im zweiten Fall $\mathcal{B}[[b]]\sigma = \texttt{tt}$ ergibt sich

$\langle \texttt{BRANCH}(\mathcal{CS}[[\texttt{S}]] : \texttt{LOOP}(\mathcal{CB}[[\texttt{b}]], \mathcal{CS}[[\texttt{S}]]), \texttt{NOOP}), \mathcal{B}[[\texttt{b}]]\sigma, \sigma\rangle$
$\quad \triangleright \langle \mathcal{CS}[[S]] : \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma\rangle$.

Ähnlich wie bei der Komposition findet man eine Konfiguration $\langle \lambda, \Gamma', \sigma'\rangle$, so daß
$\langle \mathcal{CS}[[S]], \lambda, \sigma\rangle \ \triangleright^{k_3} \ \langle \lambda, \Gamma'\sigma'\rangle$ und
$\langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \Gamma', \sigma'\rangle \ \triangleright^{k_4} \ \langle \lambda, \Gamma, \sigma''\rangle$

mit $k_3 + k_4 = k_2 - 1$. Wegen $k_3 \le k_0$ können wir die Induktionshypothese auf den ersten Teil anwenden und erhalten $\langle S, \sigma\rangle \to \sigma'$ mit $\Gamma' = \lambda$.
Analog folgt für $k_4 \le k_0$ die Transition $\langle \texttt{LOOP}(\mathcal{CB}[[b]], \mathcal{CS}[[S]]), \lambda, \sigma'\rangle \ \triangleright^{k_4} \ \langle \lambda, \Gamma, \sigma''\rangle$ und damit

$$\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma'\rangle \to \sigma'' \quad \text{und} \quad \Gamma = \lambda \ .$$

Aus der Regel $[\texttt{while}^{\texttt{tt}}_{\texttt{ns}}]$ ergibt sich dann $\langle$ `while` b `do` $S, \sigma\rangle \to \sigma''$.

Damit ist das Lemma bewiesen. ∎

3.4 Bisimulation

Alternativ kann die semantische Äquivalenz auch für die operationale Semantik

$$\mathcal{S}_{am}[[S]] = \mathcal{S}_{sos}[[S]]$$

durch eine sogenannte **Bisimulation** gezeigt werden. Dazu wird eine Äquivalenzrelation zwischen den Konfigurationen der operationalen Semantik und denen der AM definiert:

$$\begin{aligned} \langle S, \sigma \rangle &\equiv \langle \mathcal{CS}[[S]],\ \lambda,\ \sigma \rangle \\ \sigma &\equiv \langle \lambda,\ \lambda,\ \sigma \rangle \end{aligned}$$

Es bleibt dann zu zeigen, daß

- es für jede Transition in der operationalen Semantik eine Folge von Transitionen in der AM-Semantik gibt, die zu einer äquivalenten Konfiguration führen und daß
- man für jede Folge von Transitionen von AM, die mit einem leeren Stack beginnt und endet, eine analoge Folge in der operationenalen Semantik finden kann.

4 Denotationelle Semantik

In der denotationellen Semantik werden mathematische Objekte konstruiert, um die Auswirkung von Programmen zu beschreiben. Entscheidend ist dabei die Kompositionalität, d.h.

- es gibt eine semantische Klausel für jedes Basiselemente der syntaktischen Kategory und
- für jede Methode, in der syntaktischen Kategorie ein zusammengesetztes Element zu erzeugen, gibt es eine semantische Klausel, die direkt angewendet wird auf die Ausgangselemente der Komposition.

4.1 Formale Definition

Für das Folgende benötigen wird eine Hilfsfunktion

$$\texttt{cond}: \ (\texttt{STATE} \to \texttt{TF}) \ \times \mathcal{F}_{\texttt{STATE}} \times \mathcal{F}_{\texttt{STATE}} \to \mathcal{F}_{\texttt{STATE}}$$

definiert durch

$$\texttt{cond}(\texttt{p}, \texttt{g}_1, \texttt{g}_2)\ \sigma \ = \ \begin{cases} g_1\sigma & \text{falls } p\ \sigma = \texttt{tt}, \\ g_2\sigma & \text{falls } p\ \sigma = \texttt{ff}. \end{cases}$$

Wir betrachten Funktionale auf $\mathcal{F}_{\texttt{STATE}}$, d.h. Abbildungen $\hat{F}_{b,S}: \ \mathcal{F}_{\texttt{STATE}} \to \mathcal{F}_{\texttt{STATE}}$ definiert durch

$$\hat{F}_{b,S}: \ g \mapsto \texttt{cond}(\mathcal{B}[[\texttt{b}]], \texttt{g} \circ \ \mathcal{S}_{\texttt{ds}}[[\texttt{S}]], \texttt{id}) \ ,$$

wobei die Parameter b und S ein Boolescher Ausdruck bzw. eine Anweisung sind. Schließlich wird noch ein Fixpunktoperator `FIXP` für ein derartiges Funktional verwendet, um die Semantik einer `while` -Schleife zu spezifizieren.

Definition 4.1 *Die semantische Funktion* $\mathcal{S}_{ds} : \texttt{STN} \to \mathcal{F}_{\texttt{STATE}}$ *ist folgendermaßen definiert:*

$\mathcal{S}_{ds}[[x := a]]\ \sigma$	$=$	$\sigma[x \mapsto \mathcal{A}[[a]]\sigma]$
$\mathcal{S}_{ds}[[\texttt{skip}]]$	$=$	$\texttt{id}$
$\mathcal{S}_{ds}[[S_1; S_2]]$	$=$	$\mathcal{S}_{ds}[[S_2]] \circ \mathcal{S}_{ds}[[S_1]]$
$\mathcal{S}_{ds}[[\texttt{if b then S}_1\ \texttt{else S}_2]]$	$=$	$\texttt{cond}(\mathcal{B}[[\texttt{b}]], \mathcal{S}_{\texttt{ds}}[[\texttt{S}_1]], \mathcal{S}_{\texttt{ds}}[[\texttt{S}_2]])$
$\mathcal{S}_{ds}[[\texttt{while b do S}]]$	$=$	$\texttt{FIXP}\ \hat{\texttt{F}}_{\texttt{b,S}}$

Tabelle 12: Denotationelle Semantik für `WHILE`

Beispiel 1: `while` $\neg(x = 0)$ `do skip`

Probleme:
1.) ein Funktional $\hat{F}$ kann mehrere Fixpunkte besitzen,
2.) es gibt auch Funktionale ohne Fixpunkt.

Beispiel 2: Funktionale ohne Fixpunkt:

Beispiel 3: Beispiel `while` $\neg(x = 0)$ `do` $x := x - 1$
Diese `while` -Schleife bietet verschiedene Möglichkeiten für Fixpunkte.

Für das erste Problem wollen wir zusätzliche Bedingungen an einen Fixpunkt stellen, damit dieser eindeutig wird. Zunächst klassifizieren wir das mögliche Verhalten einer Schleife `while b do S`:

1. die Schleife terminiert:
   In diesem Fall gibt es eine endliche Folge von Zuständen $\sigma_1, \ldots, \sigma_\tau$, die auf den Anfangszustand σ_0 folgen, mit $\mathcal{S}_{ds}[[S]]\sigma_i = \sigma_{i+1}$ und $\mathcal{B}[[b]]\ \sigma_i = \texttt{tt}$ für $i \in [0..\tau-1]$ sowie $\mathcal{B}[[b]]\ \sigma_\tau = \texttt{ff}$ – dies gilt beispielsweise für die Schleife `while` $x \geq 0$ `do` $x := x - 1$.

2. die Schleife loopt lokal, d.h. im Schleifenrumpf:
   Nun gilt $\mathcal{B}[[b]]\ \sigma_i = \texttt{tt}$ für $i \in [0..\tau]$ und $\mathcal{S}_{ds}[[S]]\sigma_i = \sigma_{i+1}$ für $i < \tau$ sowie $\mathcal{S}_{ds}[[S]]\sigma_\tau = \bot$.

3. die Schleife loopt global, d.h. die Schelifenbedingung wird nie verletzt.
   In diesem Fall ergibt sich

Um den **kleinsten Fixpunkt** eines Funktionals zu definieren, benötigen wir zunächst einige Begriffe. Wir definieren eine Ordnungsrelation $\sqsubseteq$ auf $\mathcal{F}_{\texttt{STATE}}$ durch

$$g \sqsubseteq h \quad \text{genau dann, wenn für alle } \sigma, \sigma' \text{ gilt: } g(\sigma) = \sigma' \implies h(\sigma) = \sigma' \ ,$$

mit anderen Worten,der Definitionsbereich von g ist in dem von h enthalten und dort, wo beide Funktionen definiert sind, haben sie die gleichen Werte. Definiert man für eine Funktion $g : A \to B$ die Menge

$$\texttt{GRAPH}(g) \ := \ \{(x, g(x)) \mid x \in A,\ g(x) \in B\}$$

als die Menge aller Paare von Argument und Funktionswert, so läßt sich $g \sqsubseteq h$ auch schreiben als $\texttt{GRAPH}(g) \subseteq \texttt{GRAPH}(h)$.

4.2 Fixpunkt-Theorie

Wir beginnen mit einer allgemeinen Einführung in die Theorie partieller Ordnungen.

Definition 4.2 *Ein Tupel* $(D, \sqsubseteq)$ *bestehend aus einer Menge* D *und einer Relation* $\sqsubseteq$ *auf* D *heißt* **partiell geordnete Menge** *oder POS (für partial ordered set), falls die folgenden drei Eigenschaften erfüllt sind:*

- *Reflexivität:* $d \sqsubseteq d \; \forall d \in D$,
- *Transitivität:* $d_1 \sqsubseteq d_2 \; \wedge \; d_2 \sqsubseteq d_3 \implies d_1 \sqsubseteq d_3$,
- *Asymmetrie:* $d_1 \sqsubseteq d_2 \; \wedge \; d_2 \sqsubseteq d_1 \implies d_1 = d_2$.

Das **kleinstes Element** *einer POS, d.h. ein Element* d_0 *mit* $d_0 \sqsubseteq d$ *für alle* $d \in D$ *, ist eindeutig, falls es existiert.*

Ist K *eine Teilmenge von* D *, so heißt* $\bar{d}$ *eine* **obere Schranke von** $\boldsymbol{K}$ *, falls* $k \leq \bar{d}$ *für alle* $k \in K$ *gilt. Eine obere Schranke* $\bar{d}$ *heißt* **kleinste obere Schranke**, *falls für jede andere obere Schranke* d' *gilt* $\bar{d} \sqsubseteq d'$ *:*

Falls eine Menge K eine kleinste obere Schranke besitzt, so ist diese eindeutig und wir notieren diese als $\sqcup K$.

Das kleinste Element in $\mathcal{F}_{\texttt{STATE}}$ ist offensichtlich die nirgends definierte Funktion. Eine obere Schranke für eine beliebige Menge von Elementen aus einer POS existiert in der Regel nicht. Wir betrachten daher die folgende Einschränkung.

Definition 4.3 *Eine Teilmenge* $K \subseteq D$ *heißt* **Kette**, *falls sich jedes Paar aus* K *anordnen läßt, d.h. für alle* $k, k' \in K$ *gilt* $k \sqsubseteq k'$ *oder* $k' \sqsubseteq k$.
Dies bedeutet, daß sich die Elemente in K *linear anordnen lassen. Mit einer geeigneten Indizierung läßt sich* K *darstellen als eine Folge*

$$\ldots, \sqsubseteq k_{-2} \sqsubseteq k_{-1} \sqsubseteq k_0 \sqsubseteq k_1 \sqsubseteq k_2 \sqsubseteq \ldots \; ,$$

wobei links und rechts endlich oder unendliche viele Elemente vorkommen können.
Eine partiell geordnete Menge heißt **Ketten-vollständig**, *CCPOS, falls eine kleinste obere Schranke* $\sqcup K$ *existiert für jede Kette* K .
Falls jede beliebige Teilmenge eine kleinste obere Schranke besitzt, so nennen wir die partiell geordnete Menge **vollständig**.

Für die partiell geordnete Menge $\mathcal{F}_{\texttt{STATE}}$ läßt sich die obere Schranke für eine Kette K berechnen durch

$$\texttt{GRAPH}(\sqcup(K) \; = \; \bigcup \{\texttt{GRAPH}(k) \mid k \in K\} \; .$$

Definition 4.4 *Eine Abbildung* $f : D \to D'$ *zwischen partiell geordneten Mengen heißt* **monoton**, *falls für alle* $d_1, d_2 \in D$ *gilt:* $d_1 \sqsubseteq d_2 \implies f(d_1) \sqsubseteq' f(d_2)$.

Ist $K \subseteq D$ eine Kette, dann bildet für eine monotone Funktion $f : D \to D'$ auch $f(K) = \{f(k) \mid k \in K\}$ eine Kette in D'. Die Monotonie impliziert des weiteren

$$\sqcup f(K) \sqsubseteq' f(\sqcup K) ,$$

die Gleichheit der beiden Seiten gilt jedoch nicht zwangsläufig.

Definition 4.5 *Eine monotone Funktion heißt* **stetig**, *falls*

$$\sqcup f(K) = f(\sqcup K) .$$

Das folgende Beispiel illustriert diese Begriffe. Die Potenzmenge einer beliebigen abzählbaren Menge A mit der Mengeninklusion ist ein CCPOS. Betrachten wir speziell $A = \mathbb{N} \cup \{\infty\}$, wobei ∞ ein neues Element ist, und die Abbildung f definiert durch $f(B) = B$, falls $B \subseteq A$ eine endlich Menge ist, und $f(B) = B \cup \{\infty\}$ für unendliche B. Dies f ist zwar monoton, aber für die Kette K gegeben durch die Folge $k_0 = \{0\}$, $k_1 = \{0, 1\}$, $k_2 = \{0, 1, 2\}$, ... gilt

$$\sqcup f(K) = \sqcup K = \mathbb{N} , \qquad \text{aber} \quad f(\sqcup K) = f(\mathbb{N}) = \mathbb{N} \cup \{\infty\} .$$

Man rechnet leicht nach, daß die Menge der stetigen Funktionen abgeschlossen ist unter Komposition.

Theorem 4.1 *Sei* $\hat{F} : \mathcal{D} \to \mathcal{D}$ *stetig auf* $(D, \sqsubseteq)$ *mit kleinstem Element* $\perp$. *Dann ist der kleinste Fixpunkt von* F *gegeben durch*

$$\texttt{FIXP}\ \hat{\texttt{F}} = \sqcup\{\texttt{F}^{\texttt{n}}(\perp) \mid \texttt{n} \geq \texttt{0}\} .$$

Lemma 4.1 *Das Funktional* $\hat{F}$ *für die Definition der Semantik der* `while`*-Schleife ist monoton und stetig.*

Dies bedeutet, daß der Fixpunktoperator `FIXP` immer einen Wert liefert, die semantische Funktion $\mathcal{S}_{ds}$ ist daher wohl definiert und total.

Theorem 4.2 *Für jedes* `WHILE`*-Statement* S *gilt:* $\mathcal{S}_{ds}[[S]] = \mathcal{S}_{sos}[[S]]$.

Bei einer Erweiterung der `WHILE`-Sprache um Prozeduren läßt sich $\mathcal{S}_{ds}$ entsprechend verallgemeinern.

Erweiterung zur Behandlung von Ausnahmen (exceptions): die Syntax wird um die beiden Konstrukte

$$\texttt{raise}\ e \quad \text{und} \quad \texttt{begin}\ S_1\ \texttt{handle}\ e : S_2\ \texttt{end}$$

ergänzt.

5 Ausblicke

5.1 Statische Programm Analyse

Vorbedingungen, Nachbedingungen, Schleifeninvarianten

5.2 Axiomatische Programm-Verifikation

geeigneter logischer Kalkül: Hoares Programm-Logik oder Hilberts Deduktionskalkül

Partielle Korrektheit + Terminierung