Semantik

Skript zur Vorlesung
Semantik von Programmiersprachen
im WS 2003/04
an der Universität Paderborn

Überarbeitet im WS 2004/05

Ekkart Kindler

Version: 0.2.7 vom 18. Januar 2005

Vorwort

Dieses Skript ist im Rahmen der gleichnamige Vorlesung entstanden, die ich im WS 2003/04 an der Universität Paderborn gehalten habe. Ähnliche Vorlesungen – allerdings mit etwas anderen Schwerpunkten – habe ich bereits früher an verschiedenen Universitäten gehalten. Eine der Grundlagen dieser früheren Vorlesungen war das Buch von Glynn Winskel [11]. Die Vorlesung im WS 2003/04 habe ich zwar vollkommen neu überarbeitet und teilweise auch anders aufgebaut. Aber ihr Ursprung im Buch von Glynn Winskel [11] ist noch deutlich sichtbar. Darüber hinaus habe ich natürlich anderes Material benutzt, von dem ich besonders das Buch von Eike Best [2] hervorheben möchte.

Dieser Text ist allerdings noch weit davon entfernt, ein vollständiges Buch zum Thema „Semantik“ zu sein. Es ist an vielen Stellen noch unvollständig und enthält wahrscheinlich noch eine Menge Fehler. Viele interessante Bemerkungen und Querbezüge zu anderen Bereichen der Informatik (z. B. der Logik, der Programmverifikation, der Algebraischen Spezifikation) sind nur durch kurze Randbemerkungen angedeutet. Als Begleitmaterial zur Vorlesung ist es aber sicher schon recht hilfreich. Im Rahmen zukünftiger Vorlesungen werde ich dieses Skript weiter überarbeiten und langsam vervollständigen und Fehler beseitigen. Übungsaufgaben zu der Vorlesung und teilweise auch Musterlösungen sind über das WWW verfügbar.

Auch wenn dieses Skript sicher noch fehlerhaft ist, so enthält es schon deutlich weniger Fehler als im ersten Entwurf. Dies habe ich den Studierenden aus der Vorlesung zu verdanken, die mich auf Fehler hingewiesen haben. Auch Florian Klein hat sich eine Vorversion dieses Skriptes angesehen und mich auf diverse Fehler hingewiesen. Dafür möchte ich mich an dieser Stelle bedanken.

Paderborn, im Februar 2004,
Ekkart Kindler

Vorwort zur überarbeiteten Fassung

Im WS 2004/05 werde ich das Skript nochmals überarbeiten und dabei weitere Fehler und Unklarheiten beseitigen. Spezieller Dank geht an Matthias Tichy, der mich mit seinem annotierten Skript auf einige Fehler und Unklarheiten hingewiesen hat.

Paderborn, im WS 2004/05,
Ekkart Kindler

Inhaltsverzeichnis

Teil A

Vorlesung

Kapitel 1

Einführung

1 Der Begriff Semantik

Unter *Semantik* versteht man ursprünglich „die Lehre von der Bedeutung sprachlicher Zeichen“ [4] und Zeichenfolgen, und damit ein Teilgebiet der Linguistik. Oft versteht man unter Semantik auch die Bedeutung eines Wortes oder eines Satzes. In der Informatik versteht man unter Semantik insbesondere die Bedeutung eines Programms. In der Informatik versteht man dementsprechend unter Semantik die Lehre und Wissenschaft von der Bedeutung von Programmen oder allgemein der Bedeutung von syntaktischen Konstrukten der Informatik. Sie beschäftigt sich mit Techniken, die es erlauben, einem Programm oder einem syntaktischen Konstrukt einer bestimmten Sprache eine Bedeutung zuzuordnen.
Im Sprachgebrauch der Informatik versteht man unter „Semantik“ je nach Kontext verschiedene Dinge:

Semantik eines Programms: Eine einem konkreten Programm zugeordnete Bedeutung.

Semantik einer Programmiersprache: Eine Abbildung jedes syntaktisch korrekten Programms einer konkreten Programmiersprache auf dessen Bedeutung.

Semantik von Programmiersprachen: Die Techniken die man zur Definition der Semantik verschiedenartiger Programmiersprachen heranziehen kann; also das Teilgebiet der Informatik.

Die Vorlesung beschäftigt sich – wie der ausführliche Name zeigt – mit dem letzten Verständnis des Begriffs Semantik. Als Beispiele werden wir aber immer wieder die Semantik konkreter Programme und konkreter Programmiersprachen[1] betrachten. Nachfolgend betrachten wir diese unterschiedlichen Bedeutungen des Begriffs Semantik noch etwas ausführlicher anhand von Beispielen.

1.1 Semantik eines Programms

Wir betrachten dazu ein konkretes Programm[2] in einer pseudoprogrammiersprachlichen Notation.

Beispiel 1.1 (Die Fakultätsfunktion)
Wir betrachten das folgende Programm:

```
function fac(x:nat):nat
  if x = 0 then 1
           else x * fac(x−1)
```

Auch, wenn jeder Informatiker sofort sieht, daß dieses Programm die Fakultätsfunktion berechnet, handelt es sich zunächt nur um reine Syntax. Die Semantik dieses Programms formulieren wir in Mathematik:

$$\begin{aligned} f &: \mathbb{N} \to \mathbb{N} \\ n &\mapsto n! \end{aligned}$$

Man schreibt dann oft auch $[\![\texttt{fac}]\!] = f$ um auszudrücken, daß dem Programm fac die Semantik f zugeordnet wird.

> *In diesem Beispiel erscheint die Angabe einer Semantik noch reichlich überflüssig zu sein. Im Laufe der Vorlesung werden wir aber noch einige Feinheiten kennen lernen, die erst bei einer genauen Formulierung der Semantik erkennbar werden. Ein Beispiel für solche Feinheiten folgt sofort.*

Beispiel 1.2 (Eine Funktion höherer Ordnung)
Wir betrachten eine Funktion, die eine Funktion als Parameter besitzt:

[1] Meist sind dies sehr einfache Programmiersprachen, um uns nicht in den Details praktischer Programmiersprachen zu verlieren, sondern um uns auf die Techniken der Semantik zu konzentrieren.

[2] Der Einfachheit halber betrachten wir hier sogar nur eine Funktionsdeklaration.

```
function G( H: function(int):int , x:int):int
  H(x) - H(x)
```

Offensichtlich ist die Semantik dieses Programms eine Abbildung, die als Parameter eine partielle Abbildung h und eine Zahl n als Argumente nimmt und als Ergebnis $h(n)-h(n)$ ausgibt. Das Ergebnis ist immer 0. Die Semantik dieses Programms können wir also wie folgt formulieren:

$$\begin{aligned} &g : ((\mathbb{Z} \rightharpoonup \mathbb{Z}) \times \mathbb{Z}) \rightharpoonup \mathbb{Z} \\ &(h, n) \mapsto 0 \end{aligned}$$

Dabei steht $\rightharpoonup$ für eine (potentiell) partielle Abbildung, d. h. für eine Abbildung, die nicht (unbedingt) auf allen Argumenten definiert ist. Wir schreiben wieder $[\![\mathrm{G}]\!] = g$.
Diese Semantik g liefert für jedes Argument das Ergebnis 0. Die Frage ist nun, ob wir das bei diesem Programm auch wirklich so erwarten würden. Angenommen G bekommt eine Abbildung h und einen Wert n als Argumente, für die $h(n)$ nicht definiert ist, d. h. daß die Auswertung von h für Argument n nicht terminiert. Dann würde der Ausdruck H(x)−H(x) bei einer operationalen Auswertung nicht terminieren – also kein Resultat liefern. Für solche Argumente entspricht die obige Semantik also nicht ganz unserer Erwartung. Wir sollten g dementsprechend etwas anders definieren:

$$(h, n) \mapsto \begin{cases} 0 & \text{falls h(n) definiert ist} \\ \text{undef} & \text{sonst} \end{cases}$$

1.2 Semantik einer Programmiersprache

Im vorangegangen Abschnitt haben wir die Semantik eines bzw. zweier Programme kennen gelernt. Wenn wir die Semantik einer Programmiersprache definieren, legen wir damit eine (totale) Abbildung fest, die jedem *syntaktisch* korrekten Programm einer bestimmten Programmiersprache seine Semantik zuordnet. Es geht also um die Abbildung von „Syntax“ auf „Semantik“.
Die Notation für diese Abbildung haben wir bereits im vorangegangenen Abschnitt eingeführt: Die *Semantikklammern* $[\![.]\!]$. Letztendlich verbirgt sich dahinter genau eine Abbildung, die jedem Programm P (Syntax) einer Programmiersprache ein semantisches Objekt zuordnet (Semantik). Wie die semantischen Objekte aussehen und wie man eine solche Abbildung präzise definieren kann, ist Gegenstand dieser Vorlesung.

... Hier fehlt noch eine Graphik, die die Abbildung verdeutlicht.

> *Die Zuordnung einer Semantik zu einem Programm entspricht genau der Zuordnung zwischen der Repräsentation einer Information und der Information selbst: die Interpretation.*

1.3 Semantik von Programmiersprachen

Unter Semantik von Programmiersprachen versteht man die Summe aller Techniken zum Definieren von Semantiken konkreter Programmiersprachen und zum Argumentieren über diese Semantiken. Dabei werden verschiedene Konzepte – insbesondere die Induktion und die Fixpunkttheorie – implizit oder explizit immer wieder benutzt. Diese Konzepte werden wir im Laufe der Vorlesung identifizieren, formalisieren und Zusammenhänge zwischen ihnen herstellen.

2 Ansätze

Es gibt verschiedene Ansätze, wie man einem Programm eine Semantik zuordnen kann. In diesem einführenden Kapitel verschaffen wir uns zunächst einen Überblick über diese verschiedenen Ansätze.

2.1 Operationale Semantik

Zunächts betrachten wir die operationale Semantik für eine einfache imperative Sprache. Die operationale Semantik wird über das schrittweise Verhalten des betrachteten Programms definiert. Wir machen uns das anhand eines einfachem Beispielprogrammes deutlich:

Beispiel 1.3 (Eine Schleife)
Wir betrachten das folgende Programm c

$$\mathtt{c} \equiv \textbf{while } \mathrm{x} > 0 \textbf{ do } \mathrm{x} := \mathrm{x}-1 \textbf{ od}$$

und einen Startzustand $\sigma = [x/2]$ (in diesem Zustand hat die Variable x den

Wert 2). Nun arbeiten wir das Programm schrittweise ab:

$$\begin{aligned}\langle \mathtt{c}, [x/2]\rangle \rightarrow \quad & \langle \mathtt{x} := \mathtt{x} - \mathtt{1}; \mathtt{c}, [x/2]\rangle \rightarrow \\ & \langle \mathtt{c}, [x/1]\rangle \rightarrow \\ & \langle \mathtt{x} := \mathtt{x} - \mathtt{1}; \mathtt{c}, [x/1]\rangle \rightarrow \\ & \langle \mathtt{c}, [x/0]\rangle \rightarrow \\ & \langle \mathbf{skip}, [x/0]\rangle \not\rightarrow\end{aligned}$$

Wir beginnen dabei mit der Abarbeitung des Programms c im Zustand $[x/2]$. Zunächst wird die Schleifenbedingung x > 0 ausgewertet. Im betrachteten Zustand ist diese Bedingung wahr. Dementsprechend muß der Schleifenrumpf x:= x $-$ 1 einmal ausgeführt werden und danach die Schleife c erneut ausgeführt werden. Deshalb fahren wir im nächsten Schritt mit der Abarbeitung des Programms x:= x $-$ 1; c fort; da die Auswertung der Bedingung den Zustand nicht verändert, setzten wir die Berechnung im gleichen Zustand fort. Nun ist x:= x $-$ 1 die erste Anweisung; die Abarbeitung führt zum Zustand $[x/1]$. Danach müssen wir nur noch die Anweisung c ausführen, also die Schleife von vorne ausführen. Nach zwei weiteren Schritten müssen wir dann die Schleife im Zustand $[x/0]$ ausführen. Dazu wird wieder die Schleifenbedingung ausgewertet, was in diesem Zustand das Ergebnis falsch liefert. Deshalb wird die Schleife beendet und mit der Abarbeitung des Programmes nach der Schleife fortgefahren. Da dort nichts steht, schreiben wir **skip** für das leere Programm, für das natürlich keine weiteren Schritte mehr ausgeführt werden können. Ingesamt endet das Programm also im Zustand $[x/0]$, wenn es im Zustand $[x/2]$ gestartet wird.

Die einzelnen Übergänge müssen natürlich für alle Konstrukte der Programmiersprache definiert werden. Wie dies geht, werden wir später sehen. Das wichtige bei der operationalen Semantik ist, daß wir ein gegebenes Programm c ausgehend von einem Anfangszustand σ schrittweise „simulieren“ bzw. „interpretieren“ und – wenn das Programm terminiert – irgendwann das Ergebnis erhalten.

2.2 Mathematische (denonationale) Semantik

Ein Problem bei der Definition der operationalen Semantik ist, daß wir streng genommen nicht dem Programm eine Semantik zuordnen, sondern immer nur einem Programm in einem Zustand. Schöner und eleganter wäre es, dem Programm insgesamt eine Semantik zuzuordnen. Wenn man das ordentlich

machen will, benötigt man etwas mathematisches Handwerkszeug, was wir uns erst im Laufe der Vorlesung erarbeiten werden. Deshalb geben wir hier nur das Endergebnis für das Beispiel 1.3 an. Da das Programm nur eine Variable besitzt, können wir den Zustand des Programms als eine ganze Zahl (den Wert der Variablen x) darstellen. Die mathematische Semantik des Programms ist eine partielle Abbildung, die jedem Anfangszustand den zugehörigen Endzustand des Programms zuordnet, wenn das Programm terminiert; ansonsten ist die Funktion für den Anfangszustand undefiniert (in unserem Beispiel terminiert das Programm aber immer). Die mathematische Semantik können wir also wie folgt definieren:

$$[\![c]\!] : \mathbb{Z} \rightharpoonup \mathbb{Z}$$

$$n \mapsto \begin{cases} 0 & \text{für } n \geq 0 \\ n & \text{sonst} \end{cases}$$

Nun haben wir zwei Semantiken für das Programm `c` definiert: die operationale und die mathematische Semantik. Natürlich sollten diese beiden Semantiken etwas miteinander zu tun haben. Tatsächlich sind die Techniken zum Formulieren und Beweisen von Beziehungen zwischen verschiedenen Semantiken auch Gegenstand des Gebietes Semantik und der Vorlesung. Für die beiden Semantiken unserer Programmiersprache sollte für alle Programme `c` gelten:

$$[\![c]\!](n) = m \qquad \text{gdw.} \qquad \langle \mathtt{c}, [x/n] \rangle \rightarrow \ldots \rightarrow \langle \textbf{skip}, [x/m] \rangle$$

Daß das stimmt, kann man sich für dieses Beispiel leicht überlegen. Allgemeine Techniken zum Beweis derartiger Eigenschaften werden wir erst später kennen lernen.

2.3 Axiomatische Semantik

Als letzte Beispiel betrachten wir die axiomatische Semantik. Diese axiomatisiert Eigenschaften von Programmen, wobei die Eigenschaft durch Vor- und Nachbedingungen formuliert sind, die auch. *Zusicherungen* genannt werden. Dazu betrachten wir wieder das Programm `c` aus Beispiel 1.3. Wenn der Wert der Variablen vor Ausführung des Programms **while** x $>$ 0 **do** x:= x $-$ 1 **od** größer als 0 ist, dann ist bei Terminierung des Programmes der Wert der Variablen x der Wert 0 (wir setzen voraus, daß x eine Variable vom Typ integer

ist). Als Zusicherung notieren wir diese Eigenschaft wie folgt:

$$\{x \geq 1\} \textbf{ while } \mathrm{x} > 0 \textbf{ do } \mathrm{x} := \mathrm{x} - 1 \textbf{ od } \{x = 0\}$$

Dabei ist $\{x \geq 1\}$ die Vorbedingung, unter der das Programm am Ende die Nachbedingung $\{x = 0\}$ erfüllt.
Die axiomatische Semantik gibt nun Regeln an, mit deren Hilfe alle gültigen Zusicherungen bewiesen werden können. Sie beschreibt damit implizit die Bedeutung eines Programmes mit Hilfe seiner Eigenschaften und stellt somit den Zusammenhang zur Programmverifikation her. Natürlich gibt es sehr viele gültige Zusicherungen für ein Programm, im allgemeinen unendlich viele. Beispielsweise gilt für unser Beispiel auch die Zusicherung

$$\{x = n \wedge x < 0\} \textbf{ while } \mathrm{x} > 0 \textbf{ do } \mathrm{x} := \mathrm{x} - 1 \textbf{ od } \{x = n\}$$

d. h. wenn der Wert von x vor Ausführung des Programms n ist und außerdem kleiner als 0 ist, dann ist der Wert von x am Ende auch n.
Auch diese Semantik hat natürlich einen Bezug zur operational und zur mathematischen Semantik. Im wesentlichen ist dieser Bezug die Formalisierung der Bedeutung von Zusicherungen mit Hilfe der operationalen oder der mathematischen Semantik. Dies werden wir aber erst später präzisieren.

2.4 Diskussion der Ansätze

Neben den drei oben vorgestellten Ansätzen, gibt es noch eine Reihe weiterer Möglichkeiten einer Programmiersprache eine Semantik zuzuordnen. Beispielsweise kann man einer Programmiersprache auch eine Semantik zuordnen, indem man eine Übersetzung in eine andere Programmiersprache mit bereits definierter Semantik angibt (*Übersetzersemantik*). Der Grund für die Existenz der verschiedenen Ansätze ist, daß die Definition einer Semantik verschiedenen Zwecken dienen kann:

- Präzisierung der informellen Semantik einer Programmiersprache
- Verständnis einer neuen Programmiersprache
- Konstruktion von Compilern
- Untersuchung der grundlegenden Konstrukte einer Programmiersprache

- Basis für die Verifikation
- Grundlegendes Verständnis für programmiersprachliche Konstrukte
- ...

Je nach dem Zweck, der mit der Definition einer Semantik verfolgt wird, sind bestimmte Ansätze besser oder schlechter geeignet. Beispielsweise ist die Axiomatische Semantik für die Verifikation besonders gut geeignet.

3 Das Dilemma der Semantik

Bei der Definition einer Semantik wird einem „Stück Syntax“ (konkret einem Programm) ein „Stück Semantik“ (konkret eine Abbildung) zugeordnet. Bei genauer Betrachtung steht aber auf der „semantischen Seite“ auch wieder „nur“ Syntax. Beispielsweise haben wir für unser erstes Beispiel (die Fakultätsfunktion) die Semantik wie folgt definiert:

$$\begin{aligned} f &: \mathbb{N} \to \mathbb{N} \\ n &\mapsto n! \end{aligned}$$

Zur Formulierung der mathematischen Abbildung haben wir aber wieder Symbole, also Syntax, benutzt: $\mathbb{N}, \to, \mapsto$ und !. Dies sind zwar nicht die Symbole aus der Programmiersprache, sondern Symbole aus der Mathematik, die uns bereits in der Schule vertraut gemacht wurden und deren Bedeutung wir „kennen“. Streng genommen müßten wir aber zunächst die Semantik dieser Symbole definieren, bevor wir sie benutzen können, um eine Semantik für eine Programmiersprache zu definieren. Wenn wir nun mit einer Formalisierung dieser Symbole beginnen, werden wir schnell feststellen, daß wir mit dem Formalisieren nie fertig werden, denn wir werden immer neue Symbole einführen, deren Bedeutung wir dann wieder definieren müssen. Letztendlich müssen wir immer Symbole benutzen, um Sachverhalte zu formulieren. Tatsächlich ist dies auch kein spezielles Problem der Semantik, sondern ein fundamentales Problem der Mathematik und der Philosophie. Und auch dort läßt sich dieses Problem nicht wirklich lösen.
Die Lösung im Rahmen des Gebietes der Semantik besteht nun darin, daß wir davon ausgehen, daß wir ein bestimmtes Gebiet der Mathematik so gut kennen und ein gemeinsames Verständnis darüber besitzen, daß es nicht nötig ist,

es weiter zu formalisieren. Man sagt, daß wir von diesem Gebiet eine gemeinsame *Pragmatik* besitzen. Für uns ist die gemeinsame Pragmatik im wesentlichen die „Schulmathematik“. Der Rückzug auf eine gemeinsame Pragmatik ist zwar keine wirkliche Lösung des oben beschriebenen Problems, aber eine sehr „pragmatische“ Lösung – insbesondere da wir wissen, daß wir das Problem prinzipiell nicht lösen können. Wir sollten uns bei der Definition von Semantiken immer bewußt machen, daß das Zielpublikum, für das wir eine Semantik formulieren, die gleiche Pragmatik für die von uns benutzten Symbole besitzen muß – ansonsten ist die Definition der Semantik bedeungslos und damit sinnlos.

4 Inhalt der Vorlesung

In der Vorlesung werden wir uns mit den verschieden Ansätzen zur Definition von Semantiken einer Programmiersprache beschäftigen und die grundlegenden Techniken dazu kennen lernen. Dabei spielen induktive Definitionen und Beweise sowie Fixpunkte eine zentrale Rolle. Wir werden sogar feststellen, daß beide Konzepte sehr eng zusammenhängen. Insgesamt orientieren wir uns dabei sehr stark am Buch von Glynn Winskel [11].
Der Schwerpunkt der Vorlesung liegt dabei auf den Konzepten zur Definition von Semantiken. Die konkret definierten Semantiken dienen nur der Veranschaulichung und Einübung dieser Techniken. Die Programmiersprachen, die wir dazu betrachten, sind sehr minimalistisch, da die Definition der Semantik von realistischen Programmiersprachen sehr aufwendig ist und die vielen technischen Details einer realistischen Programmiersprache den Blick auf die wesentliche Semantischen Konzepte verstellen. Am Ende der Vorlesung sind wir aber prinzipiell in der Lage auch Semantiken für realistische Programmiersprachen zu formulieren. Ein Beispiel für die Definition einer Semantik einer realistischen Programmiersprache findet sich in dem Buch von Elfriede Fehr [5].

Kapitel 2

Grundlegende Begriffe und Notationen

In diesem Kapitel führen wir Begriffe und Konzepte ein, die als bekannt vorausgesetzt werden. Sie stellen also die gemeinsame *Pragmatik* dar, die wir als Ausweg aus dem Dilemma der Semantik benötigen. Die Begriffe selbst sollten aus der Schule oder spätestens aus den Vorlesungen des Grundstudiums bekannt sein. Da jedoch die Notationen für bestimmte Konzepte nicht „normiert“ sind, legen wir im folgenden die von uns benutzte Notation fest. Nebenbei frischen wir die Kenntnisse über die zugrundeliegenden Begriffe und Konzepte auf.

1 Mengen

Der fundamentalste Begriff, auf dem wir aufbauen, ist der Begriff der *Menge*. Tatsächlich würde eine fundierte Einführung dieses Begriffes eine eigene Vorlesung erfordern. Für uns genügt aber im wesentlichen das naive Verständnis des Mengenbegriffes, der bereits in der Schule vermittelt wurde: Eine Zusammenfassung oder Ansammlung von *Elementen*. Dem interessierten Leser sei jedoch die Lektüre eines Buches zur Mengenlehre empfohlen (z. B. [6]).
Wenn ein Element x zu einer Menge X gehört, schreiben wir dafür $x \in X$; wenn x nicht zu der Menge X gehört, schreiben wir $x \notin X$.
Einige Mengen haben eine besondere Bedeutung, so daß wir eine eigne Bezeichnung für diese Mengen einführen:

- $\emptyset$ bezeichnet die *leere Menge*, d. h. die Menge, die kein Element enthält.

- $\mathbb{N}$ bezeichnet die Menge aller *natürlichen Zahlen* (inkl. 0):
  $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.
- $\mathbb{Z}$ bezeichnet die Menge aller *ganzen Zahlen*:
  $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.
- $\mathbb{B}$ bezeichnet die Menge der *Wahrheitswerte*:
  $\mathbb{B} = \{true, false\}$.

Oft wird eine Menge X darüber definiert, daß man die Eigenschaft $P(x)$ aller ihre Elemente x angibt. Das ist insbesondere bei unendlichen Mengen erforderlich. Dafür benutzt man die *Mengenkomprehension*: $X = \{x \mid P(x)\}$, wobei P eine Prädikat über bzw. eine Eigenschaft von Objekten bezeichnet.
Eine Menge X heißt *Teilmenge* einer Menge Y, wenn für jedes Element $x \in X$ auch gilt $x \in Y$. Wir schreiben dafür $X \subseteq Y$. Die Menge X heißt *echte Teilmenge* von Y, wenn wenigstens ein Element $y \in Y$ nicht in X vorkommt (d. h. $y \notin X$). Wir schreiben dann $X \subset Y$.
Auf Mengen sind verschiedene Operationen definiert. Für zwei Mengen X und Y bezeichnen

- $X \cup Y$ die *Vereinigung* der Elemente der beiden Mengen,
  d. h. $X \cup Y = \{z \mid z \in X \text{ oder } z \in Y\}$.
- $X \cap Y$ den *Durchschnitt* der Elemente der beiden Mengen,
  d. h. $X \cap Y = \{z \mid z \in X \text{ und } z \in Y\}$.
- $X \setminus Y$ die *Differenz* der Elemente der beiden Mengen,
  d. h. $X \setminus Y = \{x \mid x \in X \text{ und } x \notin Y\}$.
- $X \times Y$ das *Produkt* der beiden Mengen,
  d. h. $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$.

Mit $|X|$ bezeichnen wir die *Kardinalität* oder die Mächtigkeit der Menge. Für endliche Mengen ist das die Anzahl der Elemente der Menge. Eine unendliche Menge hat aber nicht die Kardinalität „unendlich“ oder ∞! Denn es gibt verschiedene unendliche Kardinalitäten. Beispielsweise haben die Menge der natürlichen Zahlen $\mathbb{N}$ und die Menge der reellen Zahlen $\mathbb{R}$ verschiedene Kardinalität, genauer $|\mathbb{R}| > |\mathbb{N}|$. Allerdings kann eine echte Teilmenge X einer Menge Y dieselbe Kardinalität haben wie Y. Beispielsweise gilt $|\mathbb{N}| = |\mathbb{Z}|$.

In der Informatik ist die Kardinalität der Menge der natürlichen Zahlen von besonderer Bedeutung und wir bezeichnen sie mit ω. Mengen mit dieser Kardinalität werden *abzählbar* genannt. Mengen mit echt größerer Kardinalität werden *überabzählbar* genannt.
Für eine Menge X bezeichnet 2^X die Menge aller Teilmengen von X, d. h. $2^X = \{Y \mid Y \subseteq X\}$. Wir nennen 2^X auch die *Potenzmenge* von X.

2 Relationen, Ordnungen und Äquivalenzen

Für zwei Mengen X und Y nennen wir eine Teilmenge $R \subseteq X \times Y$ eine *Relation* über X und Y. Wir betrachten meist Relationen mit $X = Y$. In diesem Falle nennen wir R eine (*binäre*) *Relation* über X. Für $(x, y) \in R$ schreiben wir dann auch kurz $x\ R\ y$. In vielen Fällen wird die Relation auch durch einen Pfeil $\rightarrow$ bezeichnet und wir schreiben dann $x \rightarrow y$.
Eine binäre Relation R über X heißt

- *reflexiv*, wenn für jedes $x \in X$ gilt $x\ R\ x$,
- *irreflexiv*, wenn für kein $x \in X$ gilt $x\ R\ x$,
- *transitiv*, wenn für alle $x, y, z \in X$ mit $x\ R\ y$ und $y\ R\ z$ auch $x\ R\ z$ gilt,
- *symmetrisch*, wenn für alle $x, y \in X$ mit $x\ R\ y$ auch $y\ R\ x$ gilt,
- *antisymmetrisch*, wenn für alle $x, y \in X$ aus $x\ R\ y$ und $y\ R\ x$ folgt $x = y$ und sie heißt
- *konnex*, wenn für alle $x, y \in X$ wenigstens eine der folgenden Bedingungen gilt: $x = y$ oder $x\ R\ y$ oder $y\ R\ x$.

Eine reflexive, transitive und antisymmetrische Relation R nennen wir eine *reflexive Ordnung*. Für reflexive Ordnungen benutzen wir meist Symbole, die „eine Richtung besitzen“ und die Gleichheit enthalten: $\leq, \preceq, \subseteq, \sqsubseteq$ etc. Eine reflexive Ordnung heißt *total* oder *linear*, wenn sie konnex ist, d. h. wenn zwei beliebige Elemente immer in der einen oder anderen Richtung geordnet sind.

Achtung! Die meisten Ordnungen, die wir kennen, sind linear. Deshalb ist man schnell geneigt, diese Eigenschaft allen Ordnungen zu unterstellen. Es

gibt aber Ordnungen, die nicht linear sind. Beispielsweise ist die Teilmengenbeziehung auf Mengen $\subseteq$ nicht linear. Der Deutlichkeit halber nennt man solche Ordnungen dann oft partielle Ordnungen.

Es gibt noch eine zweite Definition von Ordnungen, bei der explizit die Gleichheit ausgeschlossen wird: Eine irreflexive und transitive Relation R nennen wir eine *irreflexive Ordnung*. Für solche Ordnungen benutzt man ebenfalls Symbole, die „eine Richtung besitzen“ aber keine Gleichheit enthalten: $<, \prec, \subset, \sqsubset$ etc. Es ist leicht, eine gegebene reflexive Ordnung in eine irreflexive Ordnung umzuwandeln und umgekehrt. Man muß dazu nur alle Paare (x, x) entfernen bzw. hinzufügen.
In der Vorlesung geht meist aus dem Kontext hervor, ob wir über eine reflexive oder irreflexive Ordnung reden. Wir reden deshalb oft nur über Ordnungen, ohne explizit dazu zu sagen, welche Variante wir meinen. Manchmal wandeln wir auch implizit eine reflexive Ordnung in eine irreflexive Ordnung um, wenn das zweckmäßiger ist. Dies entspricht dem Übergang von $\leq$ zu $<$.
Eine irreflexive Ordnung $\prec$ über X heißt *wohlgegründet*, wenn es keine unendlich absteigende Kette $x_1 \succ x_2 \succ x_3 \succ \ldots$ von Elementen $x_i \in X$ gibt.

Nebenbei wird hier ein beliebter Trick mit den gerichteten Symbolen für Ordnungen eingeführt: Wir schreiben $x_i \succ x_{i+1}$ anstelle von $x_{i+1} \prec x_i$, d. h. wir drehen die Symbole um, wenn uns das zweckmäßiger erscheint.

Eine Ordnung $\preceq$ über X wird häufig auch als Paar $(X, \preceq)$ notiert. Für eine Teilmenge $Y \subseteq X$ definieren wir nun einige weitere Begriffe. Ein Element $x \in Y$ heißt *minimal* (in Y bzgl. $\preceq$), wenn kein Element $y \in Y$ mit $y \preceq x$ und $x \neq y$ existiert, d. h. wenn für x kein echter Vorgänger in Y existiert. Ein Element $x \in Y$ heißt das *kleinste Element von* Y (bzgl. $\preceq$), wenn für alle $y \in Y$ gilt $x \preceq y$, d. h. x ist kleiner (oder gleich) als alle anderen Elemente von Y. Symmetrisch kann man die *maximalen* und das *größte Element* einer Menge bzgl. einer Ordnung definieren.

Achtung! Die Begriffe des minimalen und des kleinsten Elementes werden oft verwechselt, weil sie für viele uns vertraute Ordnungen zusammenfallen. Wir müssen diese beiden Begriffe aber sorgfältig auseinander halten. Denn eine Menge kann bezüglich einer Ordnung nur ein kleinstes Element besitzen (es ist also eindeutig, wenn es existiert). Dagegen kann eine Menge mehrere minimale Elemente besitzen. Dieser Unterschied wird später noch sehr wichtig werden.

Für eine reflexive Ordnung $(X, \preceq)$ und eine Teilmenge $Y \subseteq X$ heißt ein $x \in X$ eine *untere Schranke* von Y, wenn für jedes $y \in Y$ gilt $x \preceq y$. Wenn die Menge der unteren Schranken von Y ein größtes Element besitzt, dann heißt dieses die *größte untere Schranke* von Y oder auch das *Infimum* von X. Das Infimum von Y wird dann auch mit $\bigwedge Y$ bezeichnet. Entsprechend heißt ein Element $x \in X$ eine *obere Schranke* von Y, wenn für jedes $y \in Y$ gilt $y \preceq x$. Die kleinste obere Schranke von Y heißt auch das *Supremum* von Y und wird mit $\bigvee Y$ bezeichnet.
Wenn für eine Ordnung $(X, \preceq)$ für jede Teilmenge $Y \subseteq X$ das Infimum existiert, dann nennen wir diese Ordnung auch einen *vollständigen Verband*.

> *Für einen vollständigen Verband wird nur gefordert, daß „alle Infima" existieren. Man kann aber zeigen, daß in einem vollständigen Verband auch für jede Menge Y das Supremum existiert. In einem vollständigen Verband existieren also „alle Infima und Suprema„.*

Man kann jede Relation R transitiv machen, indem man alle transitiven Abhängigkeiten zur Relation hinzufügt. Diese Relation bezeichnen wir dann mit R^+ und wird auch die *transitive Hülle* von R genannt. Intuitiv ist sofort klar, was die transitive Hülle ist. Mathematisch gibt es verschiedene Techniken, die transitive Hülle zu definieren. Eine Möglichkeit dazu ist, sie als die kleinste transitive Relation zu definieren, die R umfaßt (dazu muß man natürlich zeigen, daß für jedes R diese Relation existiert). Ganz analog ist der Begriff der *reflexiv-transitiven Hülle* einer Relation R definiert. Es ist die kleinste reflexive und transitive Relation, die R umfaßt. Die reflexiv-transitive Hülle von R wird mit R^* bezeichnet. Zusätzlich zu R^+ kommen noch alle Paare (x, x) zu R^* hinzu (wg. der Reflexivität).
Eine Relation R heißt *Äquivalenzrelation* oder kurz *Äquivalenz*, wenn sie reflexiv, transitiv und symmetrisch ist.

3 Abbildungen

Eine Relation f über X und Y, für die für jedes $x \in X$ genau ein y mit $(x, y) \in f$ existiert, heißt *totale Abbildung* von X nach Y. Eine Abbildung f ordnet also jedem Element $x \in X$ eindeutig ein Element $y \in Y$ zu. Wir schreiben dafür auch $f(x) = y$.
Die Menge aller Abbildungen von X nach Y bezeichnen wir mit $X \to Y$ und an Stelle $f \in (X \to Y)$ von schreiben wir wie üblich $f : X \to Y$.

Eine Relation f über X und Y, für die für jedes $x \in X$ höchstens ein $y \in Y$ mit $(x, y) \in f$ existiert heißt *partielle Abbildung*. Auch hier schreiben wir $f(x) = y$. Der Unterschied zu den totalen Abbildungen ist, daß der Wert $f(x)$ nicht für jedes $x \in X$ definiert ist; wir schreiben dafür auch $f(x) = \mathit{undef}$. Wenn für alle $x \in X$ gilt $f(x) = \mathit{undef}$, nennen wir f die überall undefinierte Abbildung; wir bezeichnen diese Abbildung auch mit Ω.
Die Menge aller partiellen Abbildungen von X nach Y bezeichnen wir mit $X \rightharpoonup Y$ und wir schreiben auch $f : X \rightharpoonup Y$ für $f \in (X \rightharpoonup Y)$.

> *Die Definition der partiellen Abbildungen fordert nicht, daß es ein Element x mit $f(x) =$* undef *geben muß. Eine partielle Abbildung kann also total sein. Tatsächlich ist jede totale Abbildung eine partielle Abbildung im Sinne der Definition. Das können wir auch als Inklusion formulieren: $(X \to Y) \subseteq (X \rightharpoonup Y) \subseteq 2^{(X \times Y)}$.*
>
> *Um diesen sprachlichen Widerspruch etwas abzufedern, sprechen wir im folgenden meist von Abbildungen, wenn wir totale Abbildungen meinen und explizit von partiellen Abbildungen, wenn wir eine (potentiell) partielle Abbildung meinen.*

Für zwei partielle Abbildungen $f : X \rightharpoonup Y$ und $g : Y \rightharpoonup Z$ bezeichnet $g \circ f : X \rightharpoonup Z$ ebenfalls eine partielle Abbildung, die wie folgt definiert ist $(g \circ f)(x) = g(f(x))$. Die Abbildung $g \circ f$ heißt die *Funktionskomposition* von f und g.
Eine Abbildung $f : X \to Y$ heißt

- *injektiv*, wenn für alle $x, y \in X$ aus $f(x) = f(y)$ folgt $x = y$, sie heißt
- *surjektiv*, wenn für jedes $y \in Y$ ein $x \in X$ mit $f(x) = y$ existiert, und sie heißt
- *bijektiv*, wenn sie injektiv und surjektiv ist.

Eine totale Abbildung kann man definieren, indem man für jedes $x \in X$ den Wert $f(x) = e$ angibt, wobei e ein Ausdruck ist, in dem x als freie Variable vorkommt und der zu einem Wert aus Y ausgewertet wird. Oft schreibt man dafür auch $x \mapsto e$ (vgl. Beispiele in Kapitel 1). Dadurch ist die Abbildung punktweise (bzw. elementweise) definiert. Diese *punktweise* Definition ist aber nicht sehr elegant, da wir die Abbildung nicht am Stück definieren[1].

[1] Darüber werden wir später bei der Einführung der mathematischen Semantik ausführlicher sprechen.

Eleganter wäre eine Definition der Form $f = \ldots$, die die Abbildung insgesamt festlegt. Dazu benutzen wir die Notation des *Lambda-Kalküls*. Wir schreiben $f = \lambda\, x \in X\ .\ e$, wobei e ein Ausdruck ist, der zu einem Wert aus Y ausgewertet wird. Mit dieser Notation können wir beispielsweise die Quadrierung wie folgt definieren:

$$f = \lambda\, x \in \mathbb{Z}\ .\ x * x$$

Der Lambda-Operator λ dient dazu, den Definitionsbereich der Abbildung zu bennenen und eine Bezeichnung festzulegen, mit der man im Ausdruck auf den aktuellen Parameter der Abbildung Bezug nehmen kann. Den Wert der Abbildung für einen konkreten Parameter können wir dann wie folgt ausrechnen:

$$f(7) = (\lambda\, x \in \mathbb{Z}\ .\ x * x)(7) = 7 * 7 = 49$$

> *$\lambda\, x \in X\ .\ e$ entspricht in Programmiersprachen dem Konzept der namenlosen Funktionen oder dem Konzept der namenlosen Klassen in Java. Wir können eine Abbildung definieren, ohne sie zu benennen. Erst durch die Gleichung $f = \lambda\, x \in X\ .\ e$ ordnen wir der Abbildung den Namen f zu.*

...

`Evtl. wird dieses Kapitel später um weitere Begriffe erweitert.`

Kapitel 3

Operationale Semantik

In diesem Kapitel werden wir eine Technik zur Definition von operationalen Semantiken kennenlernen. Diese Technik wird am Beispiel der Semantik für eine einfache imperativen Programmiersprache IMP vorgeführt. Dazu führen wir zunächst die Syntax dieser Sprache ein. Danach definieren wir der Reihe nach die Semantik der verschiedenen programmiersprachlichen Konstrukte: arithmetische Ausdrücke, boolesche Ausdrücke und Anweisungen. Am Ende werden wir dann verschiedene Varianten der Semantik diskutieren.
Insgesamt werden wir dabei eine Technik zur Definition einer operationalen Semantik kennenlernen und zur Argumentation über diese.

1 Die Programmiersprache IMP

In diesem Abschnitt definieren wir die Syntax der einfachen imperativen Programmiersprache IMP. Diese Programmiersprache enthält die üblichen programmiersprachlichen Konstrukte Sequenz, bedingte Anweisung und Schleife und die Zuweisung eines Wertes an eine Variable. Bevor wir die Syntax formal definieren, betrachten wir ein Beispiel:

Beispiel 3.1 (Euklids Algorithmus)
Das folgende Programm berechnet den größten gemeinsamen Teile (ggT) zweier positiver ganzer Zahlen x und y:

```
if (1 < x)∧(1 < y) then
  while ¬(x = y) do
    if x ≤ y then y:= y − x else x:= x − y
```

```
else
⌜ x:= 0; y:= 0 ⌟
```

1.1 Syntax

In diesem Programm kommen *Konstanten* und *Variablen* vor, aus denen *arithmetische* und *boolesche Ausdrücke* gebildet werden. Die booleschen Ausdrücke werden ihrerseits benutzt, um daraus die bedingte Anweisung und die Schleife zu konstruieren. Dieser Konstruktion liegt das Prinzip der induktiven Definition zugrunde.
Der Einfachheit halber lassen wir in unserer Programmiersprache nur Variablen der Sorte `integer` bzw. der ganzen Zahlen zu. Für Konstanten und Variablen werden wir uns nicht einmal die Mühe machen, deren konkrete Syntax zu definieren. Wir gehen einfach davon aus, daß wir diese Konstrukte irgendwie syntaktisch ausdrücken können. Konkret heißt das für die Sprache IMP, daß wir die ganzen Zahlen $\mathbb{Z}$ und die Menge der Wahrheitswerte $\mathbb{B}$ „irgendwie“ syntaktisch ausdrücken können. Die Menge der *Programmvariablen* bezeichnen wir mit $\mathbb{V}$. In unserem obigen Programm haben wir x und y als Programmvariablen benutzt. Aber wir werden je nach Bedarf auch weitere Bezeichnungen einsetzen, z. B. ggt oder result . Damit wir immer genug Programmvariablen zur Vefügung haben, muß $\mathbb{V}$ nur genügend groß sein, nämlich abzählbar.
Für die weiteren Konstrukte geben wir später eine Syntax in Backus-Naur-Form (BNF) an. Damit wir nicht jedes mal sagen müssen, für welche syntaktische Menge bzw. Kategorie ein Symbol steht, legen wir für alle syntaktischen Kategorien bestimmte Symbole fest. Dabei steht *Aexp* für die *syntaktische Menge* bzw. Kategorie der arithmetischen Ausdrücke, *Bexp* für die booleschen Ausdrücke und *Com* für die Anweisungen (bzw. Programme) der Programmiersprache IMP.

Kategorie	Symbole	Varianten
$\mathbb{Z}$	$n,\ m$	$n_0, n_1, n_2, \ldots, n', m', \ldots$
$\mathbb{B}$	t	$t_0, t_1, t_2, \ldots, t', t'', \ldots$
$\mathbb{V}$	$u,\ v$	$v_0, v_1, v_2, \ldots, u', v', \ldots$
Aexp	a	$a_0, a_1, a_2, \ldots, a', a'', \ldots$
Bexp	b	$b_0, b_1, b_2, \ldots, b', b'', \ldots$
Com	c	$c_0, c_1, c_2, \ldots, c', c'', \ldots$

Die Syntax für *Aexp*, *Bexp* und *Com* werden wir nachfolgend definieren. Dabei nutzen wir aus, daß bestimmte Symbole für ein syntaktisches Objekt einer bestimmten Kategorie gehören. Beispielsweise stehen a_0 und a_1 für arithmetische Ausdrücke. Im Gegensatz zu der sonst üblichen Form der BNF unterscheiden wir durch einen Index das mehrfache Auftreten von Objekten derselben syntaktischen Kategorie. Dies ermöglicht es uns, später bei der Definition der Semantik direkt auf die richtigen Objekte zu verweisen.

$$\begin{array}{lll} \textit{Aexp}: & a ::= & n \mid v \mid a_0{+}a_1 \mid a_0{-}a_1 \mid a_0{*}a_1 \\ \\ \textit{Bexp}: & b ::= & t \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg\, b_0 \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \\ \\ \textit{Com}: & c ::= & \textbf{skip} \mid v := a_0 \mid c_0 \;;\; c_1 \mid \\ & & \textbf{if } b_0 \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b_0 \textbf{ do } c_0 \end{array}$$

Beispiel 3.2 (Beispiele für syntaktische Konstrukte)
Wir betrachten nun einige Beispiele für syntaktische Objekte der verschiedenen Kategorien:

1. Arithmetische Ausdrücke aus *Aexp*:
   - 3 + 5 und 5 + 3
   - 4711 und 04711
   - x − 7 + 3 (Auf ein Problem mit diesem Ausdruck werden wir weiter unten noch eingehen)
   - x ∗ y
2. Boolesche Ausdrücke aus *Bexp*:
   - true und false
   - 3 ≤7

     7 ≥ 3 und 7 > 3 sind gemäß unserer Definition keine booleschen Ausdrücke. Wir werden sie später – wenn wir Syntax nicht mehr ganz so ernst nehmen – in Beispielen aber als Abkürzung zulassen. Beispielsweise steht dann x > y für (y ≤ x) ∧ ¬ (x = y).
   - ¬ 3 ≤ x
   - x = 8 ∧y ≤27

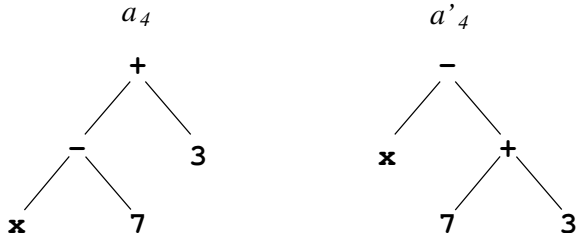
3. Anweisungen aus Com: Das Programm aus Beispiel 3.1 ist eine korrekte Anweisung. Auch das folgende Programm ist eine korrekte Anweisung:

```
z := 1;
y := 1;
while y ≤ x do
   z := y * x;
   y := y + 1
```

   Aber auch über diese Anweisung werden wir noch diskutieren müssen.

1.2 Abstrakte und konkrete Syntax

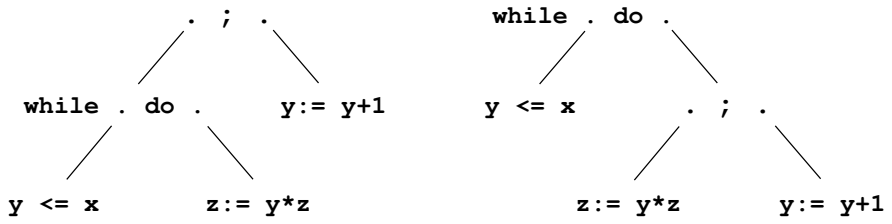
Wie schon angedeutet gibt es mit der Definition unsere Syntax noch ein Problem. Dazu betrachten wir nochmals den arithmetischen Ausdruck $x - 7 + 3$. Denn diesen Ausdruck können wir gemäß der BNF auf zwei verschiedene Weisen bilden: Zunächst bilden wir die drei Ausdrücke $a_0 \equiv x$, $a_1 \equiv 7$ und $a_2 \equiv 3$. Aus a_0 und a_1 können wir dann den Ausdruck $a_3 \equiv x - 7$ und zusammen mit a_2 dann den Ausdruck $a_4 \equiv x - 7 + 3$ bilden. Wir können aber auch erst den Ausdruck $a'_3 \equiv 7 + 3$ und dann zusammen mit a_0 den Ausdruck $a'_4 \equiv x - 7 + 3$. Je nach dem wie wir die Ausdrücke gebildet haben besitzen sie eine andere Struktur, die wir wie folgt durch (geordnete[1]) Bäume darstellen können:



Dieselbe Zeichenreihe $x - 7 + 3$ bezeichnet also zwei verschiedene Bäume und damit zwei verschiedene Ausdrücke, die zu allem Überfluß auch noch verschiedene Ergebnisse liefern, wenn man für x einen Wert einsetzt.

[1]Ein Baum heißt geordnet, wenn auf den Kindern jedes Knotens eine Ordnung definiert ist. Bei uns ist diese Ordnung durch die Leserichtung von links nach rechts definiert.

Dasselbe Problem tritt auch bei der Definition von Anweisungen auf. Die Teilanweisung **while** y $\leq$ x **do** z:= y $*$ z; y:= y + 1 können wir gemäß der BNF auf zwei verschiedene Weisen aufbauen:



Je nach Interpretation verhalten sich die beiden Programme sehr unterschiedlich, weil die Anweisung y:= y + 1 im einen Fall zur Schleife gehört, im anderen aber nicht.
Es gibt verschiedene Möglichkeiten, dieses Problem zu Lösen.

1. Die ordentliche Lösung benutzt Techniken aus dem Gebiet des Übersetzerbaus bzw. der formalen Sprachen. Man kann durch verschiedene Techniken dafür sorgen, daß die Grammatiken eindeutig sind und es für jede Zeichenreihe, die von der Grammatik erzeugt wird, nur einen Ableitungsbaum gibt. Allerdings werden die Grammatiken dann meist sehr viel aufwendiger.

   Da Syntaxanalyse und Übersetzerbau nicht das Thema dieser Vorlesung sind, verfolgen wir diesen Ansatz hier nicht weiter, sondern suchen uns einen „billigeren Ausweg“.

2. Wir betrachten nicht die Zeichenreihen als die syntaktischen Objekte, sondern die Ableitungsbäume, die wir oben angeben haben. Wir benutzen also die *abstrakte Syntax* um die Semantik einer Programmiersprache zu definieren. Da die Ableitungsbäume die Struktur des syntaktischen Konstruktes liefern, haben wir damit das Problem der Mehrdeutigkeit gelöst.

   Allerdings haben wir damit das Problem auf eine andere Ebene verschoben. Denn wir wollen später Ausdrücke und Anweisungen nicht wirklich als Bäume darstellen, weil das viel zu aufwendig wäre. Außerdem ist eine textuelle Darstellung für uns viel schneller zu erfassen. Die Struktur eines Ausdrucks werden wir dann jedoch durch Klammerung

angeben. Für unser obiges Beispiel können wir beispielsweise schreiben: $a_4 \equiv (\mathrm{x} - 7) + 3$ bzw. $a_4' \equiv \mathrm{x} - (7 + 3)$. Auf ähnliche Weise benutzen wir in Anweisungen die Klammern[2] $\ulcorner$ und $\lrcorner$ um die Struktur der Anweisung explizit zu machen. Unser vorangegangenes Beispiel war wie folgt gemeint:

```
z := 1;
y := 1;
while y ≤ x do
⌜ z := y * x;
  y := y + 1  ⌟
```

Diese Klammern gehören nicht zu der Syntax unserer Programmiersprache, weil wir ja nur die abstrakte Syntax, d. h. die Ableitungsbäume betrachten. Sie dienen uns nur dazu, diese Struktur in einer ansonsten mehrdeutigen Zeichenreihe zu finden. Die Klammern sind *konkrete Syntax*, um die Struktur eines Ausdruck oder Anweisung eindeutig zu machen. Wir gehen im folgenden immer davon aus, daß eine textuelle Repräsentation einer Anweisung genug konkrete Syntax enthält, um eindeutig auf die Struktur der Anweisung zu schließen.

Streng genommen ist in der Anweisung

```
z := 1;
y := 1;
while y ≤ x do
⌜ z := y * x;
  y := y + 1  ⌟
```

immer noch nicht genügend konkrete Syntax enthalten, um eindeutig auf die Struktur zu schließen. Aber das ist nicht ganz so schlimm. Warum?

1.3 Syntaktische Gleichheit

Wir nennen zwei Ausdrücke oder Anweisungen *syntaktisch gleich*, wenn sie denselben Ableitungsbaum besitzen, d. h. wenn die abstrakte Syntax gleich ist. Um die syntaktische Gleichheit auszudrücken, benutzen wir das Symbol $\equiv$, das wir auch schon benutzt haben, um Ausdrücke und Anweisungen zu benennen. Wenn also zwei Ausdrücke a_0 und a_1 gleich sind, schreiben wir

[2] Diese Klammern ersetzen das **begin** und **end** in herkömmlichen Programmiersprachen oder die geschweiften Klammern in Java oder C.

$a_0 \equiv a_1$. Entsprechend schreiben wir für zwei Anweisungen $c_0 \equiv c_1$, wenn sie syntaktisch gleich sind.

> *Das Symbol $\equiv$ gehört nicht zur Syntax der Sprache. Wir führen es als „Meta-Symbol“ ein, um über die Syntax der Sprache zu reden. Insbesondere ist $a_0 \equiv a_1$ kein boolescher Ausdruck!*

Da wir die Gleichheit über die abstrakte Syntax formulieren, können textuell verschiedene Anweisungen syntaktisch gleich sein. Beispielsweise können sich die Anweisungen textuell durch einige Leerzeichen oder Zeilenumbrüche unterscheiden. Sie können sich aber auch durch zusätzliche redundante Klammern, also durch konkrete Syntax unterscheiden. Beispielsweise sind die Zeichenreihen x + y und (x + (y)) verschieden; die Ausdrücke sind dennoch syntaktisch gleich, da sie denselben Ableitungsbaum besitzen und damit die abstrakte Syntax gleich ist.
Eine weitere Möglichkeit für textuell verschiedene aber syntaktisch gleiche Ausdrücke und Anweisungen ist die Darstellung von Konstanten, für die wir ja keine konkrete Syntax definiert haben. Beispielsweise sind die Ausdrücke 007 und 7 syntaktisch gleich, da sie dieselbe Zahl aus $\mathbb{Z}$ bezeichnen. Entsprechend sind dann die Ausdrücke 007 + 693 und 7 + 693 syntaktisch gleich.
Allerdings sind die Ausdrücke 3 + 4 und 4 + 3 syntaktisch verschieden, da sie verschiedene (geordnete) Ableitungsbäume besitzen. Wir werden zwar später sehen, daß die beiden Ausdrücke semantisch gleich sind (wir nennen das dann äquivalent), aber syntaktisch sind sie verschieden. Entsprechend sind die beiden booleschen Ausdrücke x = y und y = x syntaktisch verschieden, obwohl sie semantisch gleich sind. Dies werden wir im folgenden sorgfältig auseinander halten.

> *Es kann passieren, daß dieselbe Zeichenreihe „syntaktisch verschieden“ zu sich selbst ist. Denn wir haben gesehen, daß x − 7 + 3 zwei Ableitungsbäume besitzt. Allerdings betrachten wir solche Zeichenreihen im folgenden nicht mehr, da wir davon ausgehen, daß die textuelle Repräsentation von Ausdrücken und Anweisungen immer genügend konkrete Syntax enthält, um die Ableitungsbäume eindeutig zu machen (vgl. Abschnitt 1.2).*

2 Semantik der Ausdrücke

Bevor wir die operationale Semantik der Anweisungen definieren können, müssen wir zunächst die Semantik der arithmetischen und booleschen Ausdrücke definieren. Die operationale Semantik eines Ausdrucks definiert die schrittweise Auswertung des Ausdrucks.

2.1 Zustände

Beispielsweise wird der Ausdruck 3 + 5 zu 8 ausgewertet und der Ausdruck 007 + 693 zu 700. Diese *Auswertungsrelation* müssen wir nun ganz allgemein für beliebige Ausdrücke definieren. So müssen wir auch den Ausdruck x + y auswerten; allerdings müssen wir dazu die aktuellen Werte von x und y kennen. Die aktuellen Werte der Variablen nennen wir *Zustand*. Im allgemeinen können wir also einen Ausdruck nur dann auswerten, wenn wir den Zustand kennen, bzw. die Auswertung eines Ausdrucks definieren wir für einen gegebenen Zustand. Wir schreiben $\langle a, \sigma \rangle \to n$, wenn der Ausdruck a im Zustand σ zu n ausgewertet wird.
Da wir in IMP nur Variablen vom Typ $\mathbb{Z}$ zugelassen haben, können wir einen Zustand wie folgt als Abbildung definieren.

Definition 3.1 (Zustand, Wert einer Variablen)
Eine totale Abbildung $\sigma : \mathbb{V} \to \mathbb{Z}$ heißt *Zustand*. Die Menge aller Zustände bezeichnen wir mit Σ (d. h. $\Sigma = \mathbb{V} \to \mathbb{Z}$). Für eine Variable $v \in \mathbb{V}$ nennen wir $\sigma(v)$ den *Wert von v im Zustand σ*.

Im folgenden können wir nun die Auswertungrelation für arithmetische und boolesche Ausdrücke in einem Zustand definieren.

2.2 Auswertungsrelation für arithmetische Ausdrücke

Die Auswertungsrelation definieren wir nun induktiv.

Definition 3.2 (Auswertungsrelation für *Aexp*)
Die *Auswertungsrelation für arithmetische Ausdrücke* ist eine dreistellige Relation über *Aexp*, Σ und $\mathbb{Z}$, wobei wir ein Element der Relation durch $\langle a, \sigma \rangle \to n$ notieren.
Die Auswertungsrelation ist induktiv über den Aufbau der arithmetischen Ausdrücke definiert:

- Für $a \equiv n \in \mathbb{Z}$ gilt: $\langle n, \sigma \rangle \to n$.
- Für $a \equiv v \in \mathbb{V}$ gilt: $\langle v, \sigma \rangle \to \sigma(v)$.
- Für $a \equiv a_0 + a_1$ mit $a_0, a_1 \in \mathit{Aexp}$ und $\langle a_0, \sigma \rangle \to n_0$ und $\langle a_1, \sigma \rangle \to n_1$ gilt: $\langle a, \sigma \rangle \to n_0 + n_1$.

Achtung, in $a_0 + a_1$ ist das Zeichen + Syntax; in n_0+n_1 ist das Zeichen + Semantik, d. h. $n_0 + n_1$ steht für die Summer der beiden Zahlen $n_0 + n_1$.
Entsprechendes gilt für die nachfolgenden Schritte.

- Für $a \equiv a_0-a_1$ mit $a_0, a_1 \in Aexp$ und $\langle a_0, \sigma\rangle \to n_0$ und $\langle a_1, \sigma\rangle \to n_1$ gilt: $\langle a, \sigma\rangle \to n_0 - n_1$.
- Für $a \equiv a_0*a_1$ mit $a_0, a_1 \in Aexp$ und $\langle a_0, \sigma\rangle \to n_0$ und $\langle a_1, \sigma\rangle \to n_1$ gilt: $\langle a, \sigma\rangle \to n_0 \cdot n_1$.

Diese Definition können wir auch durch Regeln notieren, wobei es für jeden Punkt der induktiven Definition genau eine Regel gibt. Dabei sind die Mengen, aus denen n, n_0, n_1, v, a_0, a_1 und σ gemäß unserer Konventionen gewählt werden können eine implizite Nebenbedingung für diese Regeln:

$$\frac{}{\langle n, \sigma\rangle \to n} \qquad \frac{}{\langle v, \sigma\rangle \to \sigma(v)}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0{+}a_1, \sigma\rangle \to n_0 + n_1} \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0{-}a_1, \sigma\rangle \to n_0 - n_1}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0{*}a_1, \sigma\rangle \to n_0 \cdot n_1}$$

Die *Regeln* lassen sich wie folgt lesen: Über dem Strich stehen bestimmte Aussagen, die die *Voraussetzung* der Regel bilden; wenn diese Voraussetzungen erfüllt sind, dann gilt auch die Aussage unter dem Strich, die *Schlußfolgerung*. Die ersten beiden Regeln haben keine Voraussetzung, d.h. daß ihre Schlußfolgerung in jedem Falle gilt. Solche Regeln heißen auch *Axiome*; sie entsprechen dem Induktionsanfang der induktiven Definition. Mit Hilfe der Axiome und Regeln lassen sich dann schrittweise weitere Aussagen herleiten. Da es für jedes Konstrukt der arithmetischen Ausdrücke genau eine Regel gibt, kann man leicht zeigen, daß es für jeden arithmetischen Ausdruck a und jeden Zustand σ genau eine Zahl $n \in \mathbb{Z}$ gibt, für die $\langle a, \sigma\rangle \to n$ gilt, d. h. jedem arithmetischen Ausdruck ist in jedem Zustand eindeutig ein Wert zugeordnet.

Beispiel 3.3 (Auswertung eines arithmetischen Ausdrucks)
Die Auswertung des arithmetischen Ausdrucks $(\mathrm{x} - 2) * \mathrm{y}$ in einem Zustand σ mit $\sigma(\mathrm{x}) = 2$ und $\sigma(\mathrm{y}) = 9$ können wir dann in Form eines Ableitungsbau-

mes notieren:

$$\dfrac{\dfrac{\dfrac{}{\langle \mathrm{x}, \sigma\rangle \to 2} \quad \dfrac{}{\langle 2, \sigma\rangle \to 2}}{\langle \mathrm{x} - 2, \sigma\rangle \to 0} \quad \dfrac{}{\langle \mathrm{y}, \sigma\rangle \to 9}}{\langle (\mathrm{x} - 2) * \mathrm{y}, \sigma\rangle \to 0}$$

Dabei wird zunächst der Baum von der Wurzel her von unten nach oben aufgebaut, ohne die rechten Seiten zu kennen. Diese werden dann von den Axiomen her von oben nach unten gemäß der Regeln eingefügt.

> *Allein für die Definition der Auswertung der arithmetischen Ausdrücke wäre der hier betriebene Aufwand etwas übertrieben. Die induktive Definition für die Auswertung hätte vollkommen gereicht. Nachfolgend werden wir aber die Semantik der booleschen Ausdrücke und vor allem die Semantik der Anweisungen ganz analog durch Regeln definieren. Deshalb benutzen wir der Einheitlichkeit halber auch hier schon dieselbe Technik.*

Mit Hilfe der Semantik für arithmetische Ausdrücke können wir nun definieren, wann zwei arithmetische Ausdrücke „semantisch“ gleich sind: nämlich dann, wenn beide Ausdrücke für jeden Zustand dasselbe Ergebnis liefern.

Definition 3.3 (Äquivalenz arithmetischer Ausdrücke)
Zwei arithmetische Ausdrücke a_0 und a_1 heißen *äquivalent*, wenn für jeden Zustand σ und jede Zahl $n \in \mathbb{Z}$ die Aussage $\langle a_0, \sigma\rangle \to n$ genau dann gilt, wenn auch $\langle a_1, \sigma\rangle \to n$ gilt. Wenn a_0 und a_1 äquivalent sind, dann schreiben wir $a_0 \sim a_1$.

Beispielsweise gilt $\mathrm{x} + \mathrm{y} \sim \mathrm{y} + \mathrm{x}$. Dies läßt sich einfach anhand der Regeln für die Auswertungsrelation zeigen: Wenn $\langle \mathrm{x} + \mathrm{y}, \sigma\rangle \to n$ gilt, muß dies mit der einzigen Regel für $\mathrm{x} + \mathrm{y}$ hergeleitet worden sein. Dementsprechend existieren ganze Zahlen n_0 und n_1 mit $n = n_0 + n_1$ und $\langle \mathrm{x}, \sigma\rangle \to n_0$ und $\langle \mathrm{y}, \sigma\rangle \to n_1$. Daraus läßt sich dann wieder mit der Regel für $\mathrm{y} + \mathrm{x}$ die Aussage $\langle \mathrm{y} + \mathrm{x}, \sigma\rangle \to n$ herleiten. Ganz analog kann man die umgekehrte Richtung der „genau-dann-wenn“-Aussage beweisen.

2.3 Auswertungsrelation für boolesche Ausdrücke

Die Auswertungsrelation für boolesche Ausdrücke definieren wir analog zur Definition der arithmetischen Ausdrücke. Wir geben die induktive Definition direkt in Form von Regeln an:

Definition 3.4 (Auswertungsrelation für $Bexp$)

In den folgenden Regel treten die Wahrheitswerte true *und* false *als Syntax und als Semantik auf. Wir unterscheiden diese Varianten durch die Darstellung in verschiedenen Schriftarten. Aus dem Kontext wäre aber auch ohne diese Unterscheidung immer klar, wo die syntaktische und wo die semantische Variante gemeint ist.*

$$\frac{}{\langle \text{true}, \sigma\rangle \to \mathit{true}} \qquad \frac{}{\langle \text{false}, \sigma\rangle \to \mathit{false}}$$

$$\frac{\langle a_0, \sigma\rangle \to n \quad \langle a_1, \sigma\rangle \to n}{\langle a_0{=}a_1, \sigma\rangle \to \mathit{true}} \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0{=}a_1, \sigma\rangle \to \mathit{false}} n_0 \neq n_1$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0{\leq}a_1, \sigma\rangle \to \mathit{true}} n_0 \leq n_1 \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0{\leq}a_1, \sigma\rangle \to \mathit{false}} n_0 > n_1$$

$$\frac{\langle b, \sigma\rangle \to \mathit{false}}{\langle \neg b, \sigma\rangle \to \mathit{true}} \qquad \frac{\langle b, \sigma\rangle \to \mathit{true}}{\langle \neg b, \sigma\rangle \to \mathit{false}}$$

$$\frac{\langle b_0, \sigma\rangle \to \mathit{false} \quad \langle b_1, \sigma\rangle \to t}{\langle b_0 \wedge b_1, \sigma\rangle \to \mathit{false}} \qquad \frac{\langle b_0, \sigma\rangle \to t \quad \langle b_1, \sigma\rangle \to \mathit{false}}{\langle b_0 \wedge b_1, \sigma\rangle \to \mathit{false}}$$

$$\frac{\langle b_0, \sigma\rangle \to \mathit{true} \quad \langle b_1, \sigma\rangle \to \mathit{true}}{\langle b_0 \wedge b_1, \sigma\rangle \to \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \to \mathit{true} \quad \langle b_1, \sigma\rangle \to t}{\langle b_0 \vee b_1, \sigma\rangle \to \mathit{true}} \qquad \frac{\langle b_0, \sigma\rangle \to t \quad \langle b_1, \sigma\rangle \to \mathit{true}}{\langle b_0 \vee b_1, \sigma\rangle \to \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \to \mathit{false} \quad \langle b_1, \sigma\rangle \to \mathit{false}}{\langle b_0 \vee b_1, \sigma\rangle \to \mathit{false}}$$

In unserer Definition erzwingen wir durch die Regeln beim UND- bzw. ODER-Operator immer die Auswertung beider Argumente. In vielen Programmiersprachen werden stattdessen diese Operatoren sequentiell definiert; d. h. wenn durch die Auswertung des ersten Argumentes das Ergebnis des booleschen Ausdrucks schon klar ist, wird das zweite Argument nicht mehr ausgewertet. Da die Auswertung von Ausdrücken bei uns zunächst immer definiert ist und auch keine Seiteneffekte haben kann, macht das im Ergebnis keinen Unterschied. Bei späteren Erweiterungen – die es erlauben werden, daß die Auswertung eines Ausdrucks kein Ergebnis liefert oder zu Seiteneffekten führt – macht dies aber einen Unterschied. Das werden wir uns in einer Übung genauer ansehen. Ebenso können wir einen parallelen UND- oder ODER-

Operator definieren, der das Ergebnis des Ausdrucks ausgibt, sobald es aus dem Ergebnis eines der beiden Argumente folgt.
Im Gegensatz zu den Regeln für die arithmetischen Ausdrücke, gibt es für die Auswertung der booleschen Ausdrücke in manchen Fällen mehrere Regeln die man bei der Auswertung anwenden kann. Ein Beispiel dafür ist ein boolescher Ausdruck $b_0 \wedge b_1$ bei dem sowohl b_0 als auch b_1 zu *false* ausgewertet werden. Denn dann sind zwei Regel anwendbar. Glücklicherweise führt die Anwendung beider Regeln zu demselben Ergebnis. Wir werden in einer Übung beweisen, daß die Auswertung der booleschen Ausdrücke gemäß der obigen Definition tatsächlich immer eindeutig ist.
Analog zur Definition der Äquivalenz von arithmetischen Ausdrücken, definieren wir nun die Äquivalenz der booleschen Ausdrücke. Zwei boolesche Ausdrücke sind äquivalent, wenn sie in jedem Zustand gleich ausgewertet werden:

Definition 3.5 (Äquivalenz boolescher Ausdrücke)
Zwei boolesche Ausdrücke b_0 und b_1 heißen *äquivalent*, wenn für jeden Zustand σ und jeden Wahrheitswert $t \in \mathbb{B}$ die Aussage $\langle b_0, \sigma \rangle \to t$ genau dann gilt, wenn auch $\langle b_1, \sigma \rangle \to t$ gilt. Wenn b_0 und b_1 äquivalent sind, dann schreiben wir $b_0 \sim b_1$.

Beispielsweise können wir die Regel von De Morgan als Äquivalenz formulieren: Für alle booleschen Ausdrücke b_0 und b_1 gilt $b_0 \vee b_1 \sim \neg(\neg b_0 \wedge \neg b_1)$. Diese kann man mit Hilfe der Regeln zur Definition der Auswertungsrelation nachweisen.

3 Semantik der Anweisungen

Nachdem wir nun Ausdrücke auswerten können, werden wir als nächstes die Semantik von Anweisungen der Programmiersprache IMP angeben. Dabei definieren wir eine dreistellige Relation über Com, Σ und Σ. Ein Element dieser Relation geben wir in der folgenden Notation an:

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

Dabei bedeutet $\langle c, \sigma \rangle \to \sigma'$, daß die Anweisung c im Zustand σ' terminiert, wenn man sie im Zustand σ startet. Für die Zuweisung x:= 5 gilt beispiels-

weise $\langle \text{x:= } 5, \sigma\rangle \to \sigma'$, wobei der Zustand σ' wie folgt definiert ist:

$$\sigma'(v) = \begin{cases} 5 & \text{für } v \equiv \text{x} \\ \sigma(v) & \text{für } v \not\equiv \text{x} \end{cases}$$

Der Zustand σ' entsteht dabei aus dem Zustand σ durch Modifikation der Abbildung an (ausschließlich) der Stelle x. Da wir solche Modifikationen bei der Definition der Semantik immer wieder benötigen, führen wir dafür eine eigene Notation ein: Wir schreiben $\sigma' = \sigma[5/\text{x}]$.

Definition 3.6 (Modifikation eines Zustandes)
Für einen Zustand $\sigma \in \Sigma$, eine ganze Zahl $n \in \mathbb{Z}$ und eine Variable $u \in \mathbb{V}$ bezeichnet $\sigma[n/u]$ einen Zustand (d. h. $\sigma[n/v] \in \Sigma$), der wie folgt definiert ist:

$$\sigma[n/u](v) = \begin{cases} n & \text{für } v \equiv u \\ \sigma(v) & \text{für } v \not\equiv u \end{cases}$$

Für einen Zustand σ, ganze Zahlen $n_1, n_2, \ldots, n_k \in \mathbb{Z}$ und Variablen $u_1, u_2, \ldots, u_k \in \mathbb{V}$ bezeichnet $\sigma[n_1/u_1, n_2/u_1, \ldots n_k/u_k]$ den Zustand $\sigma[n_1/u_1][n_2/u_1]\ldots[n_k/u_k]$.

Mit Hilfe dieser Notation können wir nun die Semantik für Anweisungen definieren:

Definition 3.7 (Semantik von Anweisungen)
Die Semantik für Anweisungen ist durch die folgenden Regeln definiert:

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma} \qquad \frac{\langle a, \sigma\rangle \to n}{\langle v{:=}a, \sigma\rangle \to \sigma[n/v]} \qquad \frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0\,; c_1, \sigma\rangle \to \sigma'}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'} \qquad \frac{\langle b, \sigma\rangle \to \mathit{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

*In der letzten Regel tritt die Anweisung **while** b **do** c in ihrer eigenen Voraussetzung wieder auf. Allerdings wird sie dort im allgemeinen in einem anderen Zustand betrachtet. Deshalb terminiert irgendwann der Aufbau des Ableitungsbaumes für die Semantik, wenn die Schleife terminiert. Wenn die Schleife nicht terminiert, können wir allerdings kein $\langle$**while** b **do** $c, \sigma\rangle \to \sigma'$ herleiten. Aber dies bedeutet ja gerade, daß die Schleife nicht terminiert.*

Wie wir eben gesehen haben, muß es nicht für jede Anweisung c und jeden Zustand σ einen Zustand σ' mit $\langle c, \sigma\rangle \to \sigma'$ geben (im Gegensatz zu der Auswertung von Ausdrücken). Wenn ein solches σ' nicht existiert, bedeutet dies gerade, daß die Anweisung c nicht terminiert, wenn sie im Zustand σ gestartet wird. Allerdings gibt es für jede Anweisung c und jeden Zustand σ' immer höchstens ein σ' mit $\langle c, \sigma\rangle \to \sigma'$. Dies bedeutet gerade, daß die Semantik der Programmiersprache IMP deterministisch ist.

Beispiel 3.4 (Semantik einer Schleife)
Wir leiten nun mit Hilfe der Regeln die Semantik für die folgende Anweisung $w \equiv$ **while** $0 \leq$ i **do** i:= i-1 im Zustand σ mit $\sigma(\mathrm{i}) = 2$ her. Dabei sind $\sigma_1 = \sigma[1/\mathrm{i}]$, $\sigma_0 = \sigma[0/\mathrm{i}]$ und $\sigma_{-1} = \sigma[-1/\mathrm{i}]$.

$$\dfrac{\dfrac{\vdots}{\langle 0 \leq \mathrm{i}, \sigma\rangle \to \mathit{true}} \quad \dfrac{\dfrac{\vdots}{\langle \mathrm{i}-1, \sigma\rangle \to 1}}{\langle \mathrm{i} := \mathrm{i}-1, \sigma\rangle \to \sigma_1} \quad \dfrac{\dfrac{\vdots}{\langle 0 \leq \mathrm{i}, \sigma_1\rangle \to \mathit{true}} \quad \dfrac{\dfrac{\vdots}{\langle \mathrm{i}-1, \sigma_1\rangle \to 0}}{\langle \mathrm{i} := \mathrm{i}-1, \sigma_1\rangle \to \sigma_0} \quad \dfrac{\dfrac{\vdots}{\langle 0 \leq \mathrm{i}, \sigma_0\rangle \to \mathit{true}} \quad \dfrac{\dfrac{\vdots}{\langle \mathrm{i}-1, \sigma_0\rangle \to -1}}{\langle \mathrm{i} := \mathrm{i}-1, \sigma_0\rangle \to \sigma_{-1}} \quad \dfrac{\vdots}{\langle 0 \leq \mathrm{i}, \sigma_{-1}\rangle \to \mathit{false}}}{\langle w, \sigma_0\rangle \to \sigma_{-1}}}{\langle w, \sigma_1\rangle \to \sigma_{-1}}}{\langle w, \sigma\rangle \to \sigma_{-1}}$$

Beispiel 3.5 (Endlosschleife)
Ein weiteres Beispiel ist die folgende Schleife: $w \equiv$ **while** true **do skip**. Wenn wir für einen Zustand σ einen Tripel $\langle w, \sigma\rangle \to \sigma'$ ableiten wollen, stellen wir fest, daß die Regel für die Schleife wieder dieselbe Voraussetzung hat:

$$\frac{\dfrac{}{\langle \text{true}, \sigma\rangle \to \mathit{true}} \quad \dfrac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma} \quad \dfrac{\vdots}{\langle w, \sigma\rangle \to ?}}{\langle w, \sigma\rangle \to ?}$$

Wir geraten also beim Suchen nach einer Ableitung in eine Endlosschleife und man sieht leicht, daß es für kein σ' eine Herleitung für $\langle w, \sigma\rangle \to \sigma'$ geben kann. Dies ist auch nicht weiter verwunderlich, da dies ja gerade dem Verhalten einer Endloschleife entspricht.

> *Wir werden später (siehe Beispiel 4.5 in Kapitel 4 auf Seite 55) eine Technik kennenlernen, mit der man auch beweisen kann, daß für kein σ und σ' ein Tripel $\langle w, \sigma\rangle \to \sigma'$ herleitbar ist.*

Ganz analog zu Ausdrücken können wir nun auch definieren, wann zwei Anweisungen äquivalent sind, nämlich genau dann, wenn beide für jeden Anfangszustand im selben Zustand terminieren (oder beide nicht terminieren).

Definition 3.8 (Äquivalenz von Anweisungen)
Zwei Anweisungen c_0 und c_1 heißen *äquivalent*, wenn für alle Zustände σ und σ' die Aussage $\langle c_0, \sigma\rangle \to \sigma'$ genau dann gilt, wenn auch $\langle c_1, \sigma\rangle \to \sigma'$ gilt. Wenn c_0 und c_1 äquivalent sind, dann schreiben wir $c_0 \sim c_1$.

Mit Hilfe der Definition der Semantik von IMP können wir dann auch beweisen, daß bestimmte Anweisungen äquivalent sind. Dazu betrachten wir wieder ein Beispiel.

Beispiel 3.6
Sei $w \equiv$ **while** b **do** c wobei b ein beliebiger boolescher Ausdruck und c eine beliebige Anweisung ist. Dann gilt $w \sim$ **if** b **then** c; w **else skip**
Um dies zu beweisen, müssen wir zeigen, daß für alle Zustände σ und σ' gilt:

$$\langle w, \sigma\rangle \to \sigma' \quad \text{gdw} \quad \langle \textbf{if } b \textbf{ then } c \ ; \ w \textbf{ else skip}, \sigma\rangle \to \sigma'$$

Wir betrachten die beiden Richtungen dieser Genau-Dann-Wenn-Aussage einzeln:

„⇒“: Gelte also $\langle w, \sigma\rangle \to \sigma'$. Dann gibt es eine Herleitung für $\langle w, \sigma\rangle \to \sigma'$. Wir zeigen nun, daß es dann auch eine Herleitung für

$$\langle \textbf{if } b \textbf{ then } c \ ; \ w \textbf{ else skip}, \sigma\rangle \to \sigma'$$

gibt. Gemäß der Regeln für die Semantik der Schleife hat die Herleitung dann eine der beiden folgenden Formen:

$$(1) \quad \dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathit{false}}}{\langle w, \sigma\rangle \to \sigma}$$

oder

$$(2) \quad \dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathit{true}} \quad \dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle w, \sigma\rangle \to \sigma'}$$

Wir betrachten nun diese beiden Fälle einzeln:

Fall (1)**:** In diesem Falle gilt $\sigma = \sigma'$ und es gibt eine Herleitung für $\langle b, \sigma\rangle \to \mathit{false}$. Daraus können wir nun die folgende Herleitung konstruieren:

$$\dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathit{false}} \quad \dfrac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}}{\langle \textbf{if } b \textbf{ then } c; \ w \textbf{ else skip}, \sigma\rangle \to \sigma}$$

Mit $\sigma' = \sigma$ gilt die Behauptung.

Fall (2)**:** In diesem Falle gibt es Herleitungen für $\langle b, \sigma\rangle \to \mathit{true}$, $\langle c, \sigma\rangle \to \sigma''$ und $\langle w, \sigma''\rangle \to \sigma'$. Aus diesen Herleitungen können wir die folgende Herleitung konstruieren:

$$\dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathit{true}} \quad \dfrac{\dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; \ w, \sigma\rangle \to \sigma'}}{\langle \textbf{if } b \textbf{ then } c; \ w \textbf{ else skip}, \sigma\rangle \to \sigma'}$$

Und damit gilt die Behauptung.

„⇐“: Sei nun also $\langle \textbf{if } b \textbf{ then } c;\ w \textbf{ else skip}, \sigma\rangle \to \sigma'$ herleitbar. Diese Herleitung kann gemäß der Regeln für die Bedingung, die Anweisung **skip** und die Sequenz die beiden folgenden Formen haben:

$$(1)\quad \frac{\dfrac{\vdots}{\langle b,\sigma\rangle \to \mathit{false}} \qquad \dfrac{}{\langle \textbf{skip},\sigma\rangle \to \sigma}}{\langle \textbf{if } b \textbf{ then } c;\ w \textbf{ else skip}, \sigma\rangle \to \sigma}$$

d. h. $\sigma' = \sigma$ und

$$(2)\quad \frac{\dfrac{\vdots}{\langle b,\sigma\rangle \to \mathit{true}} \qquad \dfrac{\dfrac{\vdots}{\langle c,\sigma\rangle \to \sigma''} \qquad \dfrac{\vdots}{\langle w,\sigma''\rangle \to \sigma'}}{\langle c;\ w,\sigma\rangle \to \sigma'}}{\langle \textbf{if } b \textbf{ then } c;\ w \textbf{ else skip}, \sigma\rangle \to \sigma'}$$

Aus diesen Herleitungen müssen wir nun jeweils eine Herleitung für $\langle w,\sigma\rangle \to \sigma'$ konstruieren:

Fall (1)**:** In diesem Falle können wir daraus die folgende Herleitung konstruieren:

$$\frac{\dfrac{\vdots}{\langle b,\sigma\rangle \to \mathit{false}}}{\langle w,\sigma\rangle \to \sigma}$$

Wegen $\sigma' = \sigma$ ist diese eine Herelitung für $\langle w,\sigma\rangle \to \sigma'$.

Fall (2)**:** In diesem Falle können wir daraus die folgende Herleitung konstruieren:

$$\frac{\dfrac{\vdots}{\langle b,\sigma\rangle \to \mathit{true}} \qquad \dfrac{\vdots}{\langle c,\sigma\rangle \to \sigma''} \qquad \dfrac{\vdots}{\langle w,\sigma''\rangle \to \sigma'}}{\langle w,\sigma\rangle \to \sigma'}$$

Dies ist die Herleitung für $\langle w,\sigma\rangle \to \sigma'$.

Insgesamt haben wir gezeigt, daß wir eine Herleitung für $\langle w,\sigma\rangle \to \sigma'$ immer in eine Herleitung für $\langle \textbf{if } b \textbf{ then } c\ ;\ w \textbf{ else skip}, \sigma\rangle \to \sigma'$ „umbauen“ können und umgekehrt. Damit ist also die Äquivalenz beider Anweisungen bewiesen.

4 Alternative Definitionen

In den vorangegangenen Abschnitten haben wir eine Semantik für die Programmiersprache IMP angegeben. Natürlich kann man dieselbe Semantik auf viele verschiedene Weisen definieren. Beispielsweise haben wir in unsere Definition keine Reihenfolge für die Auswertung der Operanden in einem arithmetischen oder booleschen Ausdruck festgelegt. Da die Programmiersprache IMP keine Seiteneffekte zuläßt, ist die Auswertungreihenfolge für das Ergebnis auch irrelevant. Wir werden aber in der Übung einige Erweiterungen von IMP betrachten, die Seiteneffekte haben. Dann macht es einen Unterschied, in welcher Reihenfolge die Operanden ausgewertet werden.

> *Ein Beispiel für einen arithmetischen Ausdruck mit einem Seiteneffekt ist das Konstrukt x++, das aus der Programmiersprache C oder Java bekannt ist. Neben der Auswertung der Variablen wird der Wert der Variablen auch verändert.*

In diesem Abschnitt zeigen wir anhand einiger Ausschnitte eine alternative Definition der Semantik der Programmiersprache IMP. In dieser Semantik wird der Charakter des schrittweisen Auswertens von Ausdrücken und des schrittweisen Abarbeitens von Anweisungen noch deutlicher:

> **Achtung:** *Die nachfolgenden Regeln sind nicht vollständig. Sie sollen nur die Idee der Semantikdefinition vermitteln. Wir werden diese Semantik in der Übung vervollständigen. Damit wir die neue Definition von der vorangegangenen unterscheiden können, geben wir ihr den Index* 2.

Regeln für die Auswertung von Ausdrücken

$$\frac{}{\langle n + m, \sigma\rangle \to_2 \langle k, \sigma\rangle}\, k = n + m$$

$$\frac{\langle a_0, \sigma\rangle \to_2 \langle a_0', \sigma'\rangle}{\langle a_0 + a_1, \sigma\rangle \to_2 \langle a_0' + a_1, \sigma'\rangle} \qquad \frac{\langle a_1, \sigma\rangle \to_2 \langle a_1', \sigma'\rangle}{\langle n + a_1, \sigma\rangle \to_2 \langle n + a_1', \sigma'\rangle}$$

$$\vdots$$

Im Gegensatz zur vorangegangenen Definition geben wir für einen Ausdruck nicht nur das Ergebnis der Auswertung an, sondern auch den resultierenden Zustand. Bei den obigen Regeln bleibt dieser Zustand immer gleich. Aber das Schema ermöglicht es uns, später bei der Auswertung von Ausdrücken auch den Zustand zu verändern.

Regeln für die Ausführung von Anweisungen

$$\frac{\langle a, \sigma\rangle \to_2 \langle a', \sigma'\rangle}{\langle v{:=}\ a, \sigma\rangle \to_2 \langle v{:=}\ a', \sigma[n/v]\rangle} \qquad \frac{}{\langle v{:=}\ n, \sigma\rangle \to_2 \langle \mathbf{skip}, \sigma[n/v]\rangle}$$

$$\frac{\langle b, \sigma\rangle \to_2 \langle b', \sigma'\rangle}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to_2 \langle \mathbf{if}\ b'\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma'\rangle}$$

$$\frac{}{\langle \mathbf{if}\ \text{true}\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to_2 \langle c_0, \sigma\rangle}$$

$$\frac{}{\langle \mathbf{if}\ \text{false}\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to_2 \langle c_1, \sigma\rangle}$$

$$\vdots$$

In dieser Semantik werden bei der Ausführung der Anweisungen, die Ausdrücke schrittweise vereinfacht und auch die Anweisungen selbst werden vereinfacht. Solche Semantiken werden auch *Textersetzungssemantiken* genannt.

5 Zusammenfassung

In diesem Kapitel haben wir gezeigt, wie man für imperative Programmiersprachen mit Hilfe von Regeln eine Semantik angeben kann. Für jedes Konstrukt (jeden Operator) der Programmiersprache gibt es Regeln, die die Semantik dieses Konstruktes beschreiben. Die Semantik jedes Konstruktes kann dabei unabhängig von der Semantik der anderen Konstrukte beschrieben werden. Deshalb kann man relativ einfach neue Konstrukte zu einer Programmiersprache hinzufügen und die Semantik einfach erweitern. Diese Prinzip ist recht vielseitig und läßt sich auf die unterschiedlichsten Programmiersprachen anwenden. Sehr verbreitet ist dies Art der Semantikdefinition im Bereich der *Prozeßalgebren*, wie z. B. CCS zur Beschreibung der Interaktion verschiedener Prozesse [9]. Dieses Prinzip wurde von Gordon Plotkin eingeführt und *Strukturelle Operationale Semantik (engl. structural operational semantics)* (SOS) genannt [10].
Tatsächlich ist die Definition einer Strukturellen Operationalen Semantik eine induktive Definition. Die Beweise die wir damit führen sind Induktionsbeweise. Deshalb – und weil Induktion in der Informatik fast überall vorkommt

– werden wir uns im nächsten Kapitel das Prinzip der induktiven Definition und des induktiven Beweisens etwas genauer ansehen. Dabei werden wir insbesondere die Beweistechniken, die wir hier benutzt haben auf eine formale Grundlage stellen.
Zum Beweis der Äquivalenz von Ausdrücken und Anweisungen haben wir diese Techniken schon angewendet. Dabei haben wir insbesondere ausgenutzt, daß die Regeln die Struktur einer Herleitung festlegen und aus der Herleitung für eine Aussage ein andere konstruiert. Diese Beweise sind meist nicht schwierig, aber aufwendig. Deshalb werden wir für den Nachweis der Korrektheit der Programme später andere Techniken kennen lernen, die nicht unmittelbar auf der Definition der Semantik arbeiten.

Kapitel 4

Induktive Definitionen und Beweise

Bei der Definition der Semantik der Programmiersprache IMP haben wir an vielen verschiedenen Stellen induktive Definitionen benutzt: angefangen bei der Syntax von IMP, über die Semantik der Ausdrücke bis hin zur Semantik der Anweisungen. Teilweise waren die Definitionen explizit induktiv, wie beispielsweise bei der Definition der Semantik für arithmetische Ausdrücke. Teilweise waren die Definitionen „versteckt“ induktiv. Beispielsweise verbirgt sich hinter der Definition der Syntax durch eine Grammatik auch eine induktive Definition; ebenso verbirgt sich hinter der Definition der Semantik durch Regeln eine induktive Definition.

Wenn man genau hinsieht, gibt es in der Informatik fast nichts, was nicht induktiv definiert wäre. Davon werden wir uns im weiteren Verlauf der Vorlesung noch überzeugen können. Deshalb spielen induktive Definitionen und Beweise in der Informatik eine ganz zentrale Rolle – ob man sie nun explizit macht oder nicht.

Aus diesem Grunde beschäftigen wir uns in diesem Kapitel ausführlich mit diesem Thema. Wir beginnen damit, daß wir das Prinzip der *vollständigen Induktion* zur *Noetherschen Induktion* verallgemeinern. Danach werden wir das Prinzip der *induktiven Definition* mit Hilfe von *Regeln* und der durch sie definierten Menge präzisieren. Dann werden wir zeigen, wie man Eigenschaften von induktiv definierten Mengen beweisen kann: die *Regelinduktion*. Am Ende beschäftigen wir uns dann mit der *Herleitung* und der Definition *induktiv über die Struktur einer Menge*.

1 Noethersche Induktion

Ein grundlegendes und sehr einfaches Beweisprinzip, das teilweise schon im Schulunterricht in der Oberstufe vermittelt wird, ist das Prinzip der *vollständigen Induktion*. Dabei beweist man, daß eine Aussage für alle natürlichen Zahlen gilt, indem man die Aussage für $i = 0$ beweist und darüber hinaus zeigt, daß die Aussage für $i + 1$ gilt, falls sie für i gilt. Man „hangelt“ sich mit diesem Prinzip ausgehend von der Aussage für 0 zu jeder natürlichen Zahl durch. Die Aussage gilt damit für jede natürliche Zahl.
Dieses Prinzip können wir wie folgt formulieren, wobei wir die Aussage durch ein Prädikat $P \subseteq \mathbb{N}$ formalisieren. Wir sagen, daß das Prädikat bzw. die Aussage für eine Zahl n gilt, wenn $n \in P$ gilt; wir schreiben dafür auch $P(n)$.

Prinzip 4.1 (Vollständige Induktion)
Sei $P \subseteq \mathbb{N}$ ein Prädikat über den natürlichen Zahlen. Wenn

Induktionsanfang: $P(0)$ gilt und

Induktionsschritt: für jedes $i \in \mathbb{N}$ mit $P(i)$ auch $P(i + 1)$ gilt,

dann gilt $P(n)$ für jedes $n \in \mathbb{N}$ (d. h. $P = \mathbb{N}$).

> *Im Induktionsschritt nennt man die Voraussetzung $P(i)$ auch die* Induktionsvoraussetzung.
>
> *Man kann das Prinzip der vollständigen Induktion auch knapp wie folgt formulieren:*
>
> $$(P(0) \wedge \forall i \in \mathbb{N}.(P(i) \Rightarrow P(i + 1))) \Rightarrow \forall n \in \mathbb{N}.P(n)$$
>
> *Tatsächlich ist das Induktionsprinzip ein Axiom zur Formalisierung der natürlichen Zahlen. Deshalb beweisen wir das Induktionsprinzip auch nicht. Allerdings beschäftigen wir uns hier nicht mit der Axiomatisierung der natürlichen Zahlen. Wir wenden das Prinzip „nur“ an – zum Beweisen von Aussagen.*

Um zu sehen, wie man das Induktionsprinzip zum Beweisen einer Aussage einsetzen kann, betrachten wir ein einfaches Beispiel.

> *In der Vorlesung wird das Prinzip der vollständigen Induktion nur kurz wiederholt. Der folgende Beweis wird gar nicht besprochen. Er sollte aber ohne Probleme mit den Kenntnissen des Grundstudiums verständlich sein.*

Beispiel 4.1
Aus der Schule wissen wir, daß sich die Summe aller Zahlen von 1 bis zu einer Zahl n geschlossen durch den Ausdruck $\frac{n\cdot(n+1)}{2}$ darstellen läßt, d. h. für jede natürliche Zahl $n \in \mathbb{N}$ gilt:

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

Wir beweisen diese Aussage nun mit Hilfe der vollständigen Induktion. Dabei ist das Prädikat P wie folgt definiert:

$$P(n) \equiv \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

Wir beweisen nun durch vollständige Induktion, daß $P(n)$ für jedes $n \in \mathbb{N}$ gilt:

Induktionsanfang: Wir müssen $P(n)$ für $n = 0$, d. h. $\sum_{i=1}^{0} i = \frac{0\cdot(0+1)}{2}$, zeigen. Offensichtlich gilt $\sum_{i=1}^{0} i = 0 = \frac{0\cdot(0+1)}{2}$.

Induktionsvoraussetzung: Wir gehen nun davon aus, daß für ein $n \in \mathbb{N}$ die Aussage $P(n)$ gilt, d. h. $\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$.

Induktionsschritt: Wir zeigen nun, daß dann auch die Aussage $P(n+1)$ gilt, d. h. $\sum_{i=1}^{n+1} i = \frac{n+1\cdot((n+1)+1)}{2}$:

$$\begin{array}{rcll}
\sum_{i=1}^{n+1} i & = & (\sum_{i=1}^{n} i) + (n+1) & \text{Aufteilung der Summe} \\
& = & \frac{n\cdot(n+1)}{2} + (n+1) & \text{Induktionsvoraussetzung} \\
& = & \frac{n\cdot(n+1)+2\cdot(n+1)}{2} & \text{Rechenregeln} \\
& = & \frac{(n+1)\cdot(n+2)}{2} & \text{Rechenregeln} \\
& = & \frac{(n+1)\cdot((n+1)+1)}{2} & \text{Rechenregeln}
\end{array}$$

Gemäß des Prinzips der vollständigen Induktion gilt damit $P(n)$ für jedes $n \in \mathbb{N}$. Die Aussage $\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$ ist damit für jedes $n \in \mathbb{N}$ bewiesen.

Diese Aussage kann man auch ohne (explizite Benutzung der vollständigen Induktion) beweisen:

$$\begin{array}{ccccccccc}
1 & + & 2 & + & \ldots & + & (n-1) & + & n \\
n & + & (n-1) & + & \ldots & + & 2 & + & 1 \\
\hline
(n+1) & + & (n+1) & + & \ldots & + & (n+1) & + & (n+1)
\end{array}$$

Die doppelte Summe aller Zahlen von 1 bis n ist also $n \cdot (n+1)$. Allerdings verbirgt sich hinter den Pünktchen ... doch wieder eine heimliche Induktion.

Das Beweisprinzip der Induktion ist nicht auf die Struktur der natürlichen Zahlen beschränkt. Die einzige Voraussetzung ist, daß die zugrundeliegende Struktur einen oder mehrere „Anfänge“ besitzt und daß jedes Element ausgehend von diesen „Anfängen“ irgendwann erreicht wird. Solche Strukturen sind gerade die wohlgegründeten Ordnungen (siehe Kapitel 2 Abschnitt2). Das Prinzip der Noetherschen Induktion sagt, daß eine Aussage für alle Elemente einer wohlgegründeten Ordnung gilt, wenn man für jedes Element zeigen kann, daß die Aussage für das Element selbst gilt, wenn sie für alle Vorgänger des Elementes gilt.

Prinzip 4.2 (Noethersche Induktion)
Sei $(X, \prec)$ eine wohlgegründete (irreflexive) Ordnung und $P \subseteq X$ ein Prädikat über X. Wenn für jedes $x \in X$, für das für jedes $y \in Y$ mit $y \prec x$ die Aussage $P(y)$ gilt, auch $P(x)$ gilt, dann gilt für jedes $z \in X$ die Aussage $P(z)$ (d. h. $P = X$).

Wir können analog zum Prinzip der vollständigen Induktion das Prinzip der Noetherschen Induktion wie folgt formulieren:

$$(\forall x \in X.((\forall y \prec x.P(y)) \Rightarrow P(x))) \Rightarrow \forall z \in X.P(z)$$

Die Bedingung $\forall y \prec x.P(y)$ ist dann gerade die Induktionsvoraussetzung für $P(x)$ in der Noetherschen Induktion.

Die Noethersche Induktion hat ihren Namen von der Mathematikerin Emmy Noether erhalten.

Oft wird das Prinzip der Noetherschen Induktion noch allgemeiner für wohlgegründete Relationen formuliert (eine Relation ist wohlgegründet, wenn sie keine unendlich absteigende Ketten besitzt).

Die Prinzipien der Noetherschen Induktion und der vollständigen Induktion sind sich strukturell sehr ähnlich. Im Induktionsschritt zeigt man für jedes Element, daß die Aussage für dieses Element gilt, wenn sie für alle seine Vorgänger gilt. Was man bei der Noetherschen Induktion auf den ersten Blick vermißt, ist der Induktionsanfang. Haben wir den Induktionsanfang vergessen?
Die Antwort ist, daß der Induktionsanfang im Induktionsschritt enthalten ist. Das sieht man, wenn wir ein minimales Element $x \in X$ der Ordnung betrachten[1]. Per Annahme besitzt x keine Vorgänger. Dementsprechend ist

[1] Zur Erinnerung: In der ersten Übung haben wir gezeigt, daß jede nicht-leere Teilmenge von X einer wohlgegründeten Ordnung ein minimales Element enthält. Deshalb besitzt X ein minimales Element, wenn X wenigstens ein Element enthält.

die Induktionsvoraussetzung $\forall y \prec x.P(y)$ für jedes minimales Element eine triviale Aussage (eine über die leere Menge). Für ein minimales Element x ist dementsprechend die Bedingung der Noetherschen Induktion äquivalent zu $P(x)$. Wir müssen also für die minimalen Elemente x die Aussage $P(x)$ ohne weitere Voraussetzungen beweisen. Das entspricht gerade dem Induktionsanfang.

Beispiel 4.2 (Euklid)
In Beispiel 3.1 hatten wir bereits den Algorithmus von Euklid zum Berechnen des größten gemeinsamen Teilers zweier Zahlen in unsere Programmiersprache IMP formuliert:

```
while ¬(x = y) do
⌜ if x ≤ y then y:= y − x
             else x:= x − y   ⌟
```

Der Einfachheit halber bezeichnen wir dieses Programm mit E für Euklid. Wir werden nun mit Hilfe der Noetherschen Induktion beweisen, daß dieses Programm für jeden Zustand σ mit $\sigma(x) \geq 1$ und $\sigma(y) \geq 1$ terminiert, d. h. daß ein Zustand σ' mit $\langle E, \sigma\rangle \to \sigma'$ existiert.
Zunächst definieren wir die irreflexive Ordnung, über die wir die Induktion ausführen. Wir definieren $X = \{\sigma \in \Sigma \mid \sigma(x) \geq 1 \wedge \sigma(y) \geq 1\}$. Die Ordnung $\prec$ auf X definieren wie folgt: $\sigma' \prec \sigma$ gdw. $\sigma'(x) \leq \sigma(x)$ und $\sigma'(y) \leq \sigma(y)$ und $\sigma'(x) \neq \sigma(x)$ oder $\sigma'(y) \neq \sigma(y)$. Die Ordnung $(X, \prec)$ ist dann wohlgegründet.
Zunächst formalisieren wir das Prädikat, das wir dann mit Hilfe der Noetherschen Induktion beweisen werden:

$$P(\sigma) = \exists \sigma' \in \Sigma.\langle E, \sigma\rangle \to \sigma'$$

Sei nun $\sigma \in \Sigma$ beliebig:

Induktionsannahme: Wir nehmen an, daß für jedes $\sigma'' \prec \sigma$ ein σ''' mit $\langle E, \sigma''\rangle \to \sigma'''$ existiert.

Induktionsschritt: Wir beweisen nun, daß dann auch für σ ein σ' mit $\langle E, \sigma\rangle \to \sigma'$ existiert:

Dazu unterscheiden wir zwei Fälle:

1. **Fall** $\sigma(x) = \sigma(y)$**:** In diesem Falle gilt $\langle \neg(x = y), \sigma\rangle \to \mathit{false}$. Mit der 1. Regel für die Schleife ist damit $\langle E, \sigma\rangle \to \sigma$ herleitbar. Damit gilt die zu beweisende Aussage (mit $\sigma' = \sigma$).

2. Fall $\sigma(x) \neq \sigma(y)$**:** In diesem Falle gilt $\langle \neg(\mathrm{x} = \mathrm{y}), \sigma\rangle \rightarrow \mathit{true}$. Außerdem ist mit den Regeln für die Bedingte Anweisung und für die Zuweisung

$$\langle \textbf{if } (\mathrm{x} \leq \mathrm{y}) \textbf{ then } \mathrm{y} := \mathrm{y} - \mathrm{x} \textbf{ else } \mathrm{x} := \mathrm{x} - \mathrm{y}, \sigma\rangle \rightarrow \sigma''$$

mit

$$\sigma'' = \begin{cases} \sigma[\sigma(y) - \sigma(x)/y] & \text{für } \sigma(y) > \sigma(x) \\ \sigma[\sigma(x) - \sigma(y)/x] & \text{für } \sigma(x) > \sigma(y) \end{cases}$$

herleitbar. Insbesondere gilt $\sigma'' \in X$ und $\sigma'' \prec \sigma$. Wegen Induktionsvoraussetzung gilt also $P(\sigma'')$. Es gibt also ein σ''' mit $\langle E, \sigma''\rangle \rightarrow \sigma'''$. Mit der Regel für die Schleife können wir damit $\langle E, \sigma\rangle \rightarrow \sigma'''$ herleiten. Damit gilt die zu beweisende Aussage (mit $\sigma' = \sigma'''$).

Damit haben wir per Noetherscher Induktion die Aussage $P(\sigma)$ für jedes $\sigma \in X$ bewiesen.

In diesem Beispiel hätten wir den Beweis mit Hilfe der Vollständigen Induktion führen können. Mit Hilfe der Noetherschen Induktion wird der Beweis aber oft viel einfacher.

2 Induktive Definitionen

Wir haben im Kapitel 3 *induktive Definitionen* in verschiedenen Formen benutzt: Grammatiken (bzw. die BNF), Regeln und die explizite Form. Hinter allen diesen Definitionen steckt dasselbe Prinzip:

- Für bestimmte Elemente wird gesagt, daß sie unbedingt zu der definierten Menge gehören.
- Für andere Elemente wird gesagt, daß sie unter der Voraussetzung zu der definierten Menge gehören, daß andere Elemente Menge bereits zu der Menge gehören.

Der erste Fall entspricht gerade den Axiomen, der zweite Fall entspricht gerade den Regeln (mit mind. einer Voraussetzung).
Um den Begriff der induktiven Definition formal zu fassen, definieren wir dazu zunächst den Begriff der Regel bzw. der Regelinstanz. Dabei gehen wir immer

davon aus, daß die Regeln auf einer vorgegebenen Menge von „potentiellen Objekten“ operieren und die Regeln dann eine Teilmenge davon definieren. In der Praxis wird diese Menge von „potentiellen Objekten“ oft nicht explizit erwähnt. Für die Formalisierung des Begriffes müssen wir diese Menge X, auf der die Regeln arbeiten, explizit machen.

Definition 4.3 (Regel und Axiom)
Sei X eine Menge. Für eine endliche Teilmenge $Y \subseteq X$ und ein Element $x \in X$ nennen wir das Paar (Y, x) eine *Regelinstanz* über X. Die Elemente der Menge Y nennen wir die *Voraussetzungen* der Regel, das Element x nennen wir die *Folgerung* der Regel.

Manchmal reden wir auch von der linken und rechten Seite einer Regel.

Eine Regelinstanz $(\emptyset, x)$ nennen wir *Axiominstanz*.

Im folgenden werden wir meist nur von Regeln und Axiomen reden, wenn wir Regelinstanzen und Axiominstanzen meinen. Der Grund für die Unterscheidung zwischen dem Begriff der Regel und der Regelinstanz ist, daß wir bei der syntaktischen Formulierung einer Regel meist unendlich viele Regelinstanzen bezeichnen. Beispielsweise steht die eine Regel bzw. das eine Axiom über $Aexp \times \Sigma \times \mathbb{Z}$ aus Abschnitt 2.2

$$\frac{}{\langle n, \sigma \rangle \to n}$$

für unendlich viele Regelinstanzen: Für jedes $n \in \mathbb{Z}$ und jeden Zustand $\sigma \in \Sigma$ ist $(\emptyset, (n, \sigma, n))$ eine Instanz dieser Regel. Um zwischen der syntaktischen Repräsentation einer Regel und ihren meist unendlich vielen Instanzen unterscheiden zu können, haben wir in der obigen Definition über Regelinstanzen geredet. Da wir uns aber über die syntaktische Repräsentation von Regeln keine weitere Gedanken machen, werden wir im folgenden nur noch von Regeln reden, wenn wir eigentlich Regelinstanzen meinen.
Wenn wir nun eine Menge von Regeln (die natürlich auch Axiome enthalten kann) angeben, ist nun die Frage, welche Menge durch diese Regeln definiert wird. Ganz klar sollte die definierte Menge die Regeln respektieren, d. h. wenn alle Voraussetzungen in der Menge liegen, dann auch ihre Folgerung. Eine Menge, die diese Eigenschaft besitzt, nennen wir *abgeschlossen* unter der Regelmenge, oder kurz R-abgeschlossen, wobei R die Menge der Regeln bezeichnet:

Definition 4.4 (Unter einer Regelmenge abgeschlossene Menge)
Sei R eine Menge von Regeln über X. Eine Menge $Q \subseteq X$ heißt *abgeschlossen* unter R (kurz R-abgeschlossen), wenn für jede Regel(instanz) $(Y, x) \in R$ mit $Y \subseteq Q$ auch $x \in Q$ gilt.

Offensichtlich enthält jede R-abgeschlossene Menge all die Elemente, die auf der rechten Seite eines Axioms auftreten. Denn für die linke Seite $\emptyset$ eines Axioms gilt immer $\emptyset \subseteq Q$ und damit muß die rechte Seite x in Q liegen. Die Frage ist nun, ob für eine gegebene Regelmenge R über X überhaupt eine R-abgeschlossene Menge existiert. Falls sie existiert, müssen wir uns überlegen, ob sie eindeutig ist. Die erste Frage ist einfach zu beantworten, denn die Menge X ist trivialerweise immer R-abgeschlossen. Und das beantwortet auch schon fast die zweite Frage: im allgemeinen gibt es mehrere verschiedene R-abgeschlossene Mengen.

Beispiel 4.3 (Abgeschlossene Mengen)
Wir betrachten die Menge $X = \{a, b\}$ und die Regeln $R = \{(\{a\}, b), (\{b\}, a)\}$. Dann sind die beiden Mengen $\emptyset$ und X abgeschlossen unter R. Die leere Menge ist für diese Regeln R auch R-abgeschlossen, da in ihr kein Axiom vorkommt. Es muß also kein Element unbedingt in die Menge aufgenommen werden.
Dagegen sind die beiden Mengen $\{a\}$ und $\{b\}$ nicht R-abgeschlossen, da die Regeln verlangen, daß das jeweils andere Element auch in die Menge gehört.

Man kann sich leicht Beispiele für Regelmengen überlegen, für die noch sehr viel mehr abgeschlossene Mengen existieren. Die Frage ist nun, welche der R-abgeschlossenen Mengen die durch die Regelmenge induktiv definierte Menge sein soll. Die Idee ist, daß wir nur das in die induktiv definierte Menge aufnehmen sollten, was unbedingt nötig ist – und nicht mehr. Wir sollten also die kleinste R-abgeschlossene Menge wählen. Zuvor müssen wir uns jedoch davon überzeugen, daß es diese kleinste R-abgeschlossene Menge überhaupt gibt.

Lemma 4.5 (Existenz der kleinsten R-abgeschlossenen Menge)
Sei R eine Menge von Regeln über X.

1. Sei nun $(Q_i)_{i \in I}$ eine Familie von R-abgeschlossenen Mengen, d. h. für jedes $i \in I$ ist Q_i abgeschlossen unter R. Dann ist auch die Menge $Q = \bigcap_{i \in I} Q_i$ unter R abgeschlossen.

2. Es gibt eine bzgl. Mengeninklusion $\subseteq$ kleinste R-abgeschlossene Menge.

Beweis: Der Beweis von 1. ist einfach. Der Beweis von 2. benutzt 1.

Den Beweis werden wir in der Übung besprechen.

□

Zur Erinnerung: Das kleinste Element einer Menge ist, wenn es existiert, eindeutig.

Da wir nun wissen, daß es für jede Regelmenge eine kleinste R-abgeschlossene Menge gibt, können wir diese als die *induktiv durch R definierte Menge* festlegen.

Definition 4.6 (Induktiv definierte Menge)
Sei R eine Regelmenge über X. Wir nennen die (bzgl. $\subseteq$) kleinste unter R abgeschlossene Menge die *durch R induktiv definierte Menge*. Wir bezeichnen diese Menge mit I_R.

Oft liest man bei induktiven Definitionen den Zusatz „nichts sonst ist in der Menge“. Das ist gemäß der obigen Definition – und dem in der Mathematik üblichen Verständnis von induktiven Definitionen – überflüssig (oder sogar unsinnig). Denn wenn man eine Menge induktive definiert, dann ist die kleinste R-abgeschlossene Menge gemeint; und die enthält keine „überflüssigen“ Elemente.

Beispiel 4.4 (Induktive Definitionen)

1. In Kapitel 3 haben wir bereits einige Beispiele für induktive Definitionen kennen gelernt. Allerdings haben wir dort die Menge X nicht explizit benannt und die Regeln mehr oder weniger explizit angegeben.

   Zur Übung können Sie sich ja mal überlegen, wie die Menge X und die zugehörigen Regelinstanzen aussehen.

2. Sei A eine beliebige Menge und $\rightarrow$ eine binäre Relation über A. Wir definieren nun die folgende Regelmenge über $A \times A$:

$$\begin{aligned} R = \quad & \{(\emptyset, (a,a)) \mid a \in A\} \cup \\ & \{(\emptyset, (a,b)) \mid a \rightarrow b\} \cup \\ & \{(\{(a,b),(b,c)\}, (a,c)) \mid a,b,c \in A\} \end{aligned}$$

   Dann bezeichnet die durch diese Regeln induktiv definierte Menge gerade die reflexiv-transitive Hülle von $\rightarrow$, d. h. $I_R = \rightarrow^*$.

Die Regelinstanzen der ersten Zeile drücken die Reflexivität aus. Die Regelinstanzen der zweiten Zeile drücken aus, daß jeder Übergang von $\rightarrow$ auch zu I_R gehört. Die Regelinstanzen der dritten Zeile drücken die Transitivität aus.

Da I_R die kleinste R-abgeschlossene Menge ist, werden zu I_R gerade die für die reflexiv-transitive Hülle nötigen Übergänge hinzugenommen.

Die Definition 4.6 definiert uns zwar eindeutig die Menge I_R. Sie liefert uns aber kein Verfahren, um an die Elemente dieser Menge heranzukommen. Wir werden nun eine weitere Charakterisierung von I_R angeben, die es uns erlaubt, die Elemente von I_R systematisch zu generieren. Die Idee ist recht einfach: Wir beginnen mit der leeren Menge und nehmen schrittweise die Elemente dazu, die man mit Hilfe der Regeln aus den bisher abgeleiteten Elemente ableiten kann. Im ersten Schritt sind das nur die Folgerungen der Axiome, da die ja keine Voraussetzungen benötigen. Im zweiten Schritt können wir dann schon mehr ableiten. Natürlich kann es sein, daß dieser Iterationsprozeß nie endet. Aber im Laufe des Interationsprozesses kommen nach und nach alle Elemente von I_R dazu.
Für eine Regelmenge R über X sieht diese Iteration wie folgt aus:

$$\begin{array}{lcll} Q_0 & = & \emptyset & \\ Q_1 & = & \{x \in X \mid (\emptyset, x) \in R\} & = \widehat{R}(Q_0) \\ Q_2 & = & \{x \in X \mid (Y, x) \in R, Y \subseteq Q_1\} & = \widehat{R}(Q_1) \\ Q_3 & = & \{x \in X \mid (Y, x) \in R, Y \subseteq Q_2\} & = \widehat{R}(Q_2) \\ \vdots & & & \end{array}$$

Die Menge I_R ergibt sich dann als Vereinigung aller Q_i, d. h. $I_R = \bigcup_{i \in \mathbb{N}} Q_i$. Dabei definiert die Operation $\widehat{R}$ genau einen Ableitungsschritt: $\widehat{R}(Q)$ ist diejenige Menge von Elementen, die man in einem Schritt aus Q ableiten kann.

Definition 4.7 (Ableitungsschritt $\widehat{R}$)
Sei R eine Menge von Regeln über X. Die Abbildung $\widehat{R} : 2^X \rightarrow 2^X$ ist wie folgt definiert:

$$\widehat{R}(Q) = \{x \in X \mid \exists Y \subseteq Q.(Y, x) \in R\}$$

Die Elemente von $\widehat{R}(Q)$ heißen *die in einem Schritt mit R aus Q ableitbaren Elemente*.

Mit Hilfe des $\widehat{R}$-Operators können wir jetzt noch einfacher formulieren, wann eine Menge Q unter R abgeschlossen ist, nämlich genau dann, wenn $\widehat{R}(Q) \subseteq Q$ gilt.

Wir können nun unsere obigen Überlegungen als Satz formulieren:

Satz 4.8
Sei R eine Regelmenge über X und sei die Folge von Teilmengen $Q_0, Q_1, Q_2, \ldots$ wie folgt definiert:

- $Q_0 = \emptyset$
- $Q_{i+1} = \widehat{R}(Q_i)$ für $i \in \mathbb{N}$

Dann gilt $I_R = \bigcup_{i\in\mathbb{N}} Q_i$ und I_R ist ein Fixpunkt von $\widehat{R}$, d. h. $\widehat{R}(I_R) = I_R$.

Beweis: Ausführlich werden wir diesen Satz in den Übungen beweisen. Hier sind die wesentlichen Schritte des Beweises:

1. Der Operator $\widehat{R}$ ist monoton (steigend), d. h. für alle Mengen Q und Q' mit $Q \subseteq Q'$ gilt $\widehat{R}(Q) \subseteq \widehat{R}(Q')$.
2. Die Folge $Q_0, Q_1, Q_2, \ldots$ bildet eine aufsteigende Kette bezüglich $\subseteq$, d. h. für jedes $i \in \mathbb{N}$ gilt $Q_i \subseteq Q_{i+1}$.
3. Die Menge $Q = \bigcup_{i\in\mathbb{N}} Q_i$ ist $\widehat{R}$-abgeschlossen.
4. Für jedes Q_i und jede R-abgeschlossene Menge Q' gilt $Q_i \subseteq Q'$.
5. Q ist die kleinste unter R abgeschlossene Menge.
6. I_R ist Fixpunkt von $\widehat{R}$.

□

Aus der Definition von I_R wissen wir, daß I_R bezüglich Mengeninklusion kleiner ist als jede R-abgeschlossene Menge Q (d. h. als jede Menge mit $\widehat{R}(Q) \subseteq Q$. Insbesondere ist I_R kleiner als jeder Fixpunkt Q von $\widehat{R}$ (d. h. als jede Menge Q mit $\widehat{R}(Q) = Q$). Das heißt, daß I_R der (bezüglich Mengeninklusion) kleinste Fixpunkt von $\widehat{R}$ ist. Wir haben damit also ganz nebenbei gezeigt, daß $\widehat{R}$ immer einen kleinsten Fixpunkt besitzt.

Tatsächlich ist der obige Satz bereits der Fixpunktsatz von Kleene (oder eine speziellen Ausprägung davon). Wir werden diesen Satz später beweisen. Der Beweis des Fixpunktsatzes von Kleene folgt exakt dem gleichen Muster.

Auch wenn der Beweis des Satzes insgesamt recht elementar ist, ist der Satz nicht ganz trivial. Denn wenn wir Regelinstanzen mit unendlich vielen Voraussetzungen zulassen würden, dann würde der Satz in dieser Form nicht gelten. Wer findet ein Gegenbeispiel?

3 Regelinduktion

Im vorangegangen Abschnitt haben wir gesehen, wie man Mengen induktiv definieren kann. Die Frage ist nun, wie man Eigenschaften der Elemente einer induktiv definierten Menge beweisen kann. Dies geht ganz analog zur Vollständigen Induktion. Wir müssen die Eigenschaft für jedes Element beweisen, das aufgrund eines Axioms in die Menge aufgenommen wird. Außerdem müssen wir beweisen, daß für jede Regel die Eigenschaft für die Folgerung (d. h. die rechte Seite der Regel) gilt, wenn die Eigenschaft für alle Voraussetzungen (d. h. alle Elemente auf der linken Seite der Regel) gilt. Dieses Prinzip wird *Induktion über die Regeln* oder kurz *Regelinduktion* genannt.

Wie bei der Noetherschen Induktion können wir den Induktionsanfang im Induktionsschritt „verstecken“, da die Axiome spezielle Regeln ohne Voraussetzung sind.

Prinzip 4.9 (Regelinduktion)
Sei R eine Menge von Regeln über X und P ein Prädikat über X. Wenn für jede Regel $(Y, x) \in R$, für die für jedes $y \in Y$ das Prädikat $P(y)$ gilt, auch das Prädikat $P(x)$ gilt, dann gilt das Prädikat $P(z)$ für jedes $z \in I_R$, d. h. für jedes Element der durch R induktiv definieren Menge.

Wir können das Prinzip der Regelinduktion auch kurz wie folgt formulieren:

$$(\forall (Y, x) \in R.((\forall y \in Y.P(y)) \Rightarrow P(x))) \Rightarrow \forall z \in I_R.P(z)$$

Eine weitere Formulierung ist die folgende:

$$(\forall (Y, x) \in R.(Y \subseteq P \Rightarrow x \in P)) \Rightarrow I_R \subseteq P$$

Beweis: Das Prinzip der Regelinduktion können wir mit Hilfe von Satz 4.8 auf das Prinzip der vollständigen Induktion zurückführen: Sei also R eine Regelmenge, für die für jede Regel $(Y, x) \in R$ mit $Y \subseteq P$ auch $x \in P$ gilt.

1. Aus der Definition des $\widehat{R}$-Operators folgt unmittelbar, daß dann für jede Teilmenge $Q \subseteq P$ auch gilt $\widehat{R}(Q) \subseteq P$.

2. Gemäß Satz 4.8 läßt sich die Menge I_R wie folgt durch die Folge von Mengen $Q_0, Q_1, \ldots$ mit

   - $Q_0 = \emptyset$
   - $Q_{i+1} = \widehat{R}(Q_i)$ für $i \in \mathbb{N}$

   charakterisieren: $I_R = \bigcup_{i \in \mathbb{N}} Q_i$.

3. Offensichtlich gilt $Q_0 = \emptyset \subseteq P$. Durch vollständige Induktion können wir nun unter Anwendung von 1. zeigen, daß für jedes $i \in \mathbb{N}$ gilt $Q_i \subseteq P$. Da jedes einzelne Q_i in P enthalten ist gilt dann auch $(\bigcup_{i \in \mathbb{N}} Q_i) \subseteq P$; also gilt $I_R \subseteq P$.

□

Das Prinzip der Regelinduktion können wir nun anwenden, um Eigenschaften der Elemente einer induktiv definierten Menge zu beweisen. Indirekt zeigen wir auf diese Weise auch, daß bestimmte Elemente nicht in der induktiv definierten Menge vorkommen: nämlich genau die Elemente, die die bewiesene Eigenschaft nicht besitzen. Dazu betrachten wir ein Beispiel.

Beispiel 4.5
Im Beispiel 3.5 in Kapitel 3 auf Seite 36 haben wir bereits informell argumentiert, daß es für die Anweisung $w \equiv$ **while** true **do skip** keine Zustände $\sigma, \sigma' \in \Sigma$ gibt, für die sich $\langle w, \sigma \rangle \to \sigma'$ ableiten läßt. Wir werden dies nun mit Hilfe der Regelinduktion beweisen. Dazu müssen wir uns zunächst ein Prädikat überlegen, das wir mit Hilfe der Regelinduktion beweisen können:

$$P(\langle c, \sigma \rangle \to \sigma') = c \not\equiv w$$

d. h. wir zeigen, daß für jeden Tripel $\langle c, \sigma \rangle \to \sigma'$, den wir aus den Regeln herleiten können, die Anweisung c definitiv nicht unsere Endlosschleife w ist. Wir beweisen die Gültigkeit des Prädikats nun für alle gemäß der Regeln herleitbare Tripel $\langle c, \sigma \rangle \to \sigma'$ durch Induktion über die Regeln:

$\overline{\langle \textbf{skip}, \sigma \rangle \to \sigma}$
Offensichtlich gilt **skip** $\not\equiv w$.

$$\frac{\langle a, \sigma\rangle \to n}{\langle v{:=}a, \sigma\rangle \to \sigma[n/v]}$$

Offensichtlich gilt v:= a $\not\equiv w$.

...

für alle Regeln bis auf die Regeln für die Schleife gilt die Aussage analog. Wir müssen also nur noch die Regeln für die Schleife w betrachten.

$$\frac{\langle \text{true}, \sigma\rangle \to \mathit{false}}{\langle \textbf{while } \text{true } \textbf{do skip } c, \sigma\rangle \to \sigma}$$

Für diese Regel ist die Voraussetzung $\langle \text{true}, \sigma\rangle \to \mathit{false}$ verletzt. Es ist also nichts zu zeigen.

$$\frac{\langle b, \sigma\rangle \to \mathit{true} \quad \langle \textbf{skip}, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } \text{true } \textbf{do skip}, \sigma''\rangle \to \sigma'}{\langle \textbf{while } \text{true } \textbf{do skip}, \sigma\rangle \to \sigma'}$$

Da für die Voraussetzung $\langle \textbf{while } \text{true } \textbf{do skip}, \sigma''\rangle \to \sigma'$ das Prädikat nicht erfüllt ist, müssen wir für diese Regel nichts zeigen.

> *Hier stellen wir das informelle Argument, daß die Konstruktion einer Herleitung für $\langle$**while** true **do skip**$, \sigma''\rangle \to \sigma'$ niemals terminieren würde, vom Kopf auf die Füsse: Wir beweisen den Induktionsschritt für diese Regel, indem wir die Ungültigkeit der Induktionsvoraussetzung für diese Regel zeigen. Das ist sicher etwas ungewöhnlich, aber es ist korrekt.*

4 Herleitungen

Bei der Definition des Begriffs der induktiv durch eine Regelmenge definierten Menge haben wir zur Motivation informell den Begriff der Ableitbarkeit und den Begriff der*Herleitung* benutzt. Insbesondere ist die alternative Charakterisierung der induktiven Mengen in Satz 4.8 durch die schrittweise Ableitbarkeit der Element motiviert.
Jetzt sind wir dazu in der Lage, diesen Begriff formal zu definieren – und zwar durch eine induktive Definition. Wir ziehen uns also fast an den eigenen Haaren aus dem Sumpf. Da wir den Begriff der Herleitung bei der formalen Definition der induktiv definierten Menge nicht benutzt haben, ist unser Vorgehen aber formal sauber.
Formal ist eine Herleitung ein Baum, an dessen Wurzel das hergeleitete Element steht. Die Verzweigungen im Baum entsprechen dabei den Regeln. Um

eine aufwendige graphische Notation zu vermeiden, definieren wir eine Herleitung technisch als ein Paar $(\{d_1, \ldots, d_n\}, x)$, wobei $d_1, \ldots, d_n$ Teilherleitungen sind und x das hergeleitete Element. Um den Aspekt der Regelanwendung besser zum Ausdruck zu bringen benutzen wir anstelle des Kommas den Schrägstrich: $(\{d_1, \ldots d_n\}/x)$. Wenn d eine Herleitung[2] für x ist, schreiben wir $d \vdash_R x$ (gesprochen „d leitet x her“). Dabei lassen wir den Index R meist weg, wenn R aus dem Kontext hervor geht.

Definition 4.10 (Herleitung)
Sei R eine Regelmenge über X. Wir definieren die Relation $\vdash_R$ induktiv durch die folgenden Regeln R':

- Für jedes Axiom $(\emptyset, x) \in R$ ist

$$\frac{}{(\emptyset/x) \vdash_R x}$$

  eine Regel aus R'.

- Für jede Regel $(\{x_1, \ldots, x_n\}, x) \in R$ ist

$$\frac{d_1 \vdash_R x_1 \quad \ldots \quad d_n \vdash_R x_n}{(\{d_1, \ldots, d_n\}/x) \vdash_R x}$$

  eine Regel aus R'.

Wenn $d \vdash_R x$ in der durch R' induktiv definierten Menge liegt, sagen wir, daß d eine *Herleitung* für x ist.

Durch diese Regeln werden die Herleitungsbäume in Form von „Klammergebirgen“ codiert. Für die Definition der Herleitung ist dies praktisch. Wenn wir aber über Herleitungen reden wollen, ist dies eher unpraktisch. Dann benutzen wir die Notation wie wir sie in Kapitel 3 in Abschnitt 3 benutzt haben (z. B. in Beispiel 3.4).
Wenn unsere Definition der induktiven Mengen und der Herleitung vernünftig sind, sollte nun gelten, daß die Elemente der durch die Regeln induktiv definierten Menge genau die Elemente sind, für die eine Herleitung existiert. Formal formulieren wir das wie folgt:

[2]Im Englischen heißt Herleitung *derivation*. Deshalb bezeichnen wir Herleitungen im folgenden mit dem Zeichen d.

Lemma 4.11 (Induktive definierte Menge und Herleitung)
Sei R eine Regelmenge. Dann gilt $x \in I_R$ genau dann, wenn ein d mit $d \vdash_R x$ existiert.

Beweis: Regelinduktion. Ein genauer Beweis wird in der Übung besprochen.

> *Fragen: Über welche Regeln geht die Regelinduktion? Wie genau ist das Prädikat formuliert, für das wir die Regelinduktion anwenden?*

□

Wenn es eine Herleitung d für ein Element x gibt, schreiben wir auch $\vdash_R x$, bzw. wenn R aus dem Kontext hervorgeht auch $\vdash x$. Die Aussagen $x \in I_R$ und $\vdash_R x$ sind dann gleichbedeutend. In der Literatur wird meist $\vdash_R x$ bzw. $\vdash x$ verwendet.
Das vorangegangene Lemma besagt nur, daß es für jedes Element einer induktiv definierten Menge mind. eine Herleitung gibt. Es kann jedoch sein, daß es für manche Elemente mehrere verschiedenen Herleitungen gibt. In der Praxis versucht man aber meist, induktive Definitionen so zu formulieren, daß es eine eindeutige Herleitung für jedes Element gibt. Um das zu formalisieren, formulieren wir nun den Begriff der *eindeutigen induktiven Definition.*

Definition 4.12 (Eindeutige induktive Definition)
Die durch eine Regelmenge R induktive definierte Menge heißt eindeutig induktiv definiert, wenn für jedes $x \in I_R$ genau eine Herleitung d mit $d \vdash_R x$ existiert.

> *Zur Übung sollte Sie sich einmal überlegen, welche der Regelmengen aus Kapitel 3 eindeutige induktive Definitionen sind und welche nicht. Besonders interessant sind die Regeln für das Auswerten der booleschen Ausdrücke.*

Oft werden induktive Definitionen nur dann induktiv genannt, wenn sie eindeutig sind. Insbesondere bei der Definition von syntaktischen Mengen legt man Wert auf die Eindeutigkeit der Definition (vgl. Diskussion zur abstrakten Syntax in Kapitel 3 in Abschnitt 1.2). Um den Begriff der eindeutigen induktiven Definition zu bilden, ist es aber zweckmäßig zunächst den Begriff der induktiven Definition zu bilden, und dann die eindeutigen als Spezialfall zu charakterisieren.
Wenn eine induktive Definition eindeutig ist, kann man (totale) Abbildungen von I_R in irgendeine Menge Y *induktiv über den Aufbau der Menge* I_R definieren. Dazu betrachten wir einige Beispiele von Abbildungen, die wir später noch mehrfach benutzen werden.

Beispiel 4.6 (Definitionen induktiv über den Aufbau einer Menge)

1. Für die Menge der arithmetischen Ausdrücke $Aexp$ definieren wir die Länge induktiv über den Aufbau: $length : Aexp \to \mathbb{N}$ ist definiert durch:

   - $length(n) = 1$
   - $length(v) = 1$
   - $length(a_0 + a_1) = length(a_0) + length(a_0) + 1$
   - $length(a_0 - a_1) = length(a_0) + length(a_0) + 1$
   - $length(a_0 * a_1) = length(a_0) + length(a_0) + 1$

   Formal könnte man die Abbildung durch die folgenden Regeln definieren:

   $$\frac{}{(n,1)} \qquad \frac{}{(v,1)}$$

   $$\frac{(a_0,n_0) \quad (a_1,n_1)}{(a_0{+}a_1, n_0+n_1+1)} \qquad \frac{(a_0,n_0) \quad (a_1,n_1)}{(a_0{-}a_1, n_0+n_1+1)}$$

   $$\frac{(a_0,n_0) \quad (a_1,n_1)}{(a_0{*}a_1, n_0+n_1+1)}$$

   Die dadurch definierte Relation $length \subseteq Aexp \times \mathbb{N}$ ist eine totale Abbildung, da die Definition der arithmetischen eindeutig ist (dies müßte man aber eigentlich beweisen).

2. Die Abbildung $assign : Com \to 2^{\mathbb{V}}$, die jeder Anweisung die Menge derjenigen Variablen zuordnet, an die ein Wert zugewiesen wird, ist induktiv definiert durch:

   - $assign(\textbf{skip}) = \emptyset$
   - $assign(v{:=}\, a) = \{v\}$
   - $assign(c_0;\ c_1) = assign(c_0) \cup assign(c_1)$
   - $assign(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) = assign(c_0) \cup assign(c_1)$
   - $assign(\textbf{while } b \textbf{ do } c) = assign(c)$

   Wir verzichten hier darauf, das Prinzip der Definition einer totalen Abbildung induktiv über den Aufbau der Menge zu formalisieren. Das erste Beispiel sollte einen guten Eindruck davon geben, wie das geht. Die Formalisierung und der Beweis, daß die so definierte Relation für jede eindeutig induktiv definierte Menge eine totale Abbildung ist, ist eine einfache Übungsaufgabe.

Weitere Beispiele für Definitionen induktiv über den Aufbau ist die Auswertung der arithmetischen Ausdrücke und der booleschen Ausdrücke. Die Semantik der Anweisungen dagegen ist *keine* Definition induktiv über den Aufbau! Warum wohl?

5 Zusammenfassung

In diesem Kapitel haben wir die Konzepte der induktiven Definition und des induktiven Beweisens formalisiert, die wir in Kapitel 3 benutzt haben, um die operationale Semantik der Programmiersprache IMP zu definieren. Methodisch hätten wir diese Definitionen vor der Definition der operationalen Semantik einführen müssen.
Aus didaktischen Gründen haben wir die Konzepte erst nach Ihrer Anwendung eingeführt. Generell stellt sich die Frage, ob wir (im Rahmen der Vorlesung Semantik) diese Konzepte formalisieren sollten, oder ob wir diese Konzepte als gemeinsame Pragmatik voraussetzen. Der Hauptgrund, diese Konzepte hier zu formalisieren, ist, daß auf dieser Ebene später deutlich wird, daß die mathematische und die operationale Ebene weit weniger unterschiedlich sind, als man zunächst erwarten würde. Ein weiterer Grund ist, ein Bewußtsein dafür zu schaffen, daß in der Informatik fast überall nur mit Wasser gekocht wird, wobei das Wasser die Konzepte des induktiven Definierens und Beweisens sind.

Kapitel 5

Mathematische Semantik

In diesem Kapitel werden wir nun eine weitere Technik zur Definition der Semantik einer Programmiersprache vorstellen: die mathematische bzw. denotationale Semantik. Außerdem werden wir Techniken kennenlernen, um die Äquivalenz verschiedener Semantiken zu beweisen.

1 Motivation

Bevor wir die mathematische Semantik für die Programmiersprache IMP definieren, betrachten wir zur Motivation nochmal kurz die operationale Semantik von IMP. Durch die Tripel

$$\langle a, \sigma \rangle \to n$$

$$\langle b, \sigma \rangle \to t$$

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

wird einem Ausdruck in einem gegebenen Zustand σ ein Wert zugeordnet. Einer Anweisung c wird für einen gegebenen Zustand σ der Endzustand σ' zugeordnet, in dem die Anweisung terminiert, falls sie terminiert.
Eigentlich haben wir damit nicht den Ausdrücken bzw. den Anweisungen eine Semantik zugeordnet, sondern jeweils nur einem Ausdruck bzw. einer Anweisung zusammen mit einem Zustand σ. Bisher haben wir also Abbildungen der folgenden Struktur

$$(Aexp \times \Sigma) \to \mathbb{Z}$$

$$(Bexp \times \Sigma) \to \mathbb{B}$$

$$(Com \times \Sigma) \rightharpoonup \Sigma$$

definiert. Was wir eigentlich wollen sind Abbildungen der folgenden Struktur:

$$\mathcal{A} : Aexp \to (\Sigma \to \mathbb{Z})$$

$$\mathcal{B} : Bexp \to (\Sigma \to \mathbb{B})$$

$$\mathcal{C} : Com \to (\Sigma \rightharpoonup \Sigma)$$

Dabei soll $\mathcal{A}$ jedem arithmetischen Ausdruck seine Semantik zuordnen: eine Abbildung $\Sigma \to \mathbb{Z}$, die für jeden Zustand den Wert zurückliefert, zu dem der Ausdruck in diesem Zustand ausgewertet wird. Entsprechendes gilt für $\mathcal{B}$ und die booleschen Ausdrücke. Die Abbildung $\mathcal{C}$ ordnet jeder Anweisung c eine partielle Abbildung $(\Sigma \rightharpoonup \Sigma)$ zu, die den Zusammenhang zwischen dem Anfangszustand und dem Endzustand bei Ausführung der Anweisung herstellt.
Natürlich ist es kein Problem, die Abbildungen $\mathcal{A}$, $\mathcal{B}$ und $\mathcal{C}$ mit Hilfe der operationalen Semantik punktweise zu definieren[1]. Beispielsweise können wir die Abbildung $\mathcal{C}$ punktweise wie folgt definieren:

$$\mathcal{C}(c)(\sigma) = \sigma' \quad \text{falls} \quad \langle c, \sigma \rangle \to \sigma'$$

Da die Schreibweise $\mathcal{C}(c)(\sigma)$ etwas gewöhnungsbedürftig ist, und um zu betonen, das $\mathcal{C}(c)$ die Semantik von c bezeichnet, benutzen wir bei der Anwendung der Abbildung $\mathcal{C}$ auf eine Anweisung c die „Semantikklammern“, d. h. wir schreiben $\mathcal{C}[\![c]\!]$ anstelle von $\mathcal{C}(c)$. Dabei ist $\mathcal{C}[\![c]\!]$ eine partielle Abbildung $\Sigma \rightharpoonup \Sigma$.
Auf diese Weise könnten wir also die mathematische Semantik der Ausdrücke und Anweisungen mit Hilfe der operationalen Semantik unmittelbar formulieren. Allerdings geht es bei der Definition der mathematischen Semantik nicht nur darum, die Semantik, d. h. die Abbildungen $\mathcal{A}$, $\mathcal{B}$ und $\mathcal{C}$ zu definieren! Es geht auch darum, wie diese Abbildungen definiert werden. Man möchte nämlich, daß die Semantik eines syntaktischen Konstruktes nur mit Hilfe der Semantik der Teilkonstrukte definiert wird. Beispielsweise können

[1] Die Umwandlung einer Abbildung der Form $(Aexp \times \Sigma) \to \mathbb{Z}$ in die Form $Aexp \to (\Sigma \to \mathbb{Z})$ bezeichnet man nach dem amerikanischen Logiker Haskell Curry auch *Currysierung* bzw. engl. *currying*.

wir die Semantik der sequentiellen Ausführung $c_0;\ c_1$ allein mit Hilfe der Semantik der Teilkonstrukte c_0 und c_1 definieren:

$$\mathcal{C}[\![c_0;\ c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

Dabei ist $\circ$ die Funktionskomposition. Letztendlich wollen wir also die Semantik der Anweisungen induktiv über den Aufbau der Anweisungen definieren; wenn man bei der Definition einer Semantik eines Konstrukts nur auf die Semantik seiner Teilkonstrukte Bezug nehmen muß, nennt man das eine *kompositionale Semantik*.
Das Problem bei der operationalen Semantik ist, daß sie nicht induktiv über den Aufbau der Anweisungen definiert ist, da in der zweiten Regel für die Schleife in der Voraussetzung die Schleife selbst wieder vorkommt. Die Frage ist also, wie sich $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ allein mit Hilfe von $\mathcal{B}[\![b]\!]$ und $\mathcal{C}[\![c]\!]$ definieren läßt. Und damit beschäftigen wir uns im Rest diese Kapitels (insbes. in Abschnitt 3).

2 Semantik für Ausdrücke

Die Definition der Semantik der arithmetischen und booleschen Ausdrücke ist nicht weiter schwierig, da bereits die operationale Semantik kompositional ist. Der Vollständigkeit halber geben wir diese Definitionen hier trotzdem an und nutzen die Gelegenheit, uns an einige Notationen zur Definition von Abbildungen zu gewöhnen.

Definition 5.1 (Mathematische Semantik für arithmetische Ausdrücke) Wir definieren die Abbildung $\mathcal{A} : \mathit{Aexp} \to (\Sigma \to \mathbb{Z})$ induktiv über den Aufbau der arithmetischen Ausdrücke:

- $\mathcal{A}[\![n]\!] = \lambda\sigma \in \Sigma.n$
- $\mathcal{A}[\![v]\!] = \lambda\sigma \in \Sigma.\sigma(v)$
- $\mathcal{A}[\![a_0{+}a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!](\sigma) + \mathcal{A}[\![a_1]\!](\sigma))$
- $\mathcal{A}[\![a_0{-}a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!](\sigma) - \mathcal{A}[\![a_1]\!](\sigma))$
- $\mathcal{A}[\![a_0{*}a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!](\sigma) \cdot \mathcal{A}[\![a_1]\!](\sigma))$

Für einen arithmetischen Ausdruck a nennen wir $\mathcal{A}[\![a]\!]$ die (mathematische) Semantik von a.

In der vorangegangenen Definition haben wir die „Lambda"-Notation benutzt, um die Abbildungen $\mathcal{A}[\![a]\!]$ kompakt zu definieren. Wir hätten diese Abbildungen auch wie folgt durch unsere Relationsschreibweise definieren können. Allerdings hätten wir dann – streng genommen – beweisen müssen, daß die derartig definierten Relationen totale Abbildungen sind.

- $\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$
- $\mathcal{A}[\![v]\!] = \{(\sigma, \sigma(v)) \mid \sigma \in \Sigma\}$
- $\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!], (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$
- $\mathcal{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!], (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$
- $\mathcal{A}[\![a_0 * a_1]\!] = \{(\sigma, n_0 \cdot n_1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!], (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$

Die Semantik für den Ausdruck $3 + (5 * \mathrm{x})$ ergibt sich dann beispielsweise wie folgt:

- $\mathcal{A}[\![3]\!] = \lambda\sigma \in \Sigma.3$
- $\mathcal{A}[\![5]\!] = \lambda\sigma \in \Sigma.5$
- $\mathcal{A}[\![\mathrm{x}]\!] = \lambda\sigma \in \Sigma.\sigma(\mathrm{x})$
- $\mathcal{A}[\![5 * \mathrm{x}]\!] = \lambda\sigma \in \Sigma.((\lambda\sigma \in \Sigma.5)(\sigma) \cdot (\lambda\sigma \in \Sigma.\sigma(\mathrm{x}))(\sigma))$
  Diesen λ-Ausdruck können wir vereinfachen zu:
  $\lambda\sigma \in \Sigma.5 \cdot \sigma(\mathrm{x})$
- $\mathcal{A}[\![3 + (5 * \mathrm{x})]\!] = \lambda\sigma \in \Sigma.((\lambda\sigma \in \Sigma.3)(\sigma) + (\lambda\sigma \in \Sigma.(5 \cdot \sigma(\mathrm{x})))(\sigma))$
  Diesen Ausdruck können wir vereinfachen zu:
  $\lambda\sigma \in \Sigma.3 + (5 \cdot \sigma(\mathrm{x}))$

Also gilt $\mathcal{A}[\![3 + (5 * \mathrm{x})]\!] = \lambda\sigma \in \Sigma.(3 + (5 \cdot \sigma(\mathrm{x})))$. Auf einen konkreten Zustand σ mit beispielsweise $\sigma(\mathrm{x}) = 7$ können wir dann die Abbildung $\mathcal{A}[\![3 + (5 * \mathrm{x})]\!]$ wie folgt anwenden: $\mathcal{A}[\![3 + (5 * \mathrm{x})]\!](\sigma) = (\lambda\sigma \in \Sigma.(3 + (5 \cdot \sigma(\mathrm{x}))))(\sigma) = 3 + (5 \cdot \sigma(\mathrm{x})) = 3 + (5 \cdot 7) = 38$.
Entsprechend können wir nun die mathematische Semantik für boolesche Ausdrücke definieren, wobei wir davon ausgehen, daß die Relationen und Operationen $=$, $\leq$, $\neg$, $\wedge$ und $\vee$ semantisch bereits definiert sind:

Definition 5.2 (Mathematische Semantik für boolesche Ausdrücke) Wir definieren die Abbildung $\mathcal{B} : Bexp \to \mathbb{B}$ induktiv über den Aufbau der booleschen Ausdrücke:

- $\mathcal{B}[\![\text{true}]\!] = \lambda\sigma \in \Sigma.\mathit{true}$
- $\mathcal{B}[\![\text{ false }]\!] = \lambda\sigma \in \Sigma.\mathit{false}$
- $\mathcal{B}[\![a_0{=}a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!](\sigma) = \mathcal{A}[\![a_1]\!](\sigma))$

  Wir könnten diese Abbildung auch wie folgt mit Hilfe der Relationenschreibeweise definieren:
  $\{(\sigma, \text{true}) \mid \sigma \in \Sigma \wedge \mathcal{A}[\![a_0]\!](\sigma) = \mathcal{A}[\![a_1]\!](\sigma)\} \cup$
  $\{(\sigma, \text{false}) \mid \sigma \in \Sigma \wedge \mathcal{A}[\![a_0]\!](\sigma) \neq \mathcal{A}[\![a_1]\!](\sigma)\}$
- $\mathcal{B}[\![a_0{\leq}a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!](\sigma) \leq \mathcal{A}[\![a_1]\!](\sigma))$
- $\mathcal{B}[\![\neg\ b]\!] = \lambda\sigma \in \Sigma.(\neg\mathcal{B}[\![b]\!](\sigma))$
- $\mathcal{B}[\![b_0{\wedge}b_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{B}[\![b_0]\!](\sigma) \wedge \mathcal{B}[\![b_1]\!](\sigma))$
- $\mathcal{B}[\![b_0{\vee}b_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{B}[\![b_0]\!](\sigma) \vee \mathcal{B}[\![b_1]\!](\sigma))$

Für einen booleschen Ausdruck b nennen wir $\mathcal{B}[\![b]\!]$ die (mathematische) Semantik von b.

Wir haben nun zwei verschiedene Semantiken für Ausdrücke definiert: die operationale Semantik und die mathematische Semantik. Natürlich sollten diese Semantiken im Ergebnis übereinstimmen. Da beide ganz analog definiert sind, kann man das auch recht einfach beweisen.

Lemma 5.3 (Äquivalenz der Semantiken für Ausdrücke)

1. Für jeden arithmetischen Ausdruck a, jeden Zustand σ und jede Zahl n gilt
$$\langle a, \sigma\rangle \to n \quad \text{gdw.} \quad \mathcal{A}[\![a]\!](\sigma) = n$$
2. Für jeden booleschen Ausdruck b, jeden Zustand σ und jeden Wahrheitswert t gilt
$$\langle b, \sigma\rangle \to t \quad \text{gdw.} \quad \mathcal{B}[\![b]\!](\sigma) = t$$

Beweis: Induktion über den induktiven Aufbau der Ausdrücke. Im Detail ist der Beweis lang aber langweilig. □

3 Semantik für Anweisungen

In diesem Abschnitt definieren wir die mathematische Semantik für Anweisungen. Bevor wir die Semantik formal definieren, sollten wir uns zunächst erst einmal klar machen, worin das Problem besteht. Dazu beginnen wir mit einer naiven Definition der Semantik – und fallen am Ende auf die Nase. Für jede Anweisung c definieren wir dazu induktiv über den Aufbau die partielle Abbildung $\mathcal{C}[\![c]\!] : \Sigma \rightharpoonup \Sigma$, wobei $\mathcal{C}[\![c]\!](\sigma) = \sigma'$ bedeutet, daß die Anweisung c im Zustand σ' terminiert, wenn sie im Zustand σ gestartet wird:

- $\mathcal{C}[\![\textbf{skip}]\!] = id_\Sigma = \lambda\sigma \in \Sigma.\sigma$
- $\mathcal{C}[\![v{:=}\ a]\!] = \lambda\sigma \in \Sigma.\sigma[\mathcal{A}[\![a]\!](\sigma)/v]$
- $\mathcal{C}[\![c_0;\ c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$

  Zur Erinnerung die Funktionskomposition $f \circ g$ ist bei uns definiert durch $(f \circ g)(x) = f(g(x))$.

- $\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] = \lambda\sigma \in \Sigma. \begin{cases} \mathcal{C}[\![c_0]\!](\sigma) & \text{falls } \mathcal{B}[\![b]\!](\sigma) = \mathit{true} \\ \mathcal{C}[\![c_1]\!](\sigma) & \text{falls } \mathcal{B}[\![b]\!](\sigma) = \mathit{false} \end{cases}$

  Wir könnten auch schreiben $\mathcal{C}[\![\textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ c_0\ \textbf{\textit{else}}\ c_1]\!] = \{(\sigma,\sigma') \in \Sigma \times \Sigma \mid \mathcal{B}[\![b]\!](\sigma) = \text{true} \wedge (\sigma,\sigma') \in \mathcal{C}[\![c_0]\!]\} \cup \{(\sigma,\sigma') \in \Sigma \times \Sigma \mid \mathcal{B}[\![b]\!](\sigma) = \text{false} \wedge (\sigma,\sigma') \in \mathcal{C}[\![c_1]\!]\}$ Beide Definitionen bedeuten dasselbe. Welche Notation wir wählen ist eine Frage der Gewöhnung und eine Frage der Adäquatheit. In der ersten Variante sieht man jedoch sofort, daß es sich um die Definition einer partiellen Abbildung handelt.

- Aus der Definition der operationalen Semantik wissen wir bereits, daß gilt: **while** b **do** c $\sim$ **if** b **then** $\ulcorner c;$ **while** b **do** $c \lrcorner$ **else skip**. Diese Äquivalenz können wir nun zur Definition der Semantik der Anweisung **while** b **do** c ausnutzen. Wir definieren:
  $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \mathcal{C}[\![\textbf{if } b \textbf{ then } \ulcorner c;\ \textbf{while } b \textbf{ do} c \lrcorner \textbf{ else skip}]\!] =$
  $\lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \mathcal{B}[\![b]\!](\sigma) = \mathit{false} \\ \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\mathcal{C}[\![c]\!](\sigma)) & \text{falls } \mathcal{B}[\![b]\!](\sigma) = \mathit{true} \end{cases}$

Soweit der naive Versuch, die mathematische Semantik $\mathcal{C}$ für Anweisungen zu definieren. Leider hat diese Definition einen massiven Fehler; denn in der Definition von $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ taucht bereits $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ auf. Deshalb ist der obige „Versuch einer Definition“ keine mathematische Definition. Denn

in einer mathematischen Definition darf der zu definierende Begriff nicht benutzt, werden um den Begriff zu bilden. Eine solche Definition (auch wenn man sie manchmal sieht) ist falsch; noch schlimmer: sie ist nicht einmal eine Definition. Im Rest dieses Abschnitts werden wir uns nun darum bemühen, wie wir diese Selbstbezüglichkeit in der Definition von $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ los werden.

Eine ähnliche Selbstbezüglichkeit haben wir bereits bei der Definition der operationalen Semantik für die Schleife gesehen. Denn in der Regel für die Schleife tritt dieselbe Anweisung in der Voraussetzung auf. Allerdings konnten wir solchen Regeln trotzdem die durch sie definierte induktiv definierte Semantik zuordnen. Daß das wirklich gut geht, haben wir uns in Kapitel 4 überlegt. Bei der Definition der mathematischen Semantik ist die Lösung dieses Problems scheinbar[2] etwas schwieriger als bei der operationalen Semantik, da wir die mathematische Semantik einer Anweisung c immer am Stück definieren müssen und die Abbildung $\mathcal{C}[\![c]\!]$ nicht punktweise für jeden Zustand σ definieren dürfen, wie wir das bei der operationalen Semantik getan haben.

Um uns der Lösung des Problems zu nähern, betrachten wir das Problem der obigen Definition auf einer etwas abstrakteren Ebene. Der Fehler, den wir in der Definition gemacht haben besteht darin, daß wir ein mathematisches Objekt x (im obigen Beispiel eine Abbildung) definiert haben, wobei x in seiner eigenen Definition vorkam. Kurz gesagt haben wir geschrieben $x = f(x)$. Viele Informatiker haben mit einer solchen Definition gar kein Problem. Denn sie meinen, daß man diese Definition „rekursiv“ interpretieren kann. Das ist aber bei einer mathematischen Definition nicht zulässig. Abgesehen davon, daß die Mathematik kein „eingebautes Konzept“ von Rekursion besitzt (eine solche Definition ist und bleibt in der Mathematik Unsinn), sind wir ja gerade dabei, die mathematischen Grundlagen der Semantik und damit auch die Grundlagen für die Rekursion zu legen – und dazu dürfen wir natürlich das Konzept der Rekursion nicht benutzen. Deshalb müssen wir dieses Problem auf andere Weise lösen.

Tatsächlich liegt die Lösung des Problems schon fast auf der Hand. Denn wenn die Abbildung f definiert ist, sind damit auch all diejenigen Elemente x definiert für die gilt $x = f(x)$: die *Fixpunkte von* f. Die Interpretation der

[2]Wenn wir unsere Lösung am Ende nochmal ansehen, werden wir feststellen, daß es genau dieselbe Lösung ist wie bei der operationalen Semantik – sie ist nur etwas anders verpackt.

„Definition“ $x = f(x)$ könnte also sein, daß wir dasjenige x meinen, das die Gleichung $x = f(x)$ löst, d. h. ein Fixpunkt von f ist. Allerdings gibt es bei dieser Interpretation noch zwei Probleme:

1. Es kann sein, daß f gar keinen Fixpunkt besitzt. Das gilt zum Beispiel für $\lambda x \in \mathbb{N}.x+1$ oder für $\lambda x \in \mathbb{B}.\neg x$. Dann existiert das durch $x = f(x)$ definierte Objekt nicht.

2. Es könnte auch sein, daß f mehrere Fixpunkte besitzt. Das gilt zum Beispiel für die Identitätsfunktion oder auch für die Abbildung $\lambda x \in \mathbb{N}.x \cdot x$.

In beiden Fällen dürfen wir nicht über „dasjenige x“ reden, für das $x = f(x)$ gilt. Die Formulierung „dasjenige x welches“ dürfen wir nur benutzen, wenn wir wissen, daß das bezeichnete Objekt existiert und eindeutig ist[3]. Wenn es mehrere Fixpunkte gibt, kann man allerdings versuchen, unter allen Fixpunkten einen auszuzeichnen – beispielsweise den kleinsten bzgl. einer Ordnung. Dann muß man aber zuvor zeigen, daß der kleinste Fixpunkt existiert[4].

Nach diesen allgemeinen Vorüberlegungen betrachten wir die obige Definition von $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ etwas genauer. Offensichtlich geht in diese Definition die Semantik der Bedingung b, d. h. $\mathcal{B}[\![b]\!]$, die Semantik des Schleifenrumpfes c , d. h. $\mathcal{C}[\![c]\!]$, und die Semantik der Schleife selbst ein. Wir ersetzen nun $\mathcal{B}[\![b]\!]$ durch β, $\mathcal{C}[\![c]\!]$ durch γ. Für die Semantik der Schleife selbst schreiben wir ξ, weil das die Unbekannte ist, nach der wir noch suchen. Mit diesen Bezeichnungen sieht unsere obige Gleichung wie folgt aus:

- $\xi = \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \xi(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases}$

Die rechte Seite dieser Gleichung können wir als eine Abbildung $\mathcal{F}$ in den drei Parametern β, γ und ξ auffassen. Die Gleichung können wir dann kompakt schreiben als $\xi = \mathcal{F}(\beta, \gamma, \xi)$. Da wir zunächst β und γ festlegen und dann eine Lösung der Gleichung für ξ suchen, verbannen wir die Parameter β und γ in den Index der Abbildung und schreiben $\mathcal{F}_{\beta,\gamma}(\xi)$ anstelle von $\mathcal{F}(\beta, \gamma, \xi)$. Dabei bildet $\mathcal{F}_{\beta,\gamma}$ eine eine partielle Abbildung $\Sigma \rightharpoonup \Sigma$ auf eine partielle Abbildung $\Sigma \rightharpoonup \Sigma$ ab.

[3]Dieser und viele weitere Tips zum mathematischen Formulieren von Gedanken finden sich in dem sehr empfehlenswerten Buch von Beutelspacher [3].

[4]Zur Erinnerung: Eindeutig ist der kleinste Fixpunkt per Definition.

Definition 5.4 (Das Schleifenfunktional $\mathcal{F}_{\beta,\gamma}$)
Für zwei Abbildungen $\beta : \Sigma \to \mathbb{B}$ und $\gamma : \Sigma \rightharpoonup \Sigma$ definieren wir die Abbildung $\mathcal{F}_{\beta,\gamma} : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ wie folgt:

$$\mathcal{F}_{\beta,\gamma}(\xi) = \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \xi(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases}$$

Die Abbildung $\mathcal{F}_{\beta,\gamma}$ nennen wir das *Schleifenfunktional*[5].

Noch eleganter – aber dafür etwas gewöhnungsbedürftiger – hätten wir schreiben können:

$$\mathcal{F}_{\beta,\gamma} = \lambda\xi \in (\Sigma \rightharpoonup \Sigma).\lambda\sigma \in \Sigma. \begin{cases} \sigma & \mathit{falls}\ \beta(\sigma) = \text{false} \\ \xi(\gamma(\sigma)) & \mathit{falls}\ \beta(\sigma) = \text{true} \end{cases}$$

Schön wäre es nun, wenn $\mathcal{F}_{\beta,\gamma}$ für jedes β und γ einen eindeutigen Fixpunkt besitzen würde, d. h. wenn genau ein ξ mit $\mathcal{F}_{\beta,\gamma}(\xi) = \xi$ existieren würde. Denn dann würde durch diese Gleichung das ξ eindeutig definiert. Wir müssen dazu der Reihe nach die folgenden Fragen klären:

1. Gibt es überhaupt ein $\xi : \Sigma \rightharpoonup \Sigma$ mit $\mathcal{F}_{\beta,\gamma}(\xi) = \xi$, d. h. besitzt $\mathcal{F}_{\beta,\gamma}$ (wenigstens) einen Fixpunkt.

   Die Antwort wird ja sein.

2. Wenn $\mathcal{F}_{\beta,\gamma}$ einen Fixpunkt besitzt: Ist dieser Fixpunkt eindeutig? D. h. gilt für alle $\xi, \xi' \in (\Sigma \rightharpoonup \Sigma)$ mit $\mathcal{F}_{\beta,\gamma}(\xi) = \xi$ und $\mathcal{F}_{\beta,\gamma}(\xi') = \xi'$ gilt $\xi = \xi'$?

   Die Antwort wird im allgemeinen nein sein. Für manche β und γ besitzt $\mathcal{F}_{\beta,\gamma}$ mehrere verschiedene Fixpunkte.

3. Wenn der Fixpunkt nicht eindeutig ist: Können wir einen bestimmten Fixpunkt unter allen Fixpunkten auszeichnen, der dann eindeutig ist?

   Wir werden unter allen Fixpunkten den bzgl. Mengeninklusion kleinsten auszeichnen und zeigen, daß er existiert.

4. Wenn wir einen Fixpunkt eindeutigen auszeichnen können: Paßt dieser Fixpunkt zur operationalen Semantik der Schleife?

[5]Um hervorzuheben, daß diese Abbildung eine (partielle) Funktion auf eine (partielle) Funktion abbildet, nennen wir die Abbildung ein Funktional.

Wir werden sehen, daß der kleinste Fixpunkt genau zur operationalen Semantik der Schleife paßt.

Bevor wir uns jedoch diesen Fragen zuwenden, werden wir uns durch die Betrachtung von einigen Beispielen noch mehr Verständnis für das Schleifenfunktional verschaffen.

Beispiel 5.1 (Schleifen und Schleifenfunktional)

1. Wir wählen $w \equiv$ **while** true **do skip**, d. h. $\beta = \mathcal{B}[\![\text{true}]\!] = \lambda\sigma \in \Sigma.\mathit{true}$ und $\gamma = \mathcal{C}[\![\textbf{skip}]\!] = id_\Sigma$. Dann gilt

$$\mathcal{F}_{\beta,\gamma}(\xi) = \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \xi(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases}$$

   Wir sehen sofort, daß gilt $\mathcal{F}_{\beta,\gamma}(\xi) = \xi$.

   D. h. für diese spezielle Wahl von β und γ ist das Schleifenfunktional $\mathcal{F}_{\beta,\gamma}$ die identische Abbildung, d. h. jede partielle Abbildung $\xi : \Sigma \rightharpoonup \Sigma$ ist ein Fixpunkt von $\mathcal{F}_{\beta,\gamma}$. Die kleinste partielle Abbildung ist die überall undefinierte Abbildung: $\Omega = \emptyset$. Und das entspricht genau der Semantik der Schleife w, da diese Schleife für keinen Anfangszustand terminiert.

2. Wir wählen nun $w \equiv$ **while** $\neg$x=0 **do** x:= x−1, d. h. $\beta = \mathcal{B}[\![\neg\,\text{x=0}]\!] = \lambda\sigma \in \Sigma.\sigma(x) \neq 0$ und $\gamma = \mathcal{C}[\![\text{x:= x−1}]\!] = \lambda\sigma \in \Sigma.\sigma[\sigma(x) - 1/x]$. Dann folgt aus

$$\mathcal{F}_{\beta,\gamma}(\xi) = \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \xi(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases}$$

   unter Anwendung der Definition von β und γ sofort

$$\mathcal{F}_{\beta,\gamma}(\xi) = \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \sigma(\text{x}) = 0 \\ \xi(\sigma[\sigma(x) - 1/x]) & \text{falls } \sigma(\text{x}) \neq 0 \end{cases}$$

   Die Frage ist nun, ob auch dieses $\mathcal{F}_{\beta,\gamma}$ einen Fixpunkt besitzt? Die Antwort ist ja. Und auch in diesem Falle gibt es mehrere Fixpunkte. Die Abbildung $f : \Sigma \rightharpoonup \Sigma$ mit

$$f(\sigma) = \begin{cases} \sigma[0/x] & \text{falls } \sigma(x) \geq 0 \\ \mathit{undef}. & \text{sonst} \end{cases}$$

   ist ein Fixpunkt von $\mathcal{F}_{\beta,\gamma}$.

Daß f ein Fixpunkt von $\mathcal{F}_{\beta,\gamma}$ ist, kann man leicht nachrechnen:

$$\begin{aligned}
\mathcal{F}_{\beta,\gamma}(f) &= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \mathit{falls}\ \sigma(x) = 0 \\ f(\sigma[\sigma(x) - 1/x]) & \mathit{falls}\ \sigma(x) \neq 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \mathit{falls}\ \sigma(x) = 0 \\ f(\sigma[\sigma(x) - 1/x]) & \mathit{falls}\ \sigma(x) > 0 \\ f(\sigma[\sigma(x) - 1/x]) & \mathit{falls}\ \sigma(x) < 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \mathit{falls}\ \sigma(x) = 0 \\ (\sigma[\sigma(x) - 1/x])[0/x] & \mathit{falls}\ \sigma(x) > 0 \\ \mathit{undef}. & \mathit{falls}\ \sigma(x) < 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \mathit{falls}\ \sigma(x) = 0 \\ \sigma[0/x] & \mathit{falls}\ \sigma(x) > 0 \\ \mathit{undef}. & \mathit{falls}\ \sigma(x) < 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \mathit{falls}\ \sigma(x) \geq 0 \\ \mathit{undef}. & \mathit{falls}\ \sigma(x) < 0 \end{cases} \\
&= f
\end{aligned}$$

Es gibt jedoch viele weitere Fixpunkte. Für jeden Zustand $\widehat{\sigma}$ ist die Abbildung $g : \Sigma \rightharpoonup \Sigma$ mit

$$g(\sigma) = \begin{cases} \sigma[0/x] & \text{falls } \sigma(x) \geq 0 \\ \widehat{\sigma} & \text{sonst} \end{cases}$$

ein Fixpunkt von $\mathcal{F}_{\beta,\gamma}$.

Daß g ein Fixpunkt von $\mathcal{F}_{\beta,\gamma}$ ist kann man auch ganz leicht nachrech-

nen:

$$\begin{aligned}
\mathcal{F}_{\beta,\gamma}(g) &= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \textit{falls } \sigma(x) = 0 \\ g(\sigma[\sigma(x) - 1/x]) & \textit{falls } \sigma(x) \neq 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \textit{falls } \sigma(x) = 0 \\ g(\sigma[\sigma(x) - 1/x]) & \textit{falls } \sigma(x) > 0 \\ g(\sigma[\sigma(x) - 1/x]) & \textit{falls } \sigma(x) < 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \textit{falls } \sigma(x) = 0 \\ (\sigma[\sigma(x) - 1/x])[0/x] & \textit{falls } \sigma(x) > 0 \\ \widehat{\sigma} & \textit{falls } \sigma(x) < 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \textit{falls } \sigma(x) = 0 \\ \sigma[0/x] & \textit{falls } \sigma(x) > 0 \\ \widehat{\sigma} & \textit{falls } \sigma(x) < 0 \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma[0/x] & \textit{falls } \sigma(x) \geq 0 \\ \widehat{\sigma} & \textit{falls } \sigma(x) < 0 \end{cases} \\
&= g
\end{aligned}$$

Auch hier ist wieder die Frage, welche der Abbildungen f und g der operationalen Semantik der Schleife entspricht. In diesem Falle ist es f, die „undefiniertere“, also die bzgl. Mengeninklusion kleinere, Abbildung.

> *$f \subseteq g$ auf partiellen Abbildungen bedeutet, daß f an weniger Stellen als g definiert ist, und daß die Abbildungen übereinstimmen, wenn beide definiert sind, d. h. für jedes x gilt $f(x)$ ist undefiniert oder es gilt $f(x) = g(x)$.*

Die Beispiele zeigen also, daß der kleinste Fixpunkt des Funktionals $\mathcal{F}_{\beta,\gamma}$ genau die Semantik der Schleife ergeben. Daß das so ist, werden wir später noch beweisen. Intuitiv läßt sich das wie folgt begründen: Der kleinste Fixpunkt ist nur für solche Zustände definiert, für die das durch das Funktional $\mathcal{F}_{\beta,\gamma}$ unbedingt gefordert ist; für alle anderen Zustände ist der Fixpunkt undefiniert, was der Nichtterminierung entspricht. Größere Fixpunkte „denken sich für diese Fälle einen mehr oder weniger beliebigen Wert aus“ (vgl. die Abbildung g im obigen Beispiel), was natürlich nicht dem operationalen Verhalten der Schleife entspricht. Im kleinste Fixpunkt gibt es diese Beliebigkeit nicht, weshalb er genau dem operationalen Verhalten entspricht. Zunächst begnügen wir uns damit, zu zeigen daß $\mathcal{F}_{\beta,\gamma}$ immer einen kleinsten Fixpunkt besitzt.

Satz 5.5 (Schleifenfunktional besitzt kleinsten Fixpunkt)
Für jede Abbildung $\beta : \Sigma \to \mathbb{B}$ und $\gamma : \Sigma \rightharpoonup \Sigma$ besitzt die Abbildung $\mathcal{F}_{\beta,\gamma}$ einen bezüglich Mengeninklusion kleinsten Fixpunkt.

Beweis: Wir definieren die folgende Regelmenge R über $\Sigma \times \Sigma$:

$$\begin{aligned} R = \quad & \{(\emptyset/(\sigma,\sigma)) \mid \beta(\sigma) = \mathit{false}\} \cup \\ & \{(\{(\sigma'',\sigma')\}/(\sigma,\sigma')) \mid \beta(\sigma) = \mathit{true}, \gamma(\sigma) = \sigma''\} \end{aligned}$$

Gemäß Satz 4.8 besitzt nun $\widehat{R}$ einen kleinsten Fixpunkt $Q \subseteq \Sigma \times \Sigma$. Wir zeigen nun, daß Q sogar eine partielle Abbildung ist, d. h. daß gilt $Q \in \Sigma \rightharpoonup \Sigma$.
Gemäß Satz 4.8 können wir Q durch die Folge von $Q_0 \subseteq Q_1 \subseteq Q_2 \subseteq \ldots$ mit $Q_0 = \emptyset$ und $Q_{i+1} = \widehat{R}(Q_i)$ charakterisieren: $Q = \bigcup_{i \in \mathbb{N}} Q_i$. Um zu zeigen, daß Q eine partielle Abbildung ist, reicht es also zu zeigen, daß jedes Q_i eine partielle Abbildung ist. Wir beweisen dies durch vollständige Induktion:

Induktionsanfang: Offensichtlich gilt $Q_0 = \emptyset \in (\Sigma \rightharpoonup \Sigma)$.

Induktionsschritt: Wir nehmen nun an, daß Q_i eine partielle Abbildung ist, d. h. $Q_i \in (\Sigma \rightharpoonup \Sigma)$ und zeigen, daß auch Q_{i+1} eine partielle Abbildung ist.

Gemäß Definition gilt

$$\begin{aligned} Q_{i+1} = \widehat{R}(Q_i) = \quad & \{(\sigma,\sigma) \mid \beta(\sigma) = \mathit{false}\} \cup \\ & \{(\sigma,\sigma') \mid \beta(\sigma) = \mathit{true}, (\sigma'',\sigma') \in Q_i, \gamma(\sigma) = \sigma''\} \end{aligned}$$

Da Q_i eine partielle Abbildung ist, können wir dies umformulieren zu

$$\begin{aligned} Q_{i+1} = \quad & \{(\sigma,\sigma) \mid \beta(\sigma) = \mathit{false}\} \cup \\ & \{(\sigma,\sigma') \mid \beta(\sigma) = \mathit{true}, Q_i(\gamma(\sigma)) = \sigma'\} \end{aligned}$$

Dies wiederum können wir umschreiben zu

$$Q_{i+1} = \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ Q_i(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases}$$

also einer partiellen Abbildung.

Wir wissen nun also, daß der kleinste Fixpunkt Q von $\widehat{R}$ eine partielle Abbildung auf Σ ist. Die Regeln R haben wir nun aber genau so gewählt, daß für jede partielle Abbildung $f : \Sigma \rightharpoonup \Sigma$ gilt $\widehat{R}(f) = \mathcal{F}_{\beta,\gamma}(f)$. Dementsprechend ist die partielle Abbildung Q auch der kleinste Fixpunkt von $\mathcal{F}_{\beta,\gamma}$. □

Wir haben diesen Beweis mit Hilfe des Umwegs über Regeln und die durch sie induktiv definierte Menge geführt. Das hat zwei Gründe: Erstens zeigt dieses Vorgehen, daß der Zusammenhang zwischen der mathematischen Semantik und der operationalen Semantik enger ist, als man oft meint. Zweitens können wir den Beweis hier führen, ohne bereits die Fixpunkttheorie in ihrer vollen Allgemeinheit eingeführt zu haben. Das werden wir später nachholen – damit erübrigt sich dann auch der obige Beweis.

Da wir nun wissen, daß der kleinste Fixpunkt von $\mathcal{F}_{\beta,\gamma}$ immer existiert, führen wir eine Bezeichnung für ihn ein (später werden wir diese Bezeichnung noch verallgemeinern).

Definition 5.6 (Kleinster Fixpunkt)
Seien $\beta : \Sigma \to \mathbb{B}$ und $\gamma : \Sigma \rightharpoonup \Sigma$ zwei Abbildungen. Den (bezüglich Mengeninklusion) kleinsten Fixpunkt von $\mathcal{F}_{\beta,\gamma}$ bezeichnen wir mit $\mathit{fix}(\mathcal{F}_{\beta,\gamma})$.

Oft schreibt man auch $\operatorname{fix}\mathcal{F}_{\beta,\gamma}$ *anstelle von* $\operatorname{fix}(\mathcal{F}_{\beta,\gamma})$.

Jetzt haben wir das mathematische Handwerkszeug bereitgestellt, um die mathematische Semantik für die Anweisungen induktiv über den Aufbau der Anweisungen definieren zu können:

Definition 5.7 (Mathematische Semantik für Anweisungen)
Wir definieren die Abbildung $\mathcal{C} : \mathit{Com} \to (\Sigma \rightharpoonup \Sigma)$ induktiv über den Aufbau der Anweisungen:

- $\mathcal{C}[\![\textbf{skip}]\!] = id_\Sigma$
- $\mathcal{C}[\![v := a]\!] = \lambda\sigma \in \Sigma.\sigma[\mathcal{A}[\![a]\!](\sigma)/v]$
- $\mathcal{C}[\![c_0;\ c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$
- $\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] = \lambda\sigma \in \Sigma.\begin{cases} \mathcal{C}[\![c_0]\!](\sigma) & \text{falls } \mathcal{B}[\![b]\!](\sigma) = \mathit{true} \\ \mathcal{C}[\![c_1]\!](\sigma) & \text{falls } \mathcal{B}[\![b]\!](\sigma) = \mathit{false} \end{cases}$
- $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \mathit{fix}(\mathcal{F}_{\mathcal{B}[\![b]\!],\mathcal{C}[\![c]\!]})$

Für eine Anweisung c nennen wir $\mathcal{C}[\![c]\!]$ die (mathematische) Semantik von c.

Durch den Trick mit dem Fixpunkt haben wir also das Problem mit der Selbstbezüglichkeit der Definition aufgelöst. Die obige Definition ist mathematisch sauber und eindeutig formuliert. Insbesondere ist die Definition *kompositional*, da sich die Semantik jedes syntaktischen Konstruktes allein mit

Hilfe der Semantik der Teilkonstrukte definieren läßt. Wir müssen uns jetzt „nur noch“ vergewissern, daß es auch die richtige Definition ist, d. h. daß die mathematische Semantik genau zum gleichen Ergebnis führt wie die operationale Semantik.

4 Betrachtungen zum kleinsten Fixpunkt

Um noch etwas mehr Verständnis für die Wahl des kleinsten Fixpunktes von $\mathcal{F}_{\beta,\gamma}$ als Semantik für eine Schleife zu entwickeln, betrachten wir ihn in diesem Abschnitt nochmal etwas genauer. Im Beweis, daß der kleinste Fixpunkt von $\mathcal{F}_{\beta,\gamma}$ existiert (Satz 5.5) haben wir die folgende Folge von partiellen Abbildungen definiert[6]:

- $f_0 = \Omega$, d. h. f_0 ist die überall undefinierte Abbildung.

- $f_{i+1} = \mathcal{F}_{\beta,\gamma}(f_i)$, d. h. f_{i+1} ergibt sich durch einmalige Anwendung des Funktion $\mathcal{F}_{\beta,\gamma}$ auf f_i.

Der kleinste Fixpunkt f von $\mathcal{F}_{\beta,\gamma}$ ist dann die Vereinigung aller f_i, d. h. $f = \bigcup_{i\in\mathbb{N}} f_i$. Die Abbildungen f_i schauen wir uns nun etwas genauer an:

[6]In dem Beweis haben wir die Abbildungen f_i mit Q_i bezeichnet, da noch zu zeigen war, daß alle Q_i partielle Abbildungen sind. Auch haben wir definiert $Q_{i+1} = \widehat{R}(Q_i)$; aber wir haben am Ende des Beweises gesehen, daß $\widehat{R}$ genau dem Funktional $\mathcal{F}_{\beta,\gamma}$ entspricht.

$$
\begin{aligned}
f_0 &= \Omega \\
f_1 &= \mathcal{F}_{\beta,\gamma}(f_0) \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ f_0(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \mathit{undef}. & \text{sonst} \end{cases} \\
f_2 &= \mathcal{F}_{\beta,\gamma}(f_1) \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ f_1(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \gamma(\sigma) & \text{falls } \beta(\sigma) = \mathit{true} \text{ und } \beta(\gamma(\sigma)) = \mathit{false} \\ \mathit{undef}. & \text{sonst} \end{cases} \\
f_3 &= \mathcal{F}_{\beta,\gamma}(f_2) \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ f_2(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \gamma(\sigma) & \text{falls } \beta(\sigma) = \mathit{true} \text{ und } \beta(\gamma(\sigma)) = \mathit{false} \\ \gamma(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true}, \beta(\gamma(\sigma)) = \mathit{true} \\ & \quad \text{und } \beta(\gamma(\gamma(\sigma))) = \mathit{false} \\ \mathit{undef}. & \text{sonst} \end{cases} \\
f_{i+1} &= \mathcal{F}_{\beta,\gamma}(f_i) \\
&= \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ f_i(\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases} \\
&= \lambda\sigma \in \Sigma. \begin{cases} \gamma^j(\sigma) & \text{falls ein } j \leq i \text{ mit } \beta(\gamma^j(\sigma)) = \mathit{false} \text{ existiert,} \\ & \quad \text{so daß für alle } k < j \text{ gilt } \beta(\gamma^k(\sigma)) = \mathit{true} \\ \mathit{undef}. & \text{sonst} \end{cases} \\
f &= \bigcup_{i\in\mathbb{N}} f_i \\
&= \lambda\sigma \in \Sigma. \begin{cases} \gamma^j(\sigma) & \text{falls ein } j \in \mathbb{N} \text{ mit } \beta(\gamma^j(\sigma)) = \mathit{false} \text{ existiert,} \\ & \quad \text{so daß für alle } k < j \text{ gilt } \beta(\gamma^k(\sigma)) = \mathit{true} \\ \mathit{undef}. & \text{sonst} \end{cases}
\end{aligned}
$$

Man sieht also, daß $f(\sigma)$ das Ergebnis $\gamma^j(\sigma)$ liefert, wenn $\beta(\gamma^j(\sigma))$ wahr ist und für alle $k < j$ gilt $\beta(\gamma^k(\sigma))$ falsch ist. Da jede Anwendung von γ genau einer Ausführung des Schleifenrumpfes entspricht, entspricht $\gamma^k(\sigma)$ dem Zustand nach dem k-ten Schleifendurchlauf und $\beta(\gamma^k(\sigma))$ der Auswertung der Schleifenbedingung nach dem k-ten Schleifendurchlauf. Insgesamt spiegelt also f genau das operationalen Verhalten der Schleife wieder. Die einzelnen f_{i+1} entsprechen dem i-maligen „Abwickeln“ der Schleife. Die Folge der Abbildungen $f_0 \subseteq f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots$ nähert sich immer weiter an f an. Man nennt diese Folge deshalb auch *Fixpunktapproximation*.
Wie die operationale Semantik können wir auch die mathematische Semantik dazu benutzen, die Äquivalenz zweier Anweisungen zu beweisen. Dabei sind zwei Anweisungen c_0 und c_1 äquivalent, wenn sie die gleiche Semantik haben, d. h. wenn gilt $\mathcal{C}[\![c_0]\!] = \mathcal{C}[\![c_1]\!]$.

> *Daß das so ist, folgt aus der Äquivalenz der mathematischen Semantik und der operationalen Semantik, die wir erst später beweisen werden (siehe Satz 5.11 und Folgerung 5.12).*

Beispiel 5.2
Wir betrachten wieder das Programm $w \equiv \textbf{while } b \textbf{ do } c$ und werden zeigen, daß gilt $w \sim \textbf{if } b \textbf{ then } c;\ w \textbf{ else skip}$. Sei nun $\beta = \mathcal{B}[\![b]\!]$ und $\gamma = \mathcal{C}[\![c]\!]$. Dann gilt

$$
\begin{array}{ll}
& \mathcal{C}[\![w]\!] \\
= & \text{Def. von } \mathcal{C}[\![w]\!] \text{ als kleinster Fixpunkt von } \mathcal{F}_{\beta,\gamma} \\
& \mathcal{F}_{\beta,\gamma}(\mathcal{C}[\![w]\!]) \\
= & \text{Def. von } \mathcal{F}_{\beta,\gamma} \\
& \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \mathcal{C}[\![w]\!](\gamma(\sigma)) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases} \\
= & \text{Def. von } \mathcal{C}[\![c;\ w]\!] \text{ und } \gamma = \mathcal{C}[\![c]\!] \\
& \lambda\sigma \in \Sigma. \begin{cases} \sigma & \text{falls } \beta(\sigma) = \mathit{false} \\ \mathcal{C}[\![c;\ w]\!](\sigma) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases} \\
= & \text{Def. } \mathcal{C}[\![\textbf{skip}]\!] \\
& \lambda\sigma \in \Sigma. \begin{cases} \mathcal{C}[\![\textbf{skip}]\!](\sigma) & \text{falls } \beta(\sigma) = \mathit{false} \\ \mathcal{C}[\![c;\ w]\!](\sigma) & \text{falls } \beta(\sigma) = \mathit{true} \end{cases} \\
= & \text{Def. von } \mathcal{C}[\![\textbf{if } b \textbf{ then } c;\ w \textbf{ else skip}]\!] \text{ mit } \beta = \mathcal{B}[\![b]\!] \\
& \mathcal{C}[\![\textbf{if } b \textbf{ then } c;\ w \textbf{ else skip}]\!]
\end{array}
$$

Das Interessante an diesem Beweis ist, daß wir nur ausgenutzt haben, daß $\mathcal{C}[\![w]\!]$ ein Fixpunkt von $\mathcal{F}_{\beta,\gamma}$ ist. Wir haben nicht ausgenutzt, daß es der kleinste Fixpunkt ist.

5 Äquivalenz der Semantiken

Bisher haben wir zwei verschiedene Semantiken für unsere Programmiersprache definiert, die operationale und die mathematische Semantik. Natürlich erwarten wir, daß sie – auch wenn sie auf unterschiedliche Weise definiert sind – übereinstimmen. Dies werden wir in diesem Abschnitt beweisen. Wie bereits in Abschnitt 1 motiviert, sind die mathematische und die operationalen Semantik *äquivalent*, wenn für jede Anweisung c und alle Zustände σ, σ' genau dann $\mathcal{C}[\![c]\!](\sigma) = \sigma'$ gilt, wenn $\langle c, \sigma\rangle \to \sigma'$ gilt. Alternativ dazu könnten wir auch formulieren: $\mathcal{C}[\![c]\!] = \{(\sigma, \sigma') \mid \langle c, \sigma\rangle \to \sigma'\}$
Bevor wir die Äquivalenz der Semantiken für Ausdrücke beweisen können, müssen wir zunächst die Äquivalenz der Semantiken für arithmetische Ausdrücke und boolesche Ausdrücke beweisen:

Lemma 5.8 (Äquivalenz der Semantiken für Ausdrücke)

1. Für jeden arithmetischen Ausdruck a gilt $\mathcal{A}[\![a]\!] = \{(\sigma, n) \mid \langle a, \sigma\rangle \to n\}$.

2. Für jeden booleschen Ausdruck b gilt $\mathcal{B}[\![b]\!] = \{(\sigma, t) \mid \langle b, \sigma\rangle \to t\}$.

Beweis: Induktion über den Aufbau der Ausdrücke.

> *Ein genauerer Beweis für 1. wird in der Übung besprochen. Der Beweis für 2. geht dann analog.*

□

Zum Beweis der Äquivalenz der Semantiken für Anweisungen teilen wir die Äquivalenz in zwei Implikationen auf. Zunächst beweisen wir:

Lemma 5.9
Für jede Anweisung c und alle Zustände σ und σ' mit $\langle c, \sigma\rangle \to \sigma'$ gilt auch $\mathcal{C}[\![c]\!](\sigma) = \sigma'$.

Beweis: Wir beweisen die Aussage durch Induktion über die Regel für die operationale Semantik. Das Prädikat P ist dabei

$$P(\langle c, \sigma\rangle \to \sigma') \equiv \mathcal{C}[\![c]\!](\sigma) = \sigma'$$

Wir betrachten nun jede Regel der operationalen Semantik:

$\frac{}{\langle \mathbf{skip}, \sigma\rangle \to \sigma}$:

Gemäß Definition von $\mathcal{C}$ (d. h. gemäß Def. 5.7) gilt $\mathcal{C}[\![\mathbf{skip}]\!] = id_\Sigma$, d. h. $\mathcal{C}[\![\mathbf{skip}]\!](\sigma) = \sigma$.

$\frac{\langle a, \sigma\rangle \to n}{\langle v{:=}a, \sigma\rangle \to \sigma[n/v]}$:

Gemäß Lemma 5.8.1 gilt $\mathcal{A}[\![a]\!](\sigma) = n$, gemäß Definition von $\mathcal{C}$ gilt $\mathcal{C}[\![v{:=}a]\!] = \lambda\sigma \in \Sigma.\sigma[\mathcal{A}[\![a]\!](\sigma)/v]$. Damit gilt $\mathcal{C}[\![v{:=}a]\!](\sigma) = \sigma[\mathcal{A}[\![a]\!](\sigma)/v] = \sigma[n/v]$.

$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0\,; c_1, \sigma\rangle \to \sigma'}$:

Gemäß Induktionsvoraussetzung gilt $\mathcal{C}[\![c_0]\!](\sigma) = \sigma''$ und $\mathcal{C}[\![c_1]\!](\sigma'') = \sigma'$.
Gemäß Definition von $\mathcal{C}$ gilt $\mathcal{C}[\![c_0; c_1]\!](\sigma) = \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!](\sigma)) = \mathcal{C}[\![c_1]\!](\sigma'') = \sigma'$.

$\frac{\langle b, \sigma\rangle \to \mathit{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to \sigma'}$:

Gemäß Lemma 5.8.2 gilt $\mathcal{B}[\![b]\!](\sigma) = \mathit{true}$ und gemäß Induktionsvoraussetzung gilt $\mathcal{C}[\![c_0]\!](\sigma) = \sigma'$.
Gemäß Definition von $\mathcal{C}$ gilt $\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!](\sigma) = \mathcal{C}[\![c_0]\!](\sigma) = \sigma'$.

$\frac{\langle b, \sigma\rangle \to \mathit{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to \sigma'}$:

Gemäß Lemma 5.8.2 gilt $\mathcal{B}[\![b]\!](\sigma) = \mathit{false}$ und gemäß Induktionsvoraussetzung gilt $\mathcal{C}[\![c_1]\!](\sigma) = \sigma'$.
Gemäß Definition von $\mathcal{C}$ gilt $\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!](\sigma) = \mathcal{C}[\![c_1]\!](\sigma) = \sigma'$.

$\frac{\langle b, \sigma\rangle \to \mathit{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma}$:

Gemäß Lemma 5.8.2 gilt $\mathcal{B}[\![b]\!](\sigma) = \mathit{false}$. Gemäß Definition von $\mathcal{C}$ gilt $\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!](\sigma) = \mathcal{F}_{\mathcal{B}[\![b]\!],\mathcal{C}[\![c]\!]}(\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!])(\sigma) = \sigma$.

$\frac{\langle b, \sigma\rangle \to \mathit{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma''\rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma'}$:

Gemäß Lemma 5.8.2 gilt $\mathcal{B}[\![b]\!](\sigma) = \mathit{true}$ und gemäß Induktionsvoraussetzung gilt $\mathcal{C}[\![c]\!](\sigma) = \sigma''$ und $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\sigma'') = \sigma'$. Gemäß Definition von $\mathcal{C}$ gilt $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\sigma) = \mathcal{F}_{\mathcal{B}[\![b]\!],\mathcal{C}[\![c]\!]}(\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])(\sigma) = \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\mathcal{C}[\![c]\!](\sigma)) = \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\sigma'') = \sigma'$.

□

Insgesamt ist der obige Beweis recht „langweilig“. Nur die beiden letzten Fälle, die über die Schleife argumentieren, sind etwas interessanter. In diesen beiden Fällen nutzen wir aus, daß die Semantik der Schleife als Fixpunkt von $\mathcal{F}_{\mathcal{B}[\![b]\!],\mathcal{C}[\![c]\!]}$ definiert ist: $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\sigma) = \mathcal{F}_{\mathcal{B}[\![b]\!],\mathcal{C}[\![c]\!]}(\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])$. In diesem Beweis nutzen wir nicht einmal aus, daß die Semantik der Schleife als kleinster Fixpunkt von $\mathcal{F}_{\mathcal{B}[\![b]\!],\mathcal{C}[\![c]\!]}$ definiert ist. Das sollte uns etwas stutzig machen. Denn an irgendeiner Stelle sollte natürlich in den Beweis der Äquivalenz auch eingehen, daß wir den kleinsten Fixpunkt als Semantik für die Schleife gewählt haben. Dies wird aber erst bei der Implikation in die andere Richtung eingehen. Diese umgekehrte Richtung beweisen wir als nächstes:

Lemma 5.10
Für jede Anweisung c und alle Zustände σ und σ' mit $\mathcal{C}[\![c]\!](\sigma) = \sigma'$ gilt auch $\langle c, \sigma\rangle \to \sigma'$.

Beweis:

Man könnte geneigt sein, zu glauben, daß der Beweis nun ganz analog zum Beweis von Lemma 5.9 geht, und sich deshalb den Beweis sparen.

Wie bereits oben erwähnt, kann das aber nicht stimmen. Denn im Beweis von Lemma 5.9 haben wir noch an keiner Stelle ausgenutzt, daß die Semantik der Schleife als kleinster Fixpunkt des Funktionals $\mathcal{F}_{\beta,\gamma}$ definiert ist. Dieses Argument muß also in den Beweis dieses Lemmas eingehen. Der Beweis kann also nicht analog zum Beweis von Lemma 5.9 sein.

Darüber hinaus sieht man schnell, daß der Beweis dieses Lemmas nicht durch Regelinduktion bewiesen werden kann.

Wir beweisen die Aussage induktiv über den Aufbau der Anweisungen. Das Prädikat P, das wir beweisen, definieren wir wie folgt:

$$P(c) \equiv \forall \sigma, \sigma' \in \Sigma.(\mathcal{C}[\![c]\!](\sigma) = \sigma' \Rightarrow \langle c, \sigma\rangle \to \sigma')$$

Natürlich ist auch in diesem Beweis der einzig spannende Fall das Schleife. Der Vollständigkeit halber betrachten wir aber alle Konstrukte.

skip:
Gemäß Def. von $\mathcal{C}$ (Def. 5.7) gilt $\mathcal{C}[\![\mathbf{skip}]\!] = id_\Sigma$. Für Zustände σ und σ' mit $\mathcal{C}[\![\mathbf{skip}]\!](\sigma) = \sigma'$ gilt also $\sigma = \sigma'$. Gemäß der Regel

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \to \sigma}$$

für die operationale Semantik läßt sich dafür auch $\langle \mathbf{skip}, \sigma\rangle \to \sigma$ ableiten.

$v{:=}a$:
Gemäß Def. von $\mathcal{C}$ gilt $\sigma' = \mathcal{C}[\![v{:=}a]\!](\sigma) = \sigma[\mathcal{A}[\![a]\!](\sigma)/v]$. Gemäß Lemma 5.8.1 gilt dann auch $\langle a, \sigma\rangle \to \mathcal{A}[\![a]\!](\sigma)$. Mit der Regel

$$\frac{\langle a, \sigma\rangle \to n}{\langle v{:=}a, \sigma\rangle \to \sigma[n/v]}$$

ist dann $\langle v{:=}a, \sigma\rangle \to \sigma[\mathcal{A}[\![a]\!](\sigma)/v]$ herleitbar.

$c_0\,;\, c_1$:
Gemäß Def. von $\mathcal{C}$ gilt $\sigma' = \mathcal{C}[\![c_0\,;c_1]\!](\sigma) = \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!](\sigma))$, d. h. es gibt ein σ'' mit $\sigma' = \mathcal{C}[\![c_1]\!](\sigma'')$ und $\sigma'' = \mathcal{C}[\![c_0]\!](\sigma)$. Gemäß Induktionsvoraussetzung gilt dann $\langle c_0, \sigma\rangle \to \sigma''$ und $\langle c_1, \sigma''\rangle \to \sigma'$. Mit der Regel

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0\,;c_1, \sigma\rangle \to \sigma'}$$

läßt sich dann $\langle c_0\,;c_1, \sigma\rangle \to \sigma'$ herleiten.

$c \equiv \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$:
Gelte nun $\sigma' = \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!](\sigma)$. Wir unterscheiden zwei Fälle:

1. $\mathcal{B}[\![b]\!](\sigma) = \mathit{false}$:
   Gemäß Lemma 5.8.2 gilt dann $\langle b, \sigma\rangle \to \mathit{false}$. Gemäß Def. von $\mathcal{C}$ gilt in diesem Fall $\sigma' = \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!](\sigma) = \mathcal{C}[\![c_1]\!](\sigma)$. Gemäß Induktionsvoraussetzung gilt damit $\langle c_1, \sigma\rangle \to \sigma'$. Mit der Regel
   $$\frac{\langle b, \sigma\rangle \to \mathit{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to \sigma'}$$
   ist dann $\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to \sigma'$ herleitbar.

2. $\mathcal{B}[\![b]\!](\sigma) = \mathit{true}$:
   Gemäß Lemma 5.8.2 gilt dann $\langle b, \sigma\rangle \to \mathit{true}$. Gemäß Def. von $\mathcal{C}$ gilt in diesem Fall $\sigma' = \mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!](\sigma) = \mathcal{C}[\![c_0]\!](\sigma)$. Gemäß Induktionsvoraussetzung gilt damit $\langle c_0, \sigma\rangle \to \sigma'$. Mit der Regel
   $$\frac{\langle b, \sigma\rangle \to \mathit{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}$$
   ist dann $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'$ herleitbar.

$w \equiv \textbf{while } b \textbf{ do } c$:
Sei $\gamma = \mathcal{C}[\![c]\!]$ und $\beta = \mathcal{B}[\![b]\!]$. Dann gilt gemäß Def. $\mathcal{C}[\![w]\!] = \mathit{fix}\, \mathcal{F}_{\beta,\gamma}$. Wir müssen nun zeigen, daß für alle σ und σ' mit $\sigma' = \mathcal{C}[\![w]\!](\sigma)$ auch $\langle w, \sigma\rangle \to \sigma'$ gilt.

Im Beweis von Satz 5.5 haben wir gesehen, daß $\mathit{fix}\, \mathcal{F}_{\beta,\gamma}$ die Vereinigung der folgenden Abbildungen ist:

- $f_0 = \Omega$
- $f_{i+1} = \mathcal{F}_{\beta,\gamma}(f_i)$

D. h. es gilt $\mathcal{C}[\![w]\!] = \mathit{fix}\, \mathcal{F}_{\beta,\gamma} = \bigcup_{i\in\mathbb{N}} f_i$. Mit $\sigma' = \mathcal{C}[\![w]\!](\sigma)$ gibt es also ein $i \in \mathbb{N}$ mit $f_i(\sigma) = \sigma'$. Wir zeigen nun durch vollständige Induktion, daß für jedes $i \in \mathbb{N}$ mit $\sigma' = f_i(\sigma)$ gilt $\langle w, \sigma\rangle \to \sigma'$:

$i = 0$**:** Wegen $f_0 = \Omega$ gibt es kein σ und σ' mit $\sigma' = f_0(\sigma)$. Deshalb ist nichts zu zeigen.

$i \to i + 1$**:** Sei nun $\sigma' = f_{i+1}(\sigma)$. Wir unterscheiden zwei Fälle:

$\beta(\sigma) = \textbf{\textit{false}}$**:** Gemäß Def. von f_{i+1} und gemäß Def. von $\mathcal{F}_{\beta,\gamma}$ gilt dann $\sigma' = \sigma$. Mit $\beta = \mathcal{B}[\![b]\!]$ und Lemma 5.8.2 gilt dann $\langle b, \sigma\rangle \to \mathit{false}$. Gemäß der Regel
$$\frac{\langle b, \sigma\rangle \to \mathit{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$
ist dann $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma$ herleitbar.

$\beta(\sigma) = \textbf{\textit{true}}$**:** Gemäß Def. von f_{i+1} und gemäß Def. von $\mathcal{F}_{\beta,\gamma}$ gilt dann $\sigma' = f_i(\gamma(\sigma))$. Es gibt also ein σ'' mit $\sigma'' = \gamma(\sigma) = \mathcal{C}[\![c]\!](\sigma)$ und $\sigma' = f_i(\sigma'')$.

Dann gilt gemäß Induktionsvoraussetzung (Induktion über den Aufbau von c): $\langle c, \sigma\rangle \to \sigma''$. Gemäß Induktionsvoraussetzung (vollständige Induktion) gilt dann $\langle w, \sigma''\rangle \to \sigma'$ und mit $\beta = \mathcal{B}[\![b]\!]$ und Lemma 5.8.2 gilt außerdem $\langle b, \sigma\rangle \to \mathit{true}$. Gemäß der Regel

$$\frac{\langle b, \sigma\rangle \to \mathit{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

ist damit auch $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$ herleitbar.

□

Mit Lemma 5.9 und 5.10 haben wir die Äquivalenz der mathematischen und operationalen Semantik für die Programmiersprache IMP bewiesen. Allerdings geht nur in Lemma 5.10 ein, daß die mathematische Semantik einer Schleife als kleinster Fixpunkt des Funktionals $\mathcal{F}_{\beta,\gamma}$ definiert ist. Wir haben hier – schon wieder – ausgenutzt, daß sich der kleinste Fixpunkt durch die Abbildungen $f_0 \subseteq f_1 \subseteq f_2 \subseteq \ldots$ approximieren läßt. Das ist eine wesentlich Eigenschaft des Funktionals $\mathcal{F}_{\beta,\gamma}$, die wir uns im nächsten Kapitel noch etwas genauer ansehen werden.
Da wir im Beweis jetzt insgesamt alle Voraussetzungen aus der Definition der mathematischen Semantik benutzt haben, können uns aber beruhigt zurücklehnen und den folgenden Satz genießen:

Satz 5.11 (Mathematische und operationale Semantik)
Für jede Anweisung $c \in \mathit{Com}$ gilt $\mathcal{C}[\![c]\!] = \{(\sigma, \sigma') \in \Sigma \times \Sigma \mid \langle c, \sigma\rangle \to \sigma'\}$.

Wir hätten auch schreiben können: $\mathcal{C}[\![c]\!](\sigma) = \sigma'$ gdw. $\langle c, \sigma\rangle \to \sigma'$.

Beweis: Folgt unmittelbar aus Lemma 5.9 und 5.10. □

Eine unmittelbare Folgerung dieses Satzes ist, daß auch für die operationale Semantik für jede Anweisung c und jeden Zustand σ höchstens ein Zustand σ' mit $\langle c, \sigma\rangle \to \sigma'$ existiert. Denn sonst wäre $\mathcal{C}[\![c]\!] = \{(\sigma, \sigma') \in \Sigma \times \Sigma \mid \langle c, \sigma\rangle \to \sigma'\}$ keine partielle Abbildung.
Aus dem Satz folgt noch eine weitere schöne Eigenschaft der mathematischen Semantik, die wir bereits in Beispiel 5.2 benutzt haben: Zwei Anweisungen sind genau dann äquivalent, wenn sie dieselbe (mathematische) Semantik haben.

Folgerung 5.12 (Äquivalenz ist Identität der Semantik)
Zwei Anweisungen c und c' sind genau dann äquivalent (in Zeichen $c \sim c'$), wenn sie dieselbe Semantik besitzen (in Zeichen $\mathcal{C}[\![c]\!] = \mathcal{C}[\![c']\!]$).

Beweis: Die Aussage folgt unmittelbar aus der Definition der Äquivalenz von Anweisungen mit Hilfe der operationalen Semantik (Def. 3.8) und der Äquivalenz der mathematischen Semantik und der operationalen Semantik (Satz 5.11). □

6 Zusammenfassung

In diesem Kapitel haben wir die mathematische Semantik für Anweisungen definiert. Die wesentliche Anforderung an die Definition der mathematischen Semantik war, daß sie jedem syntaktischen Objekt c direkt ein semantisches Objekt $\mathcal{C}[\![c]\!]$ zuordnet. Darüber hinaus muß diese Definition kompositional sein, d. h. die Definition der Semantik eines syntaktischen Objektes darf nur die Semantik seiner Teilkonstrukte benutzen. Wir haben gesehen, daß das bei Schleifen nicht ganz einfach ist. Und wir sind schnell darauf gekommen, daß wir den kleinsten Fixpunkt des Schleifenfunktionals $\mathcal{F}_{\beta,\gamma}$ benutzen müssen, um aus diesem Dilemma heraus zu kommen.
Die Beweise der Äquivalenz der mathematischen und operationalen Semantik haben zwei Dinge gezeigt:

1. Wenn man sich die Definitionen der beidene Semantiken bis zum bitteren Ende ansieht (d. h. wenn man sich auch überlegt, was die „Semantik“ einer induktiven Definition ist), unterscheiden sich die Definitionen der mathematischen und der operationalen Semantik gar nicht so wesentlich. Bei der mathematischen Semantik wird die Fixpunkttheorie explizit benutzt – bei der operationalen Semantik nur implizit. Die explizite Benutzung der Fixpunkttheorie liefert uns eine kompositionale Definition der Semantik.

2. Der kleinste Fixpunkt des Funktionals läßt sich durch iterierte Anwendung des Funktionals, also durch die Abbildungen $f_0 \subseteq f_1 \subseteq f_2 \subseteq \ldots$, approximieren. Darauf kommen wir im nächsten Kapitel noch ausführlich zu sprechen.

Kapitel 6

Fixpunkte und semantische Bereiche

Sowohl bei der Definition der operationalen Semantik als auch bei der Definition der mathematischen Semantik haben wir mehr oder weniger explizit Fixpunkte benutzt. Die Existenz der Fixpunkte haben wir jeweils ad hoc bewiesen. Es stellt sich jedoch heraus, daß die Beweise immer wieder nach demselben Schema ablaufen. In diesem Kapitel werden wir deshalb etwas allgemeiner untersuchen, unter welchen Voraussetzungen Fixpunkte bestimmter Funktionen existieren. Dies werden wir in *Fixpunktsätzen* formulieren.

Tatsächlich haben wir bereits zwei Fixpunktsätze kennen gelernt. Beim Beweis, daß für jede Regelmenge die kleinste R-abgeschlossene Menge existiert (Lemma 4.5 in Kapitel 4) haben wir bereits den Fixpunktsatz von Knaster und Tarski kennengelernt. Die äquivalente Charakterisierung der induktiv definierten Menge (Satz 4.8 in Kapitel 4) und auch der Satz über die Existenz des kleinsten Fixpunktes des Funktionals $\mathcal{F}_{\beta,\gamma}$ (Satz 5.5) entspricht dem Fixpunktsatz von Kleene. Diese beiden Sätze werden wir in diesem Kapitel allgemein formulieren, wobei der Satz von Kleene in der Semantik die größere Bedeutung hat.

Mit Hilfe des Satzes von Kleene wissen wir dann, daß für bestimmte Strukturen der kleinste Fixpunkt immer existiert. Allerdings müssen wir dafür beweisen, daß die betrachtete Struktur die Eigenschaft hat, die im Satz von Kleene gefordert sind. Weil dieser Nachweis oft recht mühsam ist, geben wir am Ende systematische Konstruktionsregeln an, die immer zu Strukturen führen, die die nötigen Eigenschaften besitzen.

1 Grundlagen

Bevor wir die Fixpunktsätze angeben können, müssen wir noch einige weitere grundlegende Begriffe einführen.

> *Diese Begriffe wurden bereits in Kapitel 2 definiert. Sie werden hier zur Auffrischung nochmal kurz wiederholt (und in der Vorlesung nur kurz angedeutet).*

Definition 6.1 (Obere Schranke und Supremem)
Sei $(X, \preceq)$ eine reflexive Ordnung und Y eine Teilmenge von X.

1. Ein Element $x \in X$ heißt *obere Schranke* von Y, wenn für alle $y \in Y$ gilt $y \preceq x$.

2. Die kleinste obere Schranke von Y nennen wir, das *Supremum* von Y; wir schreiben dafür auch $\bigvee Y$.

> *Das Symbol, das für das Supremum einer Menge Y benutzt wird, wird oft an das Symbol der zugrundeliegenden Ordnung angepaßt. Beispielsweise schreibt man $\bigcup Y$ für das Supremum bzgl. der Ordnung $\subseteq$ oder oder $\bigsqcup Y$ für das Supremum bzgl. der Ordnung $\sqsubseteq$.*

Eine obere Schranke von Y ist also ein Element, das größer ist als jedes Element aus Y. Es gibt Teilmengen, die keine obere Schranke besitzen. Für $X = Y = \mathbb{N}$ mit der üblichen Ordnungsrelation besitzt Y beispielsweise keine obere Schranke. Wenn wir allerdings X um das Element ω erweitern, besitzt $Y = \mathbb{N}$ eine obere Schranke: ω. Es ist spielt also eine große Rolle, innerhalb welcher Menge bzw. Ordnung X wir die obere Schranken von Y betrachten.
Auch das Supremum einer Menge muß nicht immer existieren, und, wie bei den oberen Schranken, kann die Existenz des Supremums einer Menge Y von der Menge X, innerhalb der wir das Supremum suchen, abhängen. Beispielsweise existiert das Supremum von $\mathbb{N}$ in $\mathbb{N}$ selbst nicht. Aber in $\mathbb{N} \cup \{\omega\}$ besitzt $\mathbb{N}$ (und jede andere Teilmenge von $\mathbb{N}$) ein Supremum.
Es besteht auch ein enger Zusammenhang zwischen dem größten Element einer Menge Y und dem Supremum. Wenn nämlich Y ein größtes Element besitzt, ist dieses größte Element auch das Supremum von Y. Allerdings kann eine Menge Y auch ein Supremum besitzen, wenn Y kein größtes Element besitzt. Beispielsweise existiert für $\mathbb{N}$ in $\mathbb{N} \cup \{\omega\}$ das Supremum, aber $\mathbb{N}$ besitzt kein größtes Element.
Entsprechend der oberen Schranke kann man auch die *unter Schranke* und das *Infimum* als größte untere Schranke definieren.

Definition 6.2 (Untere Schranke und Infimum)
Sei $(X, \preceq)$ eine reflexive Ordnung und Y eine Teilmenge von X.

1. Ein Element $x \in X$ heißt *untere Schranke* von Y, wenn für alle $y \in Y$ gilt $x \preceq y$.

2. Die größte untere Schranke von Y nennen wir, das *Infimum* von Y; wir schreiben dafür auch $\bigwedge Y$.

   Das Symbol, das für das Infimum einer Menge Y benutzt wird, wird oft an das Symbol der zugrundeliegenden Ordnung angepaßt. Beispiels weise schreibt man $\bigcap Y$ für das Infimum bzgl. der Ordnung $\subseteq$ oder oder $\bigsqcap Y$ für das Infimum bzgl. der Ordnung $\sqsubseteq$.
   Oft schreiben wir auch etwas suggestiver $\bigvee_{y \in Y} y$ anstelle von $\bigvee Y$.

Beispiel 6.1

1. Sei Q eine beliebige Menge. Dann ist $(2^Q, \subseteq)$ eine Ordnung. Für jede Teilmenge $Y \subseteq 2^Q$ existiert das Supremum $\bigcup Y = \bigvee_{y \in Y} y$ und das Infimum $\bigcap Y = \bigwedge_{y \in Y} y$.

   Die Potenzmengen sind eine sehr schöne Struktur, da für alle Teilmengen die Infima und Suprema existieren. Leider gilt das für viele andere Strukturen nicht.

2. Für $(\mathbb{N}, \leq)$ existiert für jede endliche Teilmenge $Y \subseteq \mathbb{N}$ das Supremum (das größte Element der endlichen Teilmenge Y); für unendliche Teilmengen $Y \subseteq \mathbb{N}$ existiert das Supremum dagegen nicht. Infima existieren aber für jede nicht-leere Teilmenge von $\mathbb{N}$.

   Fragen: Was ist das Supremum der leeren Menge? Warum existiert das Infimum der leeren Menge nicht?

3. Für $(\mathbb{N} \cup \{\omega\}, \leq)$ existieren – wie bereits erwähnt – für jede Teilmenge das Infimum und das Supremum.

   Frage: Was ist das Infimum der leeren Menge?

4. Für die Menge $X = \{a, b, c, d\}$ mit $a < c$, $b < c$, $a < d$ und $b < d$ existieren die Suprema und die Infima von $\{a, b\}$ und $\{c, d\}$ nicht.

   Frage: Für welche Teilmengen von X existieren die Infima und die Suprema?

Strukturen, für die „alle Suprema und alle Infima“ existieren, sind natürlich besonders schön. Solche Strukturen heißen *vollständige Verbände*.

Definition 6.3 (Vollständiger Verband)
Eine reflexive Ordnung $(X, \preceq)$ heißt *vollständiger Verband*, wenn für jede Teilmenge $Y \subseteq X$ das Infimum existiert.

In der obigen Definition wird für vollständige Verbände nur gefordert, daß „alle Infima“ existieren. In unserer informellen Definition hatten wir jedoch auch verlangt, daß alle „Suprema“ existieren. Der Grund dafür ist, daß aus der Existenz aller Infima auch die Existenz aller Suprema folgt.

Lemma 6.4 (Vollstängige Verbände und Existenz der Suprema)
Sei $(X, \preceq)$ ein vollständiger Verband. Dann existiert für jede Teilmenge $Y \subseteq X$ das Supremum.

Beweis: Sehr einfach (Übung). □

Zuletzt definieren wir die bezüglich einer Ordnung monoton steigenden Abbildungen.

Definition 6.5 (Monoton steigende Abbildungen)
Seien $(X_1, \preceq_2)$ und $(X_2, \preceq_2)$ reflexive Ordnungen. Eine totale Abbildung $f : X_1 \to X_2$ heißt *monoton steigend*, wenn für alle $x, y \in X_1$ mit $x \preceq_1 y$ auch $f(x) \preceq_2 f(y)$ gilt.

Da bei uns keine monoton fallenden Abbildungen vorkommen, reden wir im folgenden nur von monotonen Abbildungen, wenn wir monoton steigende Abbildungen meinen.

> *Wenn sowohl monoton steigende als auch monoton fallende Abbildungen betrachtet werden, nennt man die monoton steigenden manchmal auch* isoton *und die monoton fallenden* antiton.

2 Satz von Knaster und Tarski

Mit Hilfe dieser Begriffe können wir nun den Fixpunktsatz von Knaster und Tarski formulieren:

Satz 6.6 (Knaster-Tarski)
Sei $(X, \preceq)$ ein vollständiger Verband und $f : X \to X$ eine monotone Abbildung. Dann ist $\bigwedge\{x \in X \mid f(x) \preceq x\}$ der kleinste Fixpunkt von f.

Beweis: Sei $Y = \{x \in X \mid f(x) \preceq x\}$ und $y = \bigwedge Y$.
Wir zeigen zunächst, daß gilt $f(y) \preceq y$ (d. h. daß y ein Präfixpunkt von f ist): Dazu zeigen wir zunächst, daß $f(y)$ eine untere Schranke von Y ist. Sei also $x \in Y$ beliebig. Da y Infimum von Y ist, gilt $y \preceq x$. Wegen der Monotonie von f gilt also auch $f(y) \preceq f(x)$. Mit $x \in Y$ und der Definition von Y gilt auch $f(x) \preceq x$; insgesamt gilt also $f(y) \preceq x$. Damit ist $f(y)$ also eine unter Schranke von Y und, da y die größte untere Schranke von Y ist gilt $f(y) \preceq y$.
Als nächstes zeigen wir, daß y ein Fixpunkt von f ist: Aus $f(y) \preceq y$ folgt wegen der Monotonie von f auch $f(f(y)) \preceq f(y)$. Damit gilt $f(y) \in Y$. Da y das Infimum von Y (also insbesondere eine untere Schranke von y) ist, gilt $y \preceq f(y)$. Zusammen mit $f(y) \preceq y$ gilt $f(y) = y$.
Es bleibt zu zeigen, daß y kleiner als jeder andere Fixpunkt ist. Sei also x ein beliebige Fixpunkt von f, d. h. ein $x \in X$ mit $f(x) = x$. Dann gilt insbesondere $f(x) \preceq x$. Damit gilt $x \in Y$. Da y das Infimum von Y ist gilt $y \prec x$.
Insgesamt haben wir damit gezeigt, daß $y = \bigwedge\{x \in X \mid f(x) \preceq x\}$ der kleinste Fixpunkt von f ist. □

Diesen Satz hätten wir bereits zum Beweis von Lemma 4.5 benutzen können, wo wir gezeigt haben, daß es eine kleinste R-abgeschlossen Menge gibt. Allerdings mußten wir dort das Lemma noch direkt beweisen (vgl. Übung), weil wir den Satz noch nicht kannten. Dabei ist dort $(2^X, \subseteq)$ der zugrundeliegende vollständige Verband und $\widehat{R}$ ist die monotone Abbildung, für die wir die Existenz des kleinsten Fixpunktes bewiesen haben.

> *Streng genommen haben wir in Lemma 4.5 nicht die Existenz des kleinsten Fixpunktes von $\widehat{R}$ bewiesen, sondern die Existenz des kleinsten Präfixpunktes. Dabei sind die Präfixpunkte von $\widehat{R}$ gerade die R-abgeschlossenen Mengen ($\widehat{R}(Q) \subseteq Q$). Wir haben also die Existenz der kleinsten R-abgeschlossenen Menge bewiesen. Wie man im obigen Beweis sieht, fällt der kleinste Präfixpunkt aber mit dem kleinsten Fixpunkt zusammen.*

3 Semantische Bereiche und Satz von Kleene

Der Fixpunktsatz von Knaster und Tarski ist ein schöner Satz und hat viele Anwendungen. Leider sind die Strukturen, die wir bei der Definition von Semantiken benutzen, meist keine vollständigen Verbände. Wir können diesen Satz in der Semantik also oft nicht anwenden. Beispielsweise ist das Funktional $\mathcal{F}_{\beta,\gamma}$ bei der Definition der mathematischen Semantik auf der Menge

der partiellen Abbildungen definiert. Diese bilden leider keinen vollständigen Verband.
Deshalb benötigen wir in der Semantik einen anderen Fixpunktsatz, der auch für allgemeinere Strukturen gilt – wie zum Beispiel für die Menge der partiellen Abbildungen. Außerdem werden wir sehen, daß wir für diese Strukturen den kleinsten Fixpunkt der Abbildung approximieren können.

> *Die Pioniere der Semantik haben lange versucht, die zugrundeliegenden Strukturen zu vollständigen Verbänden zu machen, um eine elegante Definition einer mathematische Semantik auf der Basis des Satzes von Knaster und Tarski zu formulieren. Es hat sich aber herausgestellt, daß dies nicht so gut funktioniert und es zweckmäßiger ist, nach anderen Strukturen mit anderen Fixpunktsätzen zu suchen.*

Die Struktur, auf der der Fixpunktsatz operiert, nennen wir *semantischen Bereich*[1].

Definition 6.7 (Semantischer Bereich)
Eine reflexive Ordnung $(D, \sqsubseteq)$ heißt *semantischer Bereich*, wenn für jede (unendliche) aufsteigende Kette $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ mit $d_i \in D$ das Supremum $\bigsqcup_{i \in \mathbb{N}} d_i$ existiert.
Der semantische Bereich heißt *semantischer Bereich mit kleinstem Element*, wenn D ein kleinstes Element besitzt. Das kleinste Element wird dann mit $\bot_D$ bezeichnet.

> *Wenn D aus dem Kontext hervor geht, schreiben wir auch kurz $\bot$ anstelle von $\bot_D$.*
>
> *Oft werden in der Literatur nur semantische Bereiche mit kleinstem Element betrachtet. Dann wird aber meist auf den Zusatz „mit kleinstem Element“ verzichtet.*

Offensichtlich ist jeder vollständige Verband auch ein semantischer Bereich, denn es existieren alle Suprema, also auch die für die aufsteigenden Ketten. Allerdings gibt es auch andere Strukturen, die ein semantischer Bereich sind – das war ja schließlich das Ziel der Übung.

Beispiel 6.2 (Semantische Bereiche)

1. Für jede Menge X ist die Menge der partiellen Abbildungen $D = (X \rightharpoonup X)$ mit $\subseteq$ (auf den entsprechenden Relationen der Funktionen) als Ordnung ein semantischer Bereich.

[1] Im Englischen nennt man semantische Bereiche *domain* oder auch *complete partial order (cpo)*. Deshalb benutzen wir D als Symbol für die zugrundeliegende Menge.

Zur Erinnerung $f \subseteq f'$ bedeutet, daß an den Stellen x, an denen $f(x)$ definiert ist, auch $f'(x) = f(x)$ gilt. f' kann aber an mehr Stellen definiert sein. $f \subseteq f'$ bedeutet also, daß f' „definierter ist als“ f.

Für eine aufsteigende Kette $f_0 \subseteq f_1 \subseteq f_2 \subseteq \ldots$ ist $f = \bigcup_{i\in\mathbb{N}} f_i$ das Supremum.

Dieser semantische Bereich besitzt auch ein kleinstes Element, nämlich die überall undefinierte Funktion: $\bot_{(X\rightharpoonup X)} = \Omega = \emptyset$

2. Für jede Menge A ist die Menge der endlichen und unendlichen Sequenzen $A^\infty = A^* \cup A^\omega$ mit der Präfixrelation $\sqsubseteq$ ein semantischer Bereich mit der leeren Sequenz als kleinstem Element $\bot_{A^\infty} = \varepsilon$.

   Für die folgende Kette $\varepsilon \sqsubseteq a \sqsubseteq ab \sqsubseteq aba \sqsubseteq abab \sqsubseteq \ldots$ ist die unendliche Sequenz $ababab\ldots$ das Supremum.

3. Für jede Menge X ist (X, id_X) ein semantischer Bereich. Dieser semantische Bereich heißt auch der *diskrete semantische Bereich* über X. Dieser semantische Bereich besitzt im allgemeinen jedoch kein kleinstes Element.

Für semantische Bereiche hat nicht mehr jede monotone Abbildung einen kleinsten Fixpunkt. Da wir die Anforderungen an die zugrundeliegende Struktur abgeschwächt haben, müssen wir die Anforderungen an die Abbildungen verschärfen, um zu gewährleisten, daß sie einen kleinsten Fixpunkt besitzen.

Definition 6.8 (Stetige Abbildung)
Seien $(D_1, \sqsubseteq_1)$ und $(D_2, \sqsubseteq_2)$ semantische Bereiche (die nicht notwendig ein kleinstes Element besitzen müssen). Eine Abbildung $f : D_1 \to D_2$ heißt *stetig*, wenn für jede nicht-leere aufsteigende Kette $d_0 \sqsubseteq_1 d_1 \sqsubseteq_1 d_2 \sqsubseteq_1 \ldots$ in D_1 das Supremum von $\{f(d_i) \mid i \in \mathbb{N}\}$ existiert und gilt $\bigsqcup_{i\in\mathbb{N}} f(d_i) = f(\bigsqcup_{i\in\mathbb{N}} d_i)$.

Die Supremumsbildung kann man als Limesbildung auffassen (vgl. das Beispiel mit der unendlichen Sequenz $abababa\ldots$ als „Limes“ der Menge $\{\varepsilon, a, ab, aba, abab, \ldots\}$). Die Stetigkeit einer Abbildung bedeutet dann, daß man Limesbildung und Funktionsanwendung vertauschen kann. In diesem Sinne entspricht die Definition der Stetigkeit dem Begriff der Stetigkeit von Abbildungen auf den reellen Zahlen, wie sie aus der Analysis bekannt ist.

Aus der Definition der stetigen Abbildungen folgt unmittelbar:

Lemma 6.9 (Stetige Abbildungen sind monoton)
Seien $(D_1, \sqsubseteq_1)$ und $(D_2, \sqsubseteq_2)$ semantische Bereiche. Jede stetige Abbildung $f : D_1 \to D_2$ ist monoton.

Umgekehrt existiert für jede monotone Abbildung $f : D_1 \to D_2$ und jede aufsteigende Kette $d_0 \sqsubseteq_1 d_1 \sqsubseteq_1 d_2 \sqsubseteq_1 \ldots$ auch das Supremum $\bigsqcup_{i\in\mathbb{N}} f(d_i)$. Denn wegen der Monotonie von f gilt $f(d_0) \sqsubseteq_2 f(d_1) \sqsubseteq_2 f(d_2) \sqsubseteq_2 \ldots$. Da $(D_2, \sqsubseteq_2)$ ein semantischer Bereich ist, existiert also auch das Supremum dieser Kette; allerdings gilt nicht unbedingt $\bigsqcup_{i\in\mathbb{N}} f(d_i) = f(\bigsqcup_{i\in\mathbb{N}} d_i)$.
Bevor wir uns Beispiele für stetige Abbildungen ansehen, betrachten wir ein Beispiel für nicht-stetige Abbildungen. Die sind viel interessanter, da die meisten monotonen Abbildungen, die wir uns ausdenken können, auch stetig sind.

Beispiel 6.3 (Monotone aber nicht stetige Abbildung)
Sei $A = \{a, b\}$ und $(A^\infty, \sqsubseteq)$ der semantische Bereich der endlichen und unendlichen Sequenzen über A mit der Präfixrelation $\sqsubseteq$ als Ordnung. Wir definieren nun die Abbildung $f : A^\infty \to A^\infty$ durch

- $f(\sigma) = \varepsilon$ für $\sigma \in A^*$, d. h. für endliche Sequenzen über A und
- $f(\sigma) = a$ für $\sigma \in A^\omega$, d. h. für unendliche Sequenzen über A.

Offensichtlich ist f monoton. Wir betrachten nun die aufsteigende Kette von Sequenzen aus A^∞ mit $\sigma_i = a^i$. Das Supremum dieser Kette ist die unendliche Sequenz $\sigma = aaaaa\ldots$. Also gilt $f(\bigsqcup_{i\in\mathbb{N}} \sigma_i) = f(aaaa\ldots) = a$. Allerdings gilt für jedes $i \in \mathbb{N}$: $f(\sigma_i) = \varepsilon$. Und damit gilt $\bigsqcup_{i\in\mathbb{N}} f(\sigma_i) = \bigsqcup_{i\in\mathbb{N}} \varepsilon = \varepsilon$.
Der Grund für dieses Verhalten ist, daß die Abbildung f für unendliche Sequenzen einen Sprung macht (von ε für alle endlichen und auch extrem langen Sequenzen auf a für unendliche Sequenzen).

Beispiele für stetige Abbildungen sind leichter zu finden: Beispielsweise ist die Abbildung $\widehat{R}$ für jede Menge von Regeln R (mit jeweils endlich vielen Voraussetzungen) über Q stetig. Auch die Abbildung $length : A^\infty \to \mathbb{N} \cup \{\omega\}$, die jeder Sequenz ihre (endliche oder unendliche) Länge zuordnet, ist stetig.
Mit diesen Begriffen können wir nun den Fixpunktsatz von Kleene formulieren:

Satz 6.10 (Fixpunktsatz von Kleene)
Sei $(D, \sqsubseteq)$ ein semantischer Bereich mit kleinsten Element $\bot$ und sei $f : D \to D$ eine stetige Abbildung. Dann existiert das Supremum $\bigsqcup_{i\in\mathbb{N}} f^i(\bot)$ und es ist der kleinste Fixpunkt von f.

d. h. $\text{fix}(f) = \bigsqcup_{i \in \mathbb{N}} f^i(\bot)$

Beweis:

Der Beweis verläuft ganz analog zum Beweis von Satz 4.8.

Zunächst beweisen wir durch vollständige Induktion, daß für jedes $i \in \mathbb{N}$ gilt $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$:

$i = 0$**:** $f^0(\bot) = \bot \sqsubseteq f(\bot) = f^1(\bot)$.

$i \to i+1$: Gemäß Induktionsvoraussetzung gilt $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$. Da f stetig und damit gemäß Lemma 6.9 auch monoton ist, gilt $f(f^i(\bot)) \sqsubseteq f(f^{i+1}(\bot))$. Also gilt $f^{i+1}(\bot) \sqsubseteq f^{i+2}(\bot)$.

Die $f^i(\bot)$ bilden also eine aufsteigende Kette in $(D, \sqsubseteq)$. Da $(D, \sqsubseteq)$ ein semantischer Bereich ist, existiert das Supremum $d = \bigsqcup_{i \in \mathbb{N}} f^i(\bot)$.
Wir zeigen nun, daß d ein Fixpunkt von f ist:

$f(d) =$	Def. d
$f(\bigsqcup_{i \in \mathbb{N}} f^i(\bot)) =$	f stetig
$\bigsqcup_{i \in \mathbb{N}} f(f^i(\bot)) =$	Def f^{i+1}
$\bigsqcup_{i \in \mathbb{N}} f^{i+1}(\bot) =$	$\bot$ ist kleinstes Element
$\bigsqcup_{i \in \mathbb{N}} f^{i+1}(\bot) \sqcup \bot =$	Def. $f^0(\bot)$
$\bigsqcup_{i \in \mathbb{N}} f^{i+1}(\bot) \sqcup f^0(\bot) =$	Umsortierung
$\bigsqcup_{i \in \mathbb{N}} f^i(\bot) = d$	

Zuletzt zeigen wir, daß d der kleinste Fixpunkt von f ist. Sei d' ein beliebiger Fixpunkt von f, d. h. $d' \in D$ mit $f(d') = d'$. Wir zeigen durch vollständige Induktion, daß für jede $i \in \mathbb{N}$ gilt $f^i(\bot) \sqsubseteq d'$:

$i = 0$**:** $f^0(\bot) = \bot \sqsubseteq d'$, da $\bot$ das kleinste Element von D ist.

$i \to i+1$**:** Gemäß Induktionsannahme gilt $f^i(\bot) \sqsubseteq d'$. Da f stetig und damit monoton ist, gilt auch $f^{i+1}(\bot) \sqsubseteq f(d') = d'$.

Damit ist d' eine obere Schranke für alle $f^i(\bot)$. Da d die kleinste obere Schranke aller $f^i(\bot)$ ist (Supremum), gilt $d \sqsubseteq d'$. □

Zum besseren Verständnis betrachten wir einige Beispiele.

Beispiel 6.4 (Fixpunkte von stetigen Abbildungen)

1. Sei R eine Regelmenge über X, wobei jede Regel nur endlich viele Voraussetzungen hat. Dann ist $\widehat{R}$ auf der Menge der Teilmengen von X bezüglich $\subseteq$ eine stetige Abbildung (das werden wir in der Übung beweisen).

   Dann ist $I_R = \bigcup_{i\in\mathbb{N}} Q_i$ mit $Q_0 = \emptyset$ und $Q_{i+1} = \widehat{R}(Q_i)$ der kleinste Fixpunkt von $\widehat{R}$ (vgl. Satz 4.8).

2. Sei A eine endliche Menge mit $a \in A$ und A^∞ die Menge der endlichen und unendlichen Sequenzen über A und $length : A^\infty \to \mathbb{N} \cup \{\omega\}$ die Abbildung, die jeder Sequenz $\sigma \in A^\infty$ ihre Länge zuordnet.

   Dann ist die Abbildung $f : A^\infty \to A^\infty$ mit $f(\sigma) = a^{length(\sigma)}$ stetig (übrigens ist auch die Abbildung $length$ stetig).

   Die Abbildung besitzt also in A^∞ einen kleinsten Fixpunkt.

   Frage: Was ist der kleinste Fixpunkt von f?
   Frage: Was ist der kleinste Fixpunkt von f mit $f(\sigma) = a^{length(\sigma)+1}$?

Der Fixpunktsatz von Kleene hat gegenüber dem Satz von Knaster und Tarski zwei wesentliche Vorteile:

1. Er funktioniert auch dann, wenn die zugrundeliegende Struktur kein vollständiger Verband ist. Es reicht, wenn sie ein semantischer Bereich ist. Erfreulicherweise sind die meisten Strukturen, die in der Semantik vorkommen semantische Bereiche, bzw. lassen sich einfach in solche überführen.

2. Ein Problem des Satzes von Knaster und Tarski ist, daß der kleinste Fixpunkt als Durchschnitt aller Präfixpunkte charakterisiert ist. Wir müssen im allgemeinen einen Durchschnitt über unendlich viele Mengen bilden (die wir nicht einmal systematisch konstruieren können). Der Satz liefert also kein konstruktives Verfahren.

   Im Gegensatz dazu erlaubt uns der Satz von Kleene die systematische Konstruktion des Fixpunktes $d = \bigsqcup_{i\in\mathbb{N}} f^i(\bot)$. Natürlich ist auch dies eine unendliche Konstruktion. Allerdings kommen wir mit jedem $f^i(\bot)$ etwas näher an den Fixpunkt heran. Der Fixpunkt wird also durch diese Folge approximiert. Wir sprechen deshalb auch von *Fixpunktapproximation*. Bei der Semantik der Schleife haben wir gesehen, daß es für

einen konkreten Zustand reicht, ein $f^i(\bot)$ zu betrachten; daß also eine Approximation reicht.

In Kapitel 4 haben wir verschiedene Techniken zum induktiven Beweisen kennen gelernt. Diese Techniken werden wir hier noch um eine weitere Technik ergänzen. Mit dieser Technik kann man Eigenschaften des kleinsten Fixpunktes beweisen. Da wir den Fixpunkt einer stetigen Abbildung betrachten, muß dazu die zu beweisende Eigenschaft (bzw. das Prädikat) auch stetig sein.

Definition 6.11 (Stetige Eigenschaft)
Sei $(D, \sqsubseteq)$ ein semantischer Bereich mit kleinstem Element. Ein Prädikat $P \subseteq D$ heißt *stetig*, wenn für jede aufsteigende Kette $p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots$ mit $p_i \in P$ für alle $i \in \mathbb{N}$ auch für das Supremum gilt $\bigsqcup_{i\in\mathbb{N}} p_i \in P$.

Achtung: Es gibt andere Definitionen von stetigen Prädikaten.

Prinzip 6.12 (Berechnungsinduktion)
Sei $(D, \sqsubseteq)$ ein semantischer Bereich mit kleinstem Element $\bot$, sei $f : D \to D$ eine stetige Abbildung und P ein stetiges Prädikat mit

- $P(\bot)$ und
- für alle $d \in D$ mit $P(d)$ gilt auch $P(f(d))$.

Dann gilt auch $P(\mathit{fix}(f))$, d. h. das Prädikat gilt auch für den kleinsten Fixpunkt von f.

Beweis: Diese Beweisprinzip läßt sich mit Hilfe der Charakterisierung des kleinsten Fixpunktes durch den Satz von Kleene und unter Ausnutzung der Stetigkeit von P auf die vollständige Induktion zurückführen (Übungsaufgabe). □

4 Konstruktion Semantischer Bereiche

Der Satz von Kleene liefert uns eine sehr elegante Theorie über die Existenz der Fixpunkte von stetigen Abbildungen. Um den Satz von Kleene anwenden zu können, müssen wir allerdings erst einmal beweisen, daß die Voraussetzungen das Satzes von Kleene vorliegen, d. h. daß die zugrundeliegende Ordnung ein semantischer Bereich ist und daß die Abbildung, deren Fixpunkt wir

betrachten, stetig ist. Der Nachweis dieser Eigenschaften ist teilweise recht mühselig. Deshalb geben wir in diesem Abschnitt einige einfache semantische Bereiche und stetige Abbildungen an; darüber hinaus geben wir Konstruktionen an, mit deren Hilfe man aus semantischen Bereichen weitere semantische Bereiche und neue stetige Abbildungen bilden kann. Wenn wir uns auf so konstruierte Bereiche und Abbildungen einschränken, wissen wir ohne weiteren Nachweis, daß es sich um semantische Bereiche und stetige Abbildungen handelt.

In diesem Kapitel verzichten wir im wesentlichen auf den Nachweis der Behauptungen. In dem meisten Fällen ist der Nachweis jedoch relativ einfach.

4.1 Diskrete Bereiche

Jede Menge X zusammen mit der identischen Relation id_X bildet einen semantischen Bereich (X, id_X). Wir nennen diese semantischen Bereiche *diskrete semantische Bereiche*. Für zwei diskrete semantische Bereiche (X, id_X) und (Y, id_Y) ist jede Abbildung $f : X \to Y$ stetig.

4.2 Komposition

Seien $(D_1, \sqsubseteq_1)$, $(D_2, \sqsubseteq_2)$ und $(D_3, \sqsubseteq_3)$ semantische Bereiche und $f : D_1 \to D_2$ und $g : D_2 \to D_3$ stetige Abbildungen. Dann ist auch $g \circ f$ eine stetige Abbildung (von D_1 nach D_3).

4.3 Produkt und Projektion

Wenn $(D_1, \sqsubseteq_1), \ldots (D_n, \sqsubseteq_n)$ semantische Bereiche sind, dann ist $(D, \sqsubseteq)$ mit $D = D_1 \times D_2 \times \ldots \times D_n$ und $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$ gdw. $d_1 \sqsubseteq d'_1, \ldots, d_n \sqsubseteq d'_n$ ein semantischer Bereich. Dieser semantische Bereich heißt das (endliche) *Produkt* der semantischen Bereiche $(D_1, \sqsubseteq_1), \ldots (D_n, \sqsubseteq_n)$. Wenn jeder semantische Bereich $(D_i, \sqsubseteq_i)$ ein kleinstes Element $\bot_{D_i}$ besitzt, dann besitzt das Produkt ebenfalls ein kleinstes Element: $\bot_D = (\bot_{D_1}, \bot_{D_2}, \ldots, \bot_{D_n})$.
Für jedes $(D_i, \sqsubseteq_i)$ ist die Abbildung $\pi_i : D \to D_i$ mit $\pi_i((d_1, \ldots, d_n)) = d_i$ eine stetige Abbildung. Wir nennen sie die *Projektion* (auf die i-te Komponente). Sei nun $(E, \sqsubseteq)$ ein weiterer semantischer Bereich und seien $f_i : E \to D_i$ stetige Abbildungen. Dann ist die Abbildung $\langle f_1, \ldots, f_n \rangle : E \to D$ mit $\langle f_1, \ldots, f_n \rangle(e) = (f_1(e), \ldots, f_n(e))$ eine stetige Abbildung.

Für $\langle f_1, \ldots, f_n \rangle$ und jedes $i \in \{1, \ldots, n\}$ gilt $\pi_i \circ \langle f_1, \ldots, f_n \rangle = f_i$. Tatsächlich wird das Produkt, die Projektionsfunktion und die Produktfunktion über diese Eigenschaft definiert.

4.4 Funktionsräume

Für die Semantik besonders interessant sind Funktionsräume, da diese die semantischen Objekte enthalten, die wir Programmen als Semantik zuordnen wollen. Für zwei semantische Bereiche $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$ definieren wir $[D \to E] = \{f : D \to E \mid f \text{ ist stetig}\}$.

Zur Erinnerung: Mit $D \to E$ oder $(D \to E)$ bezeichnen wir die totalen Abbildungen von D nach E. Dagegen bezeichnet $[D \to E]$ nur die stetigen totalen Abbildungen.

Auf den stetigen Abbildungen $[D \to E]$ definieren wir die Ordnung $\sqsubseteq_{[D \to E]}$ wie folgt: Für zwei Abbildungen $f, g \in [D \to E]$ gilt $f \sqsubseteq_{[D \to E]} g$ genau dann, wenn für jedes $d \in D$ gilt $f(d) \sqsubseteq_E g(d)$, d. h. die Ordnung ist punktweise definiert.

Diese Definition entspricht gerade unserer Sichtweise bei der Definition der Ordnung auf partiellen Abbildungen: f „ist weniger stark definiert als“ g.Dabei bedeutet $f(d) = \perp_E$, daß f an der Stelle d völlig undefiniert ist. Darauf kommen wir später noch zurück.

Dann ist $([D \to E], \sqsubseteq_{[D \to E]})$ ein semantischer Bereich, der *Funktionsraum* von $(D, \sqsubseteq_D)$ nach $(E, \sqsubseteq_E)$. Wenn $(E, \sqsubseteq_E)$ ein kleinstes Element $\perp_E$ besitzt, dann besitzt auch der semantische Bereich $([D \to E], \sqsubseteq_{[D \to E]})$ ein kleinstes Element: $\perp_{[D \to E]}(d) = \perp_E$ für jedes $d \in D$.
Mit D, E und $[D \to E]$ ist natürlich auch das Produkt $[D \to E] \times D$ ein semantischer Bereich. Die folgende Abbildung $apply : ([D \to E] \times D) \to E$ mit $apply(f, d) = f(d)$ ist ebenfalls stetig.

Kurz können wir dafür auch schreiben $apply \in [([D \to E] \times D) \to E]$.

Mit D, E und F sind auch $[(F \times D) \to E]$ und $[F \to [D \to E]]$ semantische Bereiche. Die Abbildung $curry : [(F \times D) \to E] \to [F \to [D \to E]]$ ist definiert durch $curry(f) = \lambda x \in F.\lambda d \in D.f(x, d)$ für alle $f \in [(F \times D) \to E]$.

curry macht aus einer Abbildung f mit zwei Argumenten, eine Abbildung g mit einem Argument. Das Ergebnis der Abbildung g ist eine weitere Abbildung in dem verbleibenden Argument, das dann das Endergebnis der ursprünglichen Abbildung f mit beiden Argumenten liefert: $curry(f) = g$ mit $g(x)(d) = f(x, d)$ oder kurz $curry(f)(x)(d) = f(x, d)$.

Auch die Abbildung *curry* ist stetig.
Zuletzt betrachten wir die Abbildung, die jeder stetigen Abbildung den kleinsten Fixpunkt zuordnet. Für einen semantischen Bereich D mit kleinstem Element ist die Abbildung $fix : [D \to D] \to D$, die jeder stetigen Abbildung $f : D \to D$ ihren kleinsten Fixpunkt zuordnet, stetig.

4.5 Lifting

Der Fixpunktsatz von Kleene (Satz 6.10) ist nur anwendbar, wenn der zugrundeliegende semantische Bereich ein kleinstes Element besitzt. Für semantische Bereiche ohne kleinstes Element ist er nicht anwendbar. Wir können einen semantischen Bereich ohne kleinstes Element, jedoch kanonisch in einen semantischen Bereich mit kleinstem Element umwandeln. Dazu fügt man im wesentlichen nur ein kleinste Element hinzu. Das nennen wir *Lifting* und für einen semantischen Bereich D bezeichnen wir das Lifting mit $D_\perp$.
Im Detail ist die Definition des Liftings etwas aufwendiger. Sei $(D, \sqsubseteq_D)$ ein semantischer Bereich, der nicht notwendigerweise ein kleinstes Element enthalten muß. Sei $\lfloor . \rfloor : D \to X$ eine injektive Abbildung und $\perp$ ein Element, das nicht im Bild von $\lfloor . \rfloor$ vorkommt.

> *D. h. für alle $d, d' \in D$ mit $d \neq d'$ gilt auch $\lfloor d \rfloor \neq \lfloor d' \rfloor$ und es gibt kein $d \in D$ mit $\lfloor d \rfloor = \perp$.*

Dann definieren wir den semantischen Bereich $(D_\perp, \sqsubseteq_{D_\perp})$ durch $D_\perp = \{\lfloor d \rfloor \mid d \in D\} \cup \{\perp\}$ mit $\perp \sqsubseteq_{D_\perp} \hat{d}$ für jedes $\hat{d} \in D_\perp$ und $\lfloor d \rfloor \sqsubseteq_{D_\perp} \lfloor d' \rfloor$ gdw. $d \sqsubseteq_D d'$. Wir nennen $(D_\perp, \sqsubseteq_{D_\perp})$ das *Lifting* von $(D, \sqsubseteq_D)$.

> *Die Abbildung $\lfloor . \rfloor$ und das Element $\perp$ „fallen bei uns vom Himmel“. Man kann sie mit den Techniken der Allgemeinen Algebra bis auf Isomorphie eindeutig charakterisieren. Darauf gehen wir hier aber nicht näher ein. Wir gehen einfach davon aus, daß sie uns gegeben werden.*

Offensichtlich ist $(D_\perp, \sqsubseteq_{D_\perp})$ ein semantischer Bereich mit kleinstem Element $\perp$, wenn $(D, \sqsubseteq_D)$ ein semantischer Bereich ist. Darüber hinaus ist die Abbildung $\lfloor . \rfloor : D \to D_\perp$ stetig.
Sei nun $(E, \sqsubseteq_E)$ ein weiterer semantischer Bereich mit kleinstem Element $\perp_E$ und sei $f : D \to E$ eine stetige Abbildung. Dann ist die Abbildung $f^* : D_\perp \to E$ mit $f^*(\lfloor d \rfloor) = f(d)$ und $f^*(\perp) = \perp_E$ ebenfalls stetig. Wir nennen die Abbildung f^* die *strikte Erweiterung* von f.

In der Semantik versucht man meist, alle Abbildungen total zu machen. Undefinierte Resultate und Eingaben werden dann durch ein spezielles Element repräsentiert – nämlich das kleinste Element des semantischen Bereiches. Wenn wir also $\perp$ als einen undefinierten Wert ansehen, liefert f^* für eine undefinierte Eingabe auch eine undefinierte Ausgabe! Funktionen, die für eine undefinierte Eingabe eine undefinierte Ausgabe liefern nennt man strikt.

Für die Abbildung f ist die undefinierte Eingabe $\perp$ noch gar nicht zulässig. Durch die strikte Erweiterung f^* wird dies zum Ausdruck gebracht.

Noch allgemeiner können wir * als einen Operator * bzw. eine Abbildung auffassen, die jede stetige Abbildung f auf die stetige Abbildung f^* abbildet. Um das deutlicher zu machen, können wir auch schreiben $f^* = (f)^*$. Dann ist $(.)^*$ eine Abbildung aus $[D \to E] \to [D_\perp \to E]$. Diese Abbildung ist sogar stetig.
Wenn f durch einen λ-Ausdruck $f \equiv \lambda x \in D.e$ definiert ist, schreibt man für die Anwendung der strikten Erweiterung von f auf ein Element $d \in D_\perp$ oft auch $f^*(d) \equiv \text{let}\, x \Leftarrow d.e$.

In dieser Notation kann man let$x \Leftarrow d$ als eine Zuweisung lesen. Dabei wird der Wert von d an x übergeben, wenn er definiert (also nicht $\perp$ ist) und der Ausdruck e wird dann mit diesem Wert ausgewertet. Wenn d jedoch nicht definiert ist (d. h. die Auswertung nicht terminiert) ist, scheitert bereits die Zuweisung und das Ergebnis ist undefiniert. Am besten sieht man den Unterschied bei einer Abbildung, die ein Konstantes Ergebnis besitzt: $f = \lambda x \in \mathbb{N}.1$. Dann gilt $f^*(\perp) \equiv$ let$x \Leftarrow \perp .1 = \perp$. Im Gegensatz dazu würde für $f' \equiv \lambda x \in D_\perp.\lfloor 1 \rfloor$ gelten $f'(\perp) = \lfloor 1 \rfloor$.

Für $f^*(7) \equiv$ let$x \Leftarrow 7.1 = \lfloor 1 \rfloor$, denn das Konstrukt let bildet implizit von $\mathbb{N}$ auf $\mathbb{N}_\perp$ ab.

Durch die Konstruktion des Liftings können wir nun alle Mengen bzw. die entsprechenden diskreten semantischen Bereiche, die wir in einer Programmiersprache benutzen wollen, zu einem semantischen Bereich mit kleinstem Element machen. Ein Beispiel ist $\mathbb{B}_\perp$. Alle Operationen auf $\mathbb{B}$ können wir mit Hilfe der strikten Erweiterung auf $\mathbb{B}_\perp$ „liften“. Beispielsweise ist $\vee^*$ die strikte Erweiterung der booleschen Operation $\vee$. Mit der oben eingeführten Notation können wir schreiben $b \vee^* b' \equiv \text{let}\, x \Leftarrow b.\text{let}\, x' \Leftarrow b'.x \vee x'$.

4.6 Endliche Summe

Als letzte Konstruktion führen wir disjunkte Vereinigung von semantischen Bereichen ein. Für Mengen $D_1, \ldots, D_n$ bezeichnet $D_1 + \ldots + D_n$ die *disjunkte*

Vereinigung. Das entspricht den variant records in PASCAL oder einer disjunkten und vollständigen Vererbungsbeziehung in UML. Es gibt verschiedene Möglichkeiten, die disjunkte Vereinigung zu definieren. Die eleganteste Methode benutzt wieder die Techniken der Allgemeinen Algebra. Wir benutzen hier die etwas unelegantere Technik der expliziten Definition. Für eine endliche Folge von Mengen $D_1, \ldots, D_n$ definieren wir die disjunkte Vereinigung wie folgt: $D = \{(i, d) \mid i \in \{1, \ldots, n\}, d \in D_i\}$. Die erste Komponente gibt an, aus welcher Menge das Element kommt, die zweite bezeichnet das Element selbst. Auf diese Weise können wir für jedes Element eindeutig sagen, aus welcher Menge es kommt, selbst dann, wenn die Ausgangsmengen nicht disjunkt sind. Wir machen die Mengen also mit Hilfe der ersten Komponente explizit disjunkt.
Für jedes $i \in \{1, \ldots, n\}$ definieren wir eine Abbildung $in_i : D_i \to D$ mit $in_i(d) = (i, d)$, die Injektion[2] von den einzelnen Mengen D_i in D.
Für semantische Bereiche $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$ definieren wir den semantischen Bereich $(D, \sqsubseteq_D)$ mit $(i, d) \sqsubseteq (j, d')$ genau dann, wenn $i = j$ und $d \sqsubseteq_{D_i} d'$. Wir nennen diesen semantischen Bereich die *Summe* der semantischen Bereiche $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$. Für $n \geq 2$ und $D_i \neq \emptyset$ besitzt die Summe kein kleinstes Element, selbst dann nicht, wenn jeder einzelne semantische Bereich $(D_i, \sqsubseteq_{D_i})$ ein kleinstes Element besitzt.
Für stetige Abbildungen $f_1 : D_1 \to E, \ldots, f_n : D_n \to E$ ist die Abbildung $[f_1, \ldots f_n] : D \to E$ definiert durch $[f_1, \ldots f_n]((i, d)) = f_i(d)$. Diese Abbildung ist stetig.

Es gilt ähnlich wie beim Produkt und den Projektionen $[f_1, \ldots, f_n] \circ in_i = f_i$.

5 Eine Sprache für stetige Abbildungen

Aufbauend auf den Konstrukten des vorangegangenen Abschnittes definieren wir nun ein Sprache, die es uns erlaubt, stetige (und nur stetige) Abbildungen zu definieren. Allerdings halten wir uns nicht mit syntaktischen Details auf. Im wesentlichen werden wir dazu die λ-Ausdrücke benutzen: $\lambda x \in D.e$. Dabei müssen wir aber darauf achten, daß der Ausdruck e so gebaut ist, daß $\lambda x \in D.e$ immer stetig ist. Der Ausdruck muß also in jeder Variable, die in ihm vorkommt, stetig sein. Wir nennen einen Ausdruck *stetig*, wenn für jede Variable x die Abbildung $\lambda x \in D.e$ stetig ist.

[2] Über die Charakterisierung dieser Abbildungen würde man die disjunkte Vereinigung mit Hilfe der Allgemeinen Algebra definieren.

Nachfolgend geben wir nun Regeln zur Konstruktion von stetigen Ausdrücken an.

Variablen Für eine Variable x vom Typ $(D, \sqsubseteq_D)$ ist x ein stetiger Ausdruck vom Typ $(D, \sqsubseteq_D)$.

Konstanten Jedes Element $d \in D$ eines semantischen Bereiches $(D, \sqsubseteq_D)$ ist ein stetiger Ausdruck vom Typ $(D, \sqsubseteq_D)$. Insbesondere sind $\bot_D$, *true*, *true*, 0, 1, ... stetige Ausdrücke entsprechenden Typs. Aber auch die zuvor definierten Abbildungen *apply*, *curry* und fix sind stetige Ausdrücke.

> *fix ist eine Konstante des semantischen Bereiches $[[D \to D] \to D]$, curry ist eine Konstante des semantischen Bereiches $[[(F \times D) \to E] \to [F \to [D \to E]]$ und apply des semantischen Bereiches $[([D \to E] \times D) \to E$, wobei wir streng genommen die Funktionen noch mit den entsprechenden semantischen Bereichen D, E und F indizieren müßten.*

Hier definieren wir sogar noch eine weitere Funktion, die *Fallunterscheidung*, eine Abbildung in drei Argumenten: $. \Rightarrow .|. : [(\mathbb{B}_\bot \times D \times D) \to D]$ wobei $(D, \sqsubseteq_D)$ ein semantischer Bereich mit kleinstem Element $\bot_D$ ist. Diese Abbildung ist definiert durch:

$$b \Rightarrow e_1|e_2 = \begin{cases} e_1 & \text{falls } b = \lfloor true \rfloor \\ e_2 & \text{falls } b = \lfloor false \rfloor \\ \bot_D & \text{falls } b = \bot_{\mathbb{B}_\bot} \end{cases}$$

Dies ist ein Beispiel für eine nicht-strikte Abbildung. Denn es kann sein, daß e_1 undefiniert ist, die Fallunterscheidung aber trotzdem ein definiertes Ergebnis (ungleich $\bot_{\mathbb{B}_\bot}$) liefert, nämlich dann, wenn b den Wert *false* hat.

Tupel Wenn $e_1, \ldots, e_n$ stetige Ausdrücke vom Typ $D_1, \ldots, D_n$ sind, dann ist $(e_1, \ldots, e_n)$ ein stetiger Ausdruck vom Typ $D_1 \times \ldots \times D_n$.

Funktionsanwendung Wenn f eine stetige Abbildung aus dem Bereich $[D \to E]$ ist, und e ein stetiger Ausdruck vom Typ D, dann ist $f(e)$ ein stetiger Ausdruck vom Typ E.

λ-Abstraktion Wenn e ein stetiger Ausdruck vom Typ E ist und x eine Variable vom Typ D, dann ist $\lambda x \in D.e$ ein stetiger Ausdruck vom Typ $[D \to E]$.

Mit Hilfe der obigen Konstrukte und Notationen können wir nun stetige Abbildungen definieren. Wir können sogar rekursive Abbildungen definieren. Dies zeigen wir anhand des Beispiels der Fakultätsfunktion. Zunächst geben wir eine pseudoprogrammiersprachliche Formulierung an:

```
function fac(x: nat) : nat {
  if x = 0 then 1
           else x * fac(x-1)
```

Diese Funktion drücken wir nun mit Hilfe unserer Sprache aus. Dazu definieren wir zunächst:

$$FAC : [\mathbb{N} \to \mathbb{N}_\perp] \to [\mathbb{N} \to \mathbb{N}_\perp]$$

mit

$$FAC \equiv \lambda f \in [\mathbb{N} \to \mathbb{N}_\perp].\lambda x \in \mathbb{N}.\lfloor x = 0 \rfloor \to \lfloor 1 \rfloor || \lfloor x \rfloor \cdot^* f(x-1)$$

$\cdot^*$ bezeichnet dabei die Erweiterung des Produkts $\cdot$ von $\mathbb{N}$ auf $\mathbb{N}_\perp$.

Dann gilt $fac \equiv fix(FAC)$ und $fac(n) \equiv apply(fix(FAC), n))$.

6 Zusammenfassung

In diesem Kapitel haben wir uns mit Sätzen über die Existenz von Fixpunkten von Abbildungen beschäftigt. Es hat sich herausgestellt, daß sich der klassische Fixpunktsatz von Knaster und Tarski für unsere Zwecke nicht so gut eignet, da die zugrundeliegenden Strukturen in der Informatik meist keine vollständigen Verbände sind (und auch nicht vernünftig in solche eingebettet werden können).
Deshalb haben wir einen Fixpunktsatz für semantische Bereiche und stetige Abbildungen betrachtet. Der hat darüber hinaus den Vorteil, daß sich damit der Fixpunkt einer Abbildung approximieren läßt.
Zuletzt haben wir dann Konstruktionsregeln und eine Sprache betrachtet, mit der wir immer innerhalb der stetigen Abbildungen und der semantischen Bereiche bleiben. Das spart uns den Aufwand nachzuweisen, daß die Fixpunkte existieren und insbesondere die Abbildung *fix* immer wohldefiniert

ist. Am Ende haben wir sogar gesehen, daß wir innerhalb dieser Notation mit Hilfe von *fix* sogar rekursive Abbildungen definieren können. Mit Hilfe dieser Sprache kann man dann auch einer funktionalen Programmiersprache (mit Rekursion eine Semantik zuordnen).

Wir haben hier nur einen kurzen Überblick über diese Konstruktion gegeben; genauer Informationen findet man unter dem Stichwort typisierter Lambda-Kalkül.

Kapitel 7

Axiomatische Semantik

In diesem Kapitel stellen wir die *axiomatische Semantik* vor. Die axiomatische Semantik definiert nicht das Verhalten eines Programms, sondern definiert seine Eigenschaften. Diese Eigenschaften werden mit Hilfe von Regeln formuliert. Diese Regeln können umgekehrt auch als Beweisregeln für diese Eigenschaften aufgefaßt werden.
Damit stellt die axiomatische Semantik den Zusammenhang zur Verifikation von Programmen her. Deshalb gehen wir hier im Rahmen der Vorlesung Semantik nur am Rande auf die axiomatische Semantik ein.
Die axiomatische Semantik – genauer die Äquivalenz dieser Semantik zur operationalen und mathematischen Semantik – hat außerdem sehr tiefgreifende theoretische Konsequenzen: mit ihrer Hilfe läßt sich der berühmte Unvollständigkeitssatz von Gödel (bzw. die Nicht-Axiomatisierbarkeit der Arithmetik) beweisen. Auch darauf können wir hier leider nicht eingehen.

1 Motivation

Wie gesagt definiert die axiomatische Semantik die Eigenschaften eines Programms. Dazu müssen wir uns also zunächst überlegen, welche Eigenschaften eines Programms wir beschreiben wollen. Wir werden uns dabei – ganz klassisch – auf den Zusammenhang zwischen Eingabe und Ausgabe bzw. zwischen Anfangs- und Endzuständen beschränken. Beispielsweise gilt für das folgende Programm die folgende *Zusicherung*:

$\{x = i \wedge i \geq 0 \wedge y = 1\}$
while $1 \leq$ x **do** $\ulcorner$ y:= y$*$x; x:= x $-$ 1 $\lrcorner$

$\{y = i!\}$

Diese Zusicherung kann man wie folgt lesen: Wenn die Anweisung
while 1 ≤x **do** ⌜ y:= y∗x; x:= x − 1 ⌟
in einem Zustand gestartet wird, für den $x = i \wedge i \geq 0 \wedge y = 1$ gilt, und das Programm irgendwann terminiert, dann gilt im Endzustand $y = i!$.

> *Wichtig ist, daß die Zusicherung NICHT fordert, daß die Anweisung terminiert, sondern nur etwas über den Endzustand sagt, wenn sie terminiert. Dies entspricht also gerade der partiellen Korrektheit von Programmen. Es gibt andere Formen der Zusicherung, die auch verlangen, daß die Anweisung auch terminiert. Aber die betrachten wir hier nicht.*

Dabei sind x und y *Programmvariablen*, weil sie im Programm vorkommen. Sie nehmen also jeweils den Wert der entsprechenden Variable im Anfangszustand bzw. im Endzustand an. Die Variable i dagegen kommt im Programm nicht vor; wir nennen sie eine *Logikvariable*. Sie kann jeden beliebigen Wert erhalten, der aber im Anfangs- und Endzustand gleich ist. Auf diese Weise können wir ausdrücken, daß am Ende die Variable y den Wert von $x!$ des Anfangsbestands hat. Denn wir wissen aufgrund der Bedingung $x = i$, daß i den Wert haben muß, den x am Anfang besitzt. Aufgrund der Bedingung $y = i!$ wissen wir, daß y am Ende die Fakultät des Wertes der Variable x vom Anfang hat.
Allgemein besteht eine Zusicherung also aus drei Teilen, der *Vorbedingung*, der Anweisung und der *Nachbedingung*: $\{A\}\, c\, \{B\}$, wobei A und B prädikatenlogische Formeln sind und c eine Anweisung. In den prädikatenlogischen Formeln dürfen auch Quantoren vorkommen, wobei wir nur über Logikvariablen quantifizieren dürfen. Hier ist ein weiteres Beispiel für eine solche Zusicherung (wobei $b(x)$ ein beliebiger boolescher Ausdruck ist, in der die Variable x vorkommt):
$\{x = 0\}$ **while** ¬b(x) **do** x:= x + 1 $\{b(x) \wedge \forall i.(0 \leq i \wedge i < x) \Rightarrow \neg b(i)\}$.

> *Nochmal zur Erinnerung: Die Zusicherung sagt nicht, daß das Programm in jedem Zustand, der die Vorbedingung erfüllt, terminiert. Beispielsweise terminiert die Anweisung in unserem Beispiel nicht, wenn b nie wahr wird. Die Zusicherung besagt nur, daß im Endzustand die Nachbedingung gilt, wenn denn der Endzustand erreicht wird.*
>
> *Eine interessante Frage ist dann, für welche Anweisungen c die Zusicherung* {true}c{false} *gilt? Es gibt tatsächlich Anweisungen, für die diese Zusicherung gilt.*

Die axiomatische Semantik macht nun nichts anderes, als Regeln anzugeben, um derartige Zusicherungen herzuleiten. Natürlich sollten diese Regeln so geartet sein, daß alle Zusicherungen, die wir mit den Regeln herleiten können, auch im oben angedeuteten intuitiven Sinne gelten. Das nennt man die *Korrektheit* der Regeln. Umgekehrt sollten auch alle Zusicherungen, die intuitiv gelten, mit Hilfe der Regeln hergeleitet werden können. Das nennt man die *Vollständigkeit* der Regeln. Damit wir diese Begriffe formalisieren können, werden wir zunächst die Gültigkeit einer Zusicherung auf eine formale Grundlage stellen. Wir werden dies mit Hilfe der operationalen Semantik tun.

2 Grundlagen der Prädikatenlogik

Bevor wir die Gültigkeit von Zusicherungen formulieren können, wiederholen wir einige wichtige Begriffe aus der Prädikatenlogik und führen einige Notationen ein, die wir später zur Definition der Gültigkeit von Zusicherungen benötigen.

2.1 Logikvariablen, Ausdrücke und Formeln

In den Beispielen haben wir gesehen, wie wir prädikatenlogische Formeln aufbauen können. Neben den Programmvariablen $\mathbb{V}$ dürfen in prädikatenlogischen Ausdrücken auch logische Variablen vorkommen. Für solche Variablen definieren wir eine neue syntaktische Menge $\mathbb{L}$, für deren Elemente wir die Bezeichnungen i, i_1, i_2 und i' und i'' reservieren. Wir gehen im folgenden davon aus, daß die Logikvariablen und die Programmvariablen disjunkt sind (d. h. $\mathbb{V} \cap \mathbb{L} = \emptyset$).
Aus diesen bauen wir dann die arithmetischen Ausdrücke $Aexp_l$ mit logischen Variablen auf, wie wir das auch schon bei den arithmetischen Ausdrücken in Anweisungen formalisiert haben. Aus diesen können wir dann die prädikatenlogischen Formeln $Form$ aufbauen:

$$Aexp_l\colon \quad a ::= \quad n \mid v \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$$

$$Form\colon \quad F ::= \quad t \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg F_0 \mid F_0 \wedge F_1 \mid F_0 \vee F_1 \mid \forall i.F \mid \exists i.F$$

In unseren Beispielen werden wir auch weitere boolesche Operatoren (insbesondere die Implikation) und arithmetische Operatoren zulassen (wie Bei-

spielsweise die Fakultätsfunktion).

2.2 Belegungen und Auswertung von Ausdrücken

Eine Abbildung $\beta : \mathbb{L} \to \mathbb{Z}$ nennen wir eine *Belegung* der Logikvariablen. In einem gegebenen Zustand σ und für eine gegebene Belegung β können wir nun Ausdrücke mit Logikvariablen auswerten. Wir schreiben dafür in Anlehnung an die entsprechende Notation der mathematischen Semantik für Ausdrücke: $\mathcal{A}[\![a]\!](\beta, \sigma)$.

Die Definition der Auswertungsfunktion ist eine einfache Übungsaufgabe. Der Vollständigkeit halber geben wir sie hier trotzdem an:

- $\mathcal{A}[\![n]\!](\beta, \sigma) = n$
- $\mathcal{A}[\![v]\!](\beta, \sigma) = \sigma(v)$
- $\mathcal{A}[\![i]\!](\beta, \sigma) = \beta(i)$
- $\mathcal{A}[\![a_0 + a_1]\!](\beta, \sigma) = \mathcal{A}[\![a_0]\!](\beta, \sigma) + \mathcal{A}[\![a_1]\!](\beta, \sigma)$
- $\mathcal{A}[\![a_0 - a_1]\!](\beta, \sigma) = \mathcal{A}[\![a_0]\!](\beta, \sigma) - \mathcal{A}[\![a_1]\!](\beta, \sigma)$
- $\mathcal{A}[\![a_0 * a_1]\!](\beta, \sigma) = \mathcal{A}[\![a_0]\!](\beta, \sigma) \cdot \mathcal{A}[\![a_1]\!](\beta, \sigma)$

Man kann auch leicht zeigen, daß für $a \in Aexp$ gilt $\mathcal{A}[\![c]\!](\beta, \sigma) = \mathcal{A}[\![c]\!](\sigma)$.

2.3 Gültigkeit einer prädikatenlogischen Aussage

Für einen gegebenen Zustand σ und eine gegebene Belegung β definieren, ob eine prädikatenlogische Formel gilt. Wenn eine Formel F in β und σ gilt, schreiben wir $\beta, \sigma \models F$.

Auch die Gültigkeitsbeziehung *können wir einfach induktiv über den Aufbau der Formel definieren:*

- $\beta, \sigma \models t$ *gilt genau dann, wenn* $t \equiv \text{true}$ *gilt.*
- $\beta, \sigma \models a_0 = a_1$ *gilt genau dann, wenn* $\mathcal{A}[\![a_0]\!](\beta, \sigma) = \mathcal{A}[\![a_1]\!](\beta, \sigma)$ *gilt.*
- $\beta, \sigma \models a_0 \leq a_1$ *gilt genau dann, wenn* $\mathcal{A}[\![a_0]\!](\beta, \sigma) \leq \mathcal{A}[\![a_1]\!](\beta, \sigma)$ *gilt.*
- $\beta, \sigma \models F_0 \wedge F_1$ *gilt genau dann, wenn* $\beta, \sigma \models F_0$ *und* $\beta, \sigma \models F_1$ *gilt.*
- $\beta, \sigma \models F_0 \vee F_1$ *gilt genau dann, wenn* $\beta, \sigma \models F_0$ *oder* $\beta, \sigma \models F_1$ *gilt.*
- $\beta, \sigma \models \neg F_0$ *gilt genau dann, wenn nicht* $\beta, \sigma \models F_0$ *gilt.*
- $\beta, \sigma \models \forall i.F_0$ *gilt genau dann, wenn für alle Belegungen* β' *mit* $\beta'(i') = \beta(i')$ *für alle* $i' \not\equiv i$ *auch* $\beta', \sigma \models F_0$ *gilt.*

- $\beta, \sigma \models \exists i.F_0$ *gilt genau dann, wenn für eine Belegungen* β' *mit* $\beta'(i') = \beta(i')$ *für alle* $i' \not\equiv i$ *auch* $\beta', \sigma \models F_0$ *gilt.*

Eine prädikatenlogische Formel F heißt *allgemeingültig*, wenn für alle Belegungen β und alle Zustände σ gilt: $\beta, \sigma \models F$. Wir schreiben dann auch $\models F$.

2.4 Substitution

Zuletzt definieren wir die Substitution einer Programmvariablen in einer Formel. Für eine Programmvariable $v \in \mathbb{V}$, einen Ausdruck $a \in Aexp$ (ohne Logikvariablen) und eine Formel F bezeichnet $F[a/v]$ diejenige Formel, in der jedes Vorkommen der Variablen v durch den Ausdruck a ersetzt wird. Dies nennen wir eine *Substitution.*

Der Vollständigkeit halber geben wir hier eine induktive Definition für die Substitution an:

- $n[a/v] \equiv n$.
- $v'[a/v] \equiv v$ *falls* $v \not\equiv v'$ *und* $v'[a/v] \equiv a$ *falls* $v \equiv v'$.
- $i[a/v] \equiv i$.
- $(a_0 + a_1)[a/v] \equiv (a_0[a/v] + a_1[a/v])$.
- $(a_0 - a_1)[a/v] \equiv (a_0[a/v] - a_1[a/v])$.
- $(a_0 * a_1)[a/v] \equiv (a_0[a/v] * a_1[a/v])$.
- $t[a/v] \equiv t$.
- $(a_0 = a_1)[a/v] \equiv (a_0[a/v] = a_1[a/v])$.
- $(a_0 \leq a_1)[a/v] \equiv (a_0[a/v] \leq a_1[a/v])$.
- $(F_0 \wedge F_1)[a/v] \equiv (F_0[a/v] \wedge F_1[a/v])$.
- $(F_0 \vee F_1)[a/v] \equiv (F_0[a/v] \vee F_1[a/v])$.
- $(\neg F)[a/v] \equiv \neg(F[a/v])$.
- $(\forall i.F)[a/v] \equiv \forall i.(F[a/v])$.
- $(\exists i.F)[a/v] \equiv \exists i.(F[a/v])$.

In der Definition der Substitution haben wir ganz bewußt darauf verzichtet, Ausdrücke mit Logikvariablen für eine Variable einzusetzen. Es dürfen nur arithmetische Ausdrücke substituiert werden. Denn sonst hätten wir darauf achten müssen, daß Logikvariablen nicht unter die Bindung eines Quantors geraten. Dies hätte zu einer unnötig komplizierten Definition geführt, die uns hier keinen weiteren Nutzen gebracht hätte.

Die Notation für die Substitution erinnert bewußt an die Zustandsmodifikation. Es gibt aber einen wichtigen Unterschied: Die Substitution operiert syntaktisch auf einer Formel, also einem syntaktischen Objekt. Die Zustandsmodifikation operiert auf einem Zustand, also einem semantischen Objekt. Es besteht aber trotzdem ein enger Zusammenhang zwischen den beiden Konzepten: das *Substitutionslemma*.

Lemma 7.1 (Substitutionslemma)
Sei $F \in Form$ eine Formel, $v \in \mathbb{V}$ eine Programmvariable, $a \in Aexp$ ein Ausdruck, β eine Belegung und σ ein Zustand. Dann gilt $\beta, \sigma \models F[a/v]$ genau dann, wenn $\beta, \sigma[\sigma(a)/v] \models F$ gilt.

Beweis: Das Substitutionslemma läßt sich leicht durch Induktion über den Aufbau der Ausdrücke und Formeln beweisen. □

3 Zusicherungen

Mit diesen Begriffen und der operationalen Semantik können wir nun den Begriff der Zusicherung und unsere informelle Beschreibung ihrer Gültigkeit formalisieren.

Definition 7.2 (Zusicherung)
Für zwei prädikatenlogische Formeln A und B und eine Anweisung c nennen wir $\{A\}\, c\, \{B\}$ eine *Zusicherung*.
Die Zusicherung heißt *gültig*, wenn für jede Belegungen β und jeden Zustand σ mit $\beta, \sigma \models A$ und jeden Zustand σ' mit σ' mit $\langle c, \sigma\rangle \to \sigma'$ auch gilt $\beta, \sigma' \models B$. Für eine gültige Zusicherung schreiben wir auch $\models \{A\}\, c\, \{B\}$.

> *Der Wert der logischen Variablen ändert sich bei den beiden Interpretationen von A und B nicht (β bleibt gleich); der Wert der Programmvariablen ändert sich dagegen (die Gültigkeit von A wird bzgl. σ überprüft, die Gültigkeit von B bezüglich σ').*

Wir geben nun Beweisregeln an, mit deren Hilfe man die Gültigkeit von Zusicherungen beweisen kann. Am Ende werden wir feststellen, daß wir mit diesen Regel genau diejenigen Zusicherungen herleiten können, die auch gültig sind. Da wir das aber a priori nicht wissen benutzen wir für die durch Regeln herleitbaren Zusicherungen ein anderes Symbol $\vdash \{A\}\, c\, \{B\}$. Diese Regel

werden *Zusicherungslogik* oder oft auch *Hoare-Kalkül* genannt, da die Regel auf C.A.R. Hoare zurück gehen [7].

Definition 7.3 (Zusicherungslogik (Hoare-Kalkül))
Die *Zusicherungslogik* besteht aus den folgenden Regeln:

$$\frac{}{\{A\}\,\mathbf{skip}\,\{A\}}$$

$$\frac{}{\{A[a/v]\}\,v{:=}a\,\{A\}}$$

$$\frac{\{A\}\,c_0\,\{C\} \quad \{C\}\,c_1\,\{B\}}{\{A\}\,c_0;c_1\,\{B\}}$$

$$\frac{\{A \wedge b\}\,c_0\,\{B\} \quad \{C \wedge \neg b\}\,c_1\,\{B\}}{\{A\}\,\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\,\{B\}}$$

$$\frac{\{A \wedge b\}\,c\,\{A\}}{\{A\}\,\mathbf{while}\ b\ \mathbf{do}\ c\,\{A \wedge \neg b\}}$$

$$\frac{\models A \Rightarrow A' \quad \{A'\}\,c\,\{B'\} \quad \models B' \Rightarrow B}{\{A\}\,c\,\{B\}}$$

Wenn eine Zusicherung $\{A\}\,c\,\{B\}$ mit diesen Regeln herleitbar ist, schreiben wir $\vdash \{A\}\,c\,\{B\}$.

In der Zusicherungslogik gibt es für jedes Konstrukt unserer Programmiersprache IMP eine Regel. Dazu kommt noch eine weitere Regel, die es erlaubt die Zusicherungen für eine Anweisung zu verändern. Sie heißt *Abschwächungsregel*. weil eine bereits bewiesene Zusicherung damit abgeschwächt werden kann. Die Regeln für die einzelnen Konstrukte unserer Programmiersprache stellen gewissermaßen die Bausteine für Beweise dar; die Abschwächungsregel stellt den Mörtel dar, der es erlaubt, die Bausteine zusammenzukleben und gewisse Anpassungen vorzunehmen (vgl. Bsp.).
Die Regeln für die meisten Anweisungen sind unmittelbar einsichtig. Die einzige Anweisung, zu der man etwas sagen sollte, ist die Regel für die Zuweisung. Zunächst wundert man sich sicher, warum die Regel „rückwärts“ formuliert ist und nicht vorwärts. Diese Regel wird deshalb oft auch *Rückwärtsregel* genannt. Man kann sich aber leicht klar machen, daß die naive Vorwärtsregel

$$\frac{}{\{\mathit{true}\}\,v{:=}a\,\{v = a\}}$$

falsch ist. Dazu muß man sich nur die Zuweisung x:= x+1 ansehen. Eine korrekte (und hinreichend ausdrucksmächtige) „Vorwärtsregel“ wäre komplizierter. Außerdem ist die Rückwärtsregel viel natürlicher – wenn man sich erstmal an diesen Gedanken gewöhnt hat. Beim praktischen Beweisen führt die Rückwärtsregel dazu, daß man Beweise von hinten nach vorne konstruiert. Das ist dann aber schon Gegenstand einer Verifikationsvorlesung.

Die zweite Regel, die einer Erläuterung bedarf, ist die Abschwächungsregel. In dieser Regel kommen nämlich Voraussetzungen vor, die gar nicht innerhalb der Zusicherungslogik ableitbar sind, nämlich die Implikationen $A \Rightarrow A'$ und $B' \Rightarrow B$. Diese müssen mit den klassischen Regeln der Logik bewiesen werden. Dies deuten wir dadurch an, daß wir das Symbol $\models$ davor setzen; das bedeutet, daß die Implikationen allgemeingültig sein müssen. Regeln zum Beweis geben wir dafür aber nicht an[1].

Auch wenn Programmverifikation nicht unser Thema ist, wollen wir wenigstens ein Programm mit Hilfe des Hoare-Kalküls beweisen.

Beispiel 7.1 (Fakultätsfunktion)
Wir beweisen die folgende Zusicherung (vgl. Abschnitt 1):

$\{x = i \wedge i \geq 0 \wedge y = 1\}$
while 1 $\leq$x **do** $\ulcorner$ y:= y$*$x; x:= x $-$ 1 $\lrcorner$
$\{y = i!\}$

Den Beweis formulieren wir allerdings nicht in Form eines Herleitungsbaumes, sondern, indem wir die Zusicherungen direkt in die Anweisung hinein schreiben. Der Herleitungsbaum läßt sich daraus jedoch relativ einfach ge-

[1]Tatsächlich ist es unmöglich einen vollständigen Satz von Regeln zum Beweis aller allgemeingültigen Aussagen anzugeben. Das ist eine Konsequenz des Unvollständigkeitssatzes von Gödel. Weil wir uns um diese Problematik hier nicht kümmern wollen, geben wir diese Voraussetzung hier semantisch an.

winnen.

$1:$ $\{x = i \wedge i \geq 0 \wedge y = 1\}$
$2:$ $\{y * x! = i! \wedge 0 \leq x\}$
$3:$ **while** $1 \leq$x **do**
$4:$ $\{y * x! = i! \wedge 0 \leq x \wedge 1 \leq x\}$
$5:$ $\{(y * x) * (x - 1)! = i! \wedge 0 \leq (x - 1)\}$
$6:$ y:= y * x;
$7:$ $\{y * (x - 1)! = i! \wedge 0 \leq (x - 1)\}$
$8:$ x:= x − 1;
$9:$ $\{y * x! = i! \wedge 0 \leq x\}$
$10:$ $\{y * x! = i! \wedge 0 \leq x \wedge \neg 1 \leq x\}$
$11:$ $\{y * x! = i! \wedge x = 0\}$
$12:$ $\{y = i!\}$

Die Frage ist, wie kommt man auf einen derartigen Beweis und wie entwickelt man ihn. Die wichtigste Aufgabe dabei ist es eine *Schleifeninvariante* zu entdecken. In unserem Beispiel ist das die Aussage $y * x! = i! \wedge 0 \leq x$, denn die bleibt bei jedem Schleifendurchlauf gültig, wenn sie vorher gilt. Diese schreiben wir also zunächst in das Programm hinein (Zeile 9). Ausgehend von Zeile 9 können wir dann durch zweimalige Anwendung der Rückwärtsregel die Zeilen 7 und 5 ergänzen; Zeile 5 können wir durch Anwendung der Abschwächungsregel zu Zeile 4 modifizieren. Mit Zeilen 4 und 9 können wir dann die Schleifenregel mit $A \equiv y*x! = i! \wedge 0 \leq x$ anwenden und wir erhalten die Zeilen 2 und 10. Die Schleifenregel entspricht also exakt der Anwendung der Schleifeninvariante! Zuletzt wenden wir die Abschwächungsregel an und erhalten die Zeilen 1 und 11. Eine weitere Abschwächung (die wir nur aus didaktischen Gründen nicht sofort im ersten Schritt durchgeführt haben) liefert uns Zeile 12. Der Beweis ist also fertig.
Diesen Beweis könnten wir nun in einen Herleitungsbaum umwandeln, was aber relativ langweilig ist.

4 Korrektheit und Vollständigkeit

Die Regeln der Zusicherungslogik können wir einerseits als eine weitere Semantik für Anweisungen auffassen, die *axiomatische Semantik* für Anweisungen. Andererseits können wir sie als Beweiskalkül für die Gültigkeit von Zusicherungen auffassen. Im ersten Fall sollten wir tunlichst beweisen, daß die axiomatische Semantik äquivalent zur operationalen oder zur mathema-

tischen Semantik ist. Im zweiten Fall sollten wir nachweisen, daß der Beweiskalkül korrekt und vollständig ist. Im Endeffekt ist das aber nur eine Frage der Sichtweise, denn die Gültigkeit einer Zusicherung wurde ja bereits mit Hilfe der operationalen Semantik definiert. Die Arbeit bleibt ist also dieselbe.

Zunächst beschäftigen wir uns mit der *Korrektheit* der Zusicherungslogik. Korrektheit bedeutet, daß jede Zusicherung, die wir mit Hilfe der Regeln herleiten können, auch gilt. Kurz können wir dies wie folgt formulieren:

$$\vdash \{A\}\, c\, \{B\} \quad \Rightarrow \quad \models \{A\}\, c\, \{B\}$$

Satz 7.4 (Korrektheit der Zusicherungslogik)
Für jede Zusicherung $\{A\}\, c\, \{B\}$ mit $\vdash \{A\}\, c\, \{B\}$ gilt auch $\models \{A\}\, c\, \{B\}$.

Beweis: Diese Aussage läßt sich durch Induktion über die Regeln der Zusicherungslogik unter Anwendung der Definition der Gültigkeit und der operationalen Semantik einfach beweisen. Der interessanteste Fall ist der Beweis der Rückwärtsregel; es stellt sich nämlich heraus, daß diese Regel exakt dem Substitutionslemma (Lemma 7.1) entspricht. Die Durchführung des Beweises ist eine relativ einfache Übungsaufgabe. □

Nun wissen wir also, daß alles, was wir mit der Zusicherungslogik beweisen können, auch wirklich stimmt. Das sollte niemanden wirklich überraschen. Die viel spannendere Frage ist, ob wir auch alle gültigen Zusicherungen beweisen können (wenn wir uns nicht zu blöd anstellen). Wenn wir wirklich alles beweisen können, was gilt, dann heißt der Kalkül vollständig. Kurz können wir die Vollständigkeit wie folgt formulieren:

$$\models \{A\}\, c\, \{B\} \quad \Rightarrow \quad \vdash \{A\}\, c\, \{B\}$$

Die Vollständigkeit ist also genau die umgekehrte Richtung der Implikation für die Korrektheit.

Satz 7.5 (Vollständigkeit der Zusicherungslogik)
Für jede Zusicherung $\{A\}\, c\, \{B\}$ mit $\models \{A\}\, c\, \{B\}$ gilt auch $\vdash \{A\}\, c\, \{B\}$.

Beweis: Diese Aussage können wir im Rahmen dieser Vorlesung nicht beweisen. Der Beweis ist sehr aufwendig. Ein Indiz dafür ist, daß aus dem Beweis dieses Satzes der Unvollständigkeitssatz von Gödel als einfache Folgerung abfällt (darauf gehen wir hier aber nicht näher ein). Im Skript zur 4-stündigen Vorlesung aus dem WS 2002/03 ist der Beweis jedoch enthalten; auch

im Buch von Winskel [11] ist er zu finden. □

Der obige Vollständigkeitssatz ist jedoch noch mit großer Vorsicht zu genießen. Denn in den Regel der Zusicherungslogik haben wir ja nur den Anteil, der über Zusicherungen redet, durch syntaktische Regeln beschrieben. Für die Allgemeingültigkeit der Implikationen in der Abschwächungsregel haben wir keine syntaktischen Regeln formuliert – die Verantwortung dafür haben wir an die Logiker abgegeben. Die Zusicherungslogik ist also nicht *effektiv*. Die Frage ist nun, ob wir einen Satz von syntaktischen Regeln angeben können, mit dem alle allgemeingültigen Implikationen hergeleitet werden können. Man kann zeigen (zum Beispiel mit Hilfe unseres obigen Vollständigkeitssatzes), daß es einen solchen Satz von Regel NICHT geben kann! Deshalb nennt man die Vollständigkeit, die wir im obigen Satz formuliert haben, auch relative Vollständigkeit. Die Zusicherungslogik ist nur relativ zur Allgemeingültigkeit der Implikationen vollständig. Mehr können wir aus prinzipiellen Gründen nicht erreichen. In der so harmlos erscheinenden Abschwächungsregel steckt also eine enorme „Power“. Über diese Problematik könnte man aber eine eigenständige Vorlesung halten, so daß wir uns hier mit diesen Andeutungen begnügen müssen.

5 Zusammenfassung

In diesem Kapitel haben wir eine ganz andere Art der Semantik kennen gelernt. Die Semantik einer Anweisung ist nicht unmittelbar durch ihr Verhalten definiert, sondern durch die Eigenschaften die für die Anweisung gelten. Tatsächlich ist es Ansichtssache, ob es ich bei der axiomatischen Semantik um eine Semantik handelt oder einen (auf der operationalen oder mathematischen Semantik aufbauende) Beweistechnik. In jedem Falle stellt die axiomatische Semantik den Zusammenhang zur Programmverifikation her. Der Nachweis der Korrektheit und Vollständigkeit der Verifikationsregeln entspricht dann gerade dem Nachweis der Äquivalenz der axiomatischen Semantik zur operationalen bzw. zur mathematischen Semantik.
Aus der Sicht der Programmverifikation sind die Regeln der axiomatischen Semantik allerdings erst der Anfang. Auf die eigentlichen Aspekte der Programmverifikation konnten wir hier leider nicht eingehen. Die Regeln der axiomatischen Semantik bilden jedoch den Kern fast aller Verifikationsansätze.

Kapitel 8

Zusammenfassung

In dieser Vorlesung haben wir uns mit den Techniken zur Definition von Semantiken und zur Argumentation über ihre Eigenschaften, insbesondere zum Nachweis der Äquivalenz zu anderen Semantiken beschäftigt. Eine zentrale Rolle spielten dabei induktive Definitionen und induktive Beweise und die semantischen Bereiche und Fixpunkte. Im Laufe der Vorlesung sollte deutlich geworden sein, daß induktive Definitionen und Fixpunkte weniger weit auseinander liegen als man zunächst erwarten würde. Eigentlich sind es nur zwei verschiedene Sichtweisen von ein und demselben Sachverhalt.

Obwohl also die mathematische Semantik und die operationale Semantik – wenn man genau hinguckt – weniger unterschiedlich sind als erwartet, gibt es doch einen wichtigen Unterschied in der Formulierung. Denn wir haben gesehen, daß die operationale Semantik nicht kompositional definiert ist, während die mathematische Semantik kompositional definiert ist. Genau um die Kompositionalität zu erreichen, haben wir bei der Definition der mathematischen Semantik Fixpunkte explizit benutzt und später die Fixpunkttheorie ganz allgemein eingeführt. Dabei sind Fixpunkte nichts Esotherisches. Sie ergeben sich ganz natürlich aus der Selbstbezüglichkeit der zugrundeliegenden Konzepte. Die Fixpunkttheorie erlaubt es uns, diese Selbstbezüglichkeit mathematisch sauber und präzise aufzulösen, und damit zu einer kompositionalen Semantik zu gelangen. Wie wir gesehen haben wird die Fixpunkttheorie implizit (in Form der induktiven Definiton) zwar auch bei der Definition der operationalen Semantik eingesetzt. Allerdings wird dort die Selbstbezüglichkeit mehr unter den Teppich gekehrt als wirklich gelöst. Dementsprechend ist die operationale Semantik nicht kompositional.

Ein weiteres Anliegen der Vorlesung ware ganz allgemein die mathematische Formulierung von von Konzepten und der Beweis von Aussagen über diese Konzepte. Im Rahmen der Vorlesung und der Übung haben wir verschiedene Beweistechniken eingeübt, die auch in ganz anderen Bereichen der Informatik genutzt werden können. Denn wer genau hinsieht, wird feststellen, daß es in der Informatik nur so von induktiven Definitionen und Beweisen und auf der Rückseite der Medaille nur so von Fixpunkten wimmelt – wenn auch oft nur unter der Oberfläche.

Teil C

Literatur und Index

Literaturverzeichnis

[1] Apt, K. R. und E.-R. Olderog: *Programmverifikation*. Springer-Lehrbuch. Springer-Verlag, 1994.

[2] Best, Eike: *Semantik*. Vieweg-Verlag, 1995. (document)

[3] Beutelspacher, Albrecht: *"Das ist o.B.d.A. trivial"*. vieweg Mathematik fÃ¼r Studienanfänger. Vieweg, 1991. 3

[4] *Duden: Rechtschreibung der deutschen Sprache*. Dudenverlag, 1996. 1

[5] Fehr, Elfriede: *Semantik von Programmiersprachen*. Studienreihe Informatik. Springer-Verlag, 1989. 4

[6] Friedrichsdorf, Ulf und Alexander Prestel: *Mengenlehre für den Mathematiker*. vieweg studium: Grundkurs Mathematik. Vieweg, 1985. 1

[7] Hoare, C.A.R.: *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12(10):576–583, Oktober 1969. 3

[8] Huth, Michael und Mark Ryan: *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[9] Milner, Robin: *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. 5

[10] Plotkin, Gordon D.: *A Structural Approach to Operational Semantics*. DAIMI FN-19, Computer Science Department, Aarhus University, September 1981. 5

[11] Winskel, Glynn: *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993. (document), 4, 4

Index