Logik, Semantik und Verifikation

Notizen zur Vorlesung
im Sommersemester 2000

Gert Smolka
Fachrichtung Informatik
Universität des Saarlandes

12. Juli 2000

Inhaltsverzeichnis

1 Einleitung **7**

2 Rekursion und Induktion **9**

2.1 Einige mathematische Begriffe 9

2.2 Rekursive Definition mit Inferenzregeln 13

2.3 Beweis durch Regelinduktion 14

2.4 Rekursive Definition der natürlichen Zahlen 18

3 Syntax und Semantik **19**

3.1 Syntax 19

3.2 Strukturelle Rekursion 22

3.3 Strukturelle Induktion 24

3.4 Operationale Semantik 25

3.5 Denotationale Semantik 27

3.6 Übereinstimmung von operationaler und denotationaler Semantik 28

3.7 Diskussion 29

4 Rekursion und Fixpunktoperatoren **33**

4.1 FP-Format und R-Strukturen 34

4.2 Ein Fixpunktoperator für Standard ML 35

4.3 Vollständige partielle Ordnungen 37

4.4 Fallstudie: Rekursive Definition von Mengen 39

4.5 Fallstudie: Rekursive Definition von partiellen Funktionen 41

4.6 Zusammenfassung 43

5 Programme mit Schleifen **45**

5.1 Syntax und operationale Semantik von IMP 45

5.2 Denotationale Semantik von Kommandos mit Standard ML 47

5.3 Denotationale Semantik von Kommandos mathematisch 49

5.4 Übereinstimmung der operationalen und denotationalen Semantik 52

6 Formeln mit Quantoren **57**

6.1 Definition von ASSN 57

6.2 Abkürzungen und Expressivität 59

6.3 Notationen und Sprechweisen 60

6.4 Bindungsstruktur 61

6.5 Substitution 63

7 Programmverifikation **69**

7.1 Spezifikationen und Korrektheit 69

7.2 Hoare-Regeln 72

7.3 Verifikationsbedingungen 73

7.4 VC-Methode 75

7.5 Vollständigkeit der VC-Methode 76

7.6 Schwächste Vorbedingungen 78

7.7 Konstruktion eines Schleifenkombinators 81

7.8 Zusammenfassung 85

8 Berechenbarkeit **87**

8.1 Gödelisierung 87

8.2 Berechenbare Funktionen 88

8.3 Abzählbare Mengen 90
8.4 Entscheidbare und testbare Mengen 91
8.5 Unentscheidbarkeit des Halteproblems 93
8.6 Gödels Unvollständigkeitssatz 94

9 Aussagenlogik **97**
9.1 Zweiwertige Boolesche Algebra 97
9.2 Eine Denksportaufgabe 100
9.3 Die logische Sprache AL 100
9.4 Boolesche Operationen als Mengenoperationen 103
9.5 Klauseln 104
9.6 Disjunktive Vereinfachung 107
9.7 Tableau-Notation 109
9.8 Konjunktive Vereinfachung 111
9.9 Sequenten 112
9.10 Resolution 115
9.11 Primdarstellungen 117
9.12 Kompaktheit 121

10 Prädikatenlogik **125**
10.1 Syntax 125
10.2 Semantik 127
10.3 Variablenfreie Formeln 132
10.4 Aussagenlogisch gültige Formeln 134
10.5 Berechnung von Skolemformen 135
10.6 Quantorenfreie Klauseldarstellung 137
10.7 Der Expansionssatz 140
10.8 Unifikation 143

10.9 Prädikatenlogische Resolution 147
10.10 Allgemeingültigkeit ist unentscheidbar 150
10.11 Theorien und Axiomatisierung 151

11 Rückblick 155

Kapitel 1

Einleitung

Die Vorlesung behandelt logische Methoden, wie sie in vielen Bereichen der Informatik angewendet werden. Dazu zählen Datenbanken, Künstliche Intelligenz, Programmiersprachen und Verifikation von Software und Harrdware. Die Vorlesung behandelt haupsächlich logische Methoden, Anwendungen werden nur im Bereich Programmierung gezeigt.

Logische Methoden sind der Gegenstand einer wissenschaftlichen Disziplin, die man Logik nennt. Man unterscheidet zwischen philosophischer und mathematischer Logik. Philosophische Logik wurde bereits im Altertum betrieben. Hier interessieren wir uns nur für mathematische Logik, die als Teilgebiet der Mathematik ab der zweiten Hälfte des neunzehnten Jahrhunderts enstanden ist (Freges Begriffschrift). Wesentliche Beiträge wurden in der ersten Hälfte des zwanzigsten Jahrhunderts geleistet. Mittlerweile ist die Informatik der wichtigste Anwender von logischen Methoden. Entsprechend tragen Informatiker heute wesentlich zur Weiterentwicklung logischer Methoden bei.

In der Logik betrachtet man sogenannten Logiken. Genau wie bei Programmiersprachen unterscheidet man bei Logiken zwischen Syntax und Semantik. Die Syntax einer Logik stellt sogenannte Formeln zur Verfügung, mit denen Eigenschaften mathematischer Objekte beschrieben werden können. Wir werden idealisierte Programmiersprachen betrachten, deren Programme Funktionen im mathematischen Sinne berechnen. Diese Programmiersprachen werden wir als Logiken auffassen, deren Formeln Funktionen beschreiben.

Die Vorlesung gehört zur Theoretischen Informatik. Das bedeutet, dass wir mit mathematischen Methoden idealisierte Modelle von Phänomenen untersuchen, die in der Informatik von Bedeutung sind.

Einige grundlegende Konzepte wie abstrakte Syntax und die rekursive Definition von Mengen und Funktionen sind in der Programmiersprache Standard ML direkt verfügbar. Wann immer möglich, werden wir neue theoretische Konzepte in Standard ML realisieren. Standard ML gibt uns die faszinierende Möglichkeit, mathematische Konstruktionen durch Programme zu realisieren. Mathematische Konstruktionen sind intelektuelle Artefakte. Machen Sie sich klar, dass Programme immer Realisierung von intelektuellen Artefakten sind.

Kapitel 2

Rekursion und Induktion

Wir behandeln rekursive Definitionen und Induktionsbeweise. Wir zeigen, wie man Mengen rekursiv durch Inferenzregeln definiert und wie man Eigenschaften solcher Mengen mit Regelinduktion beweist.

Diese Techniken sind für die Vorlesung grundlegend. Wir werden regelmäßig komplexe Mengen rekursiv definieren und Eigenschaften dieser Mengen durch Induktion beweisen.

Wir setzen voraus, dass Sie Übung mit Induktionsbeweisen für natürliche Zahlen haben. Mit der rekursiven Definition von Mengen sollten Sie ebenfalls vertraut sein (aus der Vorlesung Programmierung).

Lesematerial

Skript zur Programmierung WS 99/00, Kapitel 2 und 5.1.

Winskel, Kapitel 1–4.

2.1 Einige mathematische Begriffe

Wir werden die mathematische Begriffe aus [Programmierung, Kapitel 2] und [Winskel, Kapitel 1] benutzen. Im Folgenden wiederholen wir einige besonders wichtige Begriffe und vereinbaren einige zusätzliche Sprechweisen.

Tupel

Zu $n \geq 0$ mathematischen Objekten existiert immer genau ein *n-stelliges* Tupel

$$\langle x_1, \ldots, x_n \rangle$$

Das i-te Teilobjekt x_i heißt die *i-te Komponente* des Tupels. Tupel, deren Komponenten mathematische Objekte sind, sind wieder mathematische Objekte. Es gibt genau ein nullstelliges Tupel $\langle\rangle$, das man auch als *leeres Tupel* bezeichnet. Zweistellige Tupel bezeichnet man auch als *Paare*. Für dreistellige Tupel verwendet man auch die Bezeichnung *Tripel*. Tupel mit keiner oder mit mindestens zwei Komponenten schreibt man oft mit runden Klammern:

$$(x_1, \ldots, x_n)$$

Produkte

Das *Produkt* von $n \geq 2$ Mengen $X_1, \ldots, X_n$ ist die Menge

$$X_1 \times \cdots \times X_n = \{ \langle x_1, \ldots, x_n \rangle \mid x_1 \in X_1, \ldots, x_n \in X_n \}$$

aller n-stelligen Tupel, deren i-te Komponente jeweils aus X_i ist.

Summen

Das *Summe* von $n \geq 2$ Mengen $X_1, \ldots, X_n$ ist die Menge

$$X_1 \uplus \cdots \uplus X_n = \{ \langle i, x \rangle \mid i \in \{1, \ldots, n\} \ \wedge \ x \in X_i \}$$

Die Elemente einer Summe sind also Paare $\langle i, x \rangle$, die aus einer sogenannten Variantennummer i und einem Element x aus dem entsprechenden *Summanden X_i* bestehen. Statt von Summen spricht man auch von *disjunkten Vereinigungen*.

Produkte und Summen in Standard ML

In Standard ML kann man mit Typen Produkte und Summen beschreiben. Beispielsweise können wir die Menge

$$\mathbb{Z} \uplus \mathbb{Z} \uplus (\mathbb{Z} * \mathbb{Z})$$

durch den Typ

```
datatype t = A of int | B of int | C of int * int
```

beschreiben. Produkte sind direkt ausdrückbar. Summen müssen mithilfe von Konstruktortypdeklarationen beschrieben werden.

Potenzmenge

Die *Potenzmenge*

$$\mathcal{P}(X) = \{ Y \mid Y \subseteq X \}$$

einer Menge X ist die Menge aller Teilmengen von X. Manchmal benötigen wir auch die Menge aller endlichen Teilmengen von X:

$$\mathcal{P}_{\mathit{fin}}(X) = \{\, Y \mid Y \subseteq X \ \wedge \ Y \text{ endlich} \,\}$$

Relationen

Eine *n-stellige Relation* ist eine Menge, deren Elemente n-stellige Tupel sind. Eine *binäre Relation* ist eine zweistellige Relation.

Sei r eine binäre Relation. Der *Definitionsbereich* (englisch domain) von r ist die Menge

$$\mathit{Dom}\ r = \{\, x \mid \exists y \colon (x, y) \in r \,\}$$

Der *Wertebereich* (englisch range) von r ist die Menge

$$\mathit{Ran}\ f = \{\, y \mid \exists x \colon (x, y) \in r \,\}$$

Funktionen

Eine *Funktion* f ist eine binäre Relation mit der Eigenschaft, dass zu jedem $x \in \mathit{Dom}\ f$ höchstens ein $y \in \mathit{Ran}\ f$ existiert mit $(x, y) \in f$. Statt $(x, y) \in f$ schreibt man meistens $f(x) = y$.

Seien X und Y Mengen. Wir werden die folgenden Notationen benutzen:

$$\begin{aligned}
X \rightharpoonup Y &= \{\, f \mid f \text{ Funktion mit } \mathit{Dom}\ f \subseteq X \text{ und } \mathit{Ran}\ f \subseteq Y \,\} \\
X \to Y &= \{\, f \in X \rightharpoonup Y \mid \mathit{Dom}\ f = X \,\} \\
X \overset{\mathit{fin}}{\rightharpoonup} Y &= \{\, f \in X \rightharpoonup Y \mid f \text{ endliche Menge} \,\}
\end{aligned}$$

Die Elemente von $X \rightharpoonup Y$ bezeichnet wir als *partielle Funktionen* von X nach Y. Die Elemente von $X \to Y$ bezeichnet wir als *totale Funktionen* von X nach Y. Die Elemente von $X \overset{\mathit{fin}}{\rightharpoonup} Y$ bezeichnet wir als *endliche Funktionen* von X nach Y.

In Standard ML entspricht der Typ $t_1 \to t_2$ der Menge $t_1 \rightharpoonup t_2$ der partiellen Funktionen von t_1 nach t_2.

Machen Sie sich klar, dass eine unendliche Funktion in keiner Weise einen Algorithmus zu ihrer Berechnung vorgibt. Wenn Sie dagegen in Standard ML eine Funktion durch eine Prozedur realisieren, müssen Sie stets einen Algorithmus zur Berechnung der Funktion angeben.

Darstellung von endlichen Funktionen

Endliche Funktionen

$$\{(x_1, y_1), \ldots, (x_n, y_n)\}$$

schreiben wir oft als

$$\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$$

Endliche Funktionen kann man sich auch anschaulich als Tabellen vorstellen:

x_1	y_1
x_2	y_2
...	...
x_n	y_n

Lambda-Notation

Mit Hilfe der sogenannten Lambda-Notation kann man Funktionen oft bequem definieren:

$$\begin{aligned} \lambda\, n \in \mathbb{N} \,.\, n \quad &= \{ (n, n) \mid n \in \mathbb{N} \} \\ \lambda\, x \in \mathbb{Z} \,.\, x^2 \quad &= \{ (x, y) \mid x \in \mathbb{Z} \wedge y = x^2 \} \end{aligned}$$

In jeder Zeile wird links und rechts dieselbe Funktion beschrieben. Dabei verwendet die linke Spalte Lambda-Notation und die rechte Spalte die übliche Notation für Mengen.

Adjunktion von Funktionen

Die *Adjunktion* von zwei Funktionen f und g ist wie folgt definiert:

$$f + g \quad = \quad \lambda\, x \in \mathit{Dom}\ f \cup \mathit{Dom}\ g \,.\ \text{if } x \in \mathit{Dom}\ g \text{ then } g(x) \text{ else } f(x)$$

Statt

$$f + \{x \mapsto y\}$$

schreibt man auch

$$f[y/x]$$

(lies „f mit y für x“).

Klammersparende Notation

Oft ist es bequem, bei Applikationen Klammern einzusparen:

$$
\begin{aligned}
f\ x &\quad \text{statt} \quad f(x) \\
f\ x\ y &\quad \text{statt} \quad (f\ x)(y)
\end{aligned}
$$

Diese Konvention ist Ihnen bereits aus Standard ML vertraut. Auch bei der Notation $X \to Y$ (alle totale Funktionen von X nach Y) kann man Klammern einsparen:

$$X \to Y \to Z \quad \text{statt} \quad X \to (Y \to Z)$$

2.2 Rekursive Definition mit Inferenzregeln

Die rekursive Definition von Mengen mit Inferenzregeln wird in [Programmierung, Kapitel 2] erklärt. Statt von rekursiver Definition spricht man auch von *induktiver Definition*.

Als Beispiel definieren wir die Menge P:

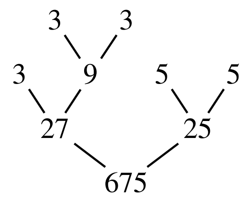
$$\frac{}{3 \in P} \qquad \frac{}{5 \in P} \qquad \frac{x \in P \quad y \in P \quad z = x \cdot y}{z \in P}$$

Diese Regeln definieren P als die kleinste Menge, die 3 und 5 enthält und unter binärer Produktbildung abgeschlossen ist.

Rekursive Definitionen sind konstruktiv in dem Sinne, dass jedes Element der definierten Menge durch *endlich viele Anwendungen* von definierenden Inferenzregeln abgeleitet werden kann. Diese Idee wird durch den Begriff der *Ableitung* expliziert. Die folgende Ableitung zeigt, dass $675 \in P$:

(1)	$3 \in P$	mit Regel 1
(2)	$5 \in P$	mit Regel 2
(3)	$9 \in P$	mit Regel 3, (1), (1) und $9 = 3 \cdot 3$
(4)	$27 \in P$	mit Regel 3, (1), (3) und $27 = 3 \cdot 9$
(5)	$25 \in P$	mit Regel 3, (2), (2) und $25 = 5 \cdot 5$
(6)	$675 \in P$	mit Regel 3, (4), (5) und $675 = 27 \cdot 25$

Die Essenz dieser Ableitung kann durch einen Ableitungsbaum übersichtlich dargestellt werden:



Oft ist es bequem, Mengen durch rekursive Gleichungen zu definieren. Beispielsweise kann man die obige Menge P auch durch die folgende Gleichung definieren:

$$P = \{3, 5\} \cup \{\, x \cdot y \mid x \in P \wedge y \in P \,\}$$

Diese Gleichung ist als eine Kurzschreibweise für die obigen Inferenzregeln zu verstehen.

2.3 Beweis durch Regelinduktion

Regelinduktion ist eine Beweistechnik für Aussagen der Form

$$\forall x \in X : A(x)$$

bei denen X eine durch Regeln rekursiv definierte Menge ist. Dabei soll $A(x)$ eine Aussage für $x \in X$ sein. Für die oben definierte Menge P können wir beispielsweise die folgen Aussage $A(x)$ angeben:

$$\exists m \in \mathbb{N}\ \exists n \in \mathbb{N} :\ x = 3^m \cdot 5^n \wedge\ m + n > 0$$

Sei X eine Menge, die durch die Regeln $R_1, \ldots, R_n$ definiert ist. Ein Beweis durch Regelinduktion für

$$\forall x \in X : A(x)$$

besteht aus n getrennten Teilbeweisen für jede der Regeln $R_1, \ldots, R_n$. Sei R die Regel

$$\frac{A_1 \quad \ldots \quad A_n}{x \in X}$$

Dann muss der Teilbeweis für die Regel R die Gültigkeit der Aussage

$$\forall z_1 \ldots \forall z_m :\quad A_1 \wedge \cdots \wedge A_n \wedge A(x_1) \wedge \cdots \wedge A(x_k) \quad\Rightarrow\quad A(x)$$

zeigen. Dabei sind

$$z_1, \dots, z_m$$

die Variablen, die in der Regel frei vorkommen und

$$x_1, \dots, x_k$$

die Variablen, für die die Regel eine Prämisse der Form $x_i \in X$ hat. Beispielsweise bekommen wir für die dritte Regel

$$\frac{x \in P \quad y \in P \quad z = x \cdot y}{z \in P}$$

der Definition von P die *Beweisverpflichtung*

$$\forall x\ \forall y\ \forall z\colon\ x \in P \wedge y \in P \wedge\ z = x \cdot y \wedge A(x) \wedge A(y) \ \Rightarrow\ A(z)$$

Die Beweisverpflichtungen für die Regel sind also allquantifizierte Implikationen der Form

$$\forall z_1 \dots \forall z_m\colon\quad A_1 \wedge \cdots \wedge A_n \wedge A(x_1) \wedge \cdots \wedge A(x_k) \quad\Rightarrow\quad A(x)$$

Diese beweist man, indem man

$$A_1 \wedge \cdots \wedge A_n \wedge A(x_1) \wedge \cdots \wedge A(x_k)$$

als gültig annimmt und darauf aufbauend zeigt, dass $A(x)$ gültig ist. Die Annahmen

$$A(x_1), \dots, A(x_k)$$

heißen *Induktionsannahmen*. Beim Aufschreiben eines Induktionsbeweises verzichtet man der Bequemlichkeit und Übersichtlichkeit halber auf die explizite Angabe der Induktionsannahmen. Stattdessen schreibt man im Beweis von $A(x)$ bei jeder Benutzung einer Induktionsannahme „nach Induktionsannahme“.

Ein Beispiel

Induktionsbeweise schreibt man nach einem gewissen Schema auf. Wir zeigen das mit einem Beispiel.

Proposition 2.3.1 *Sei die Menge P durch die Regeln*

$$\frac{}{3 \in P} \qquad\qquad \frac{}{5 \in P} \qquad\qquad \frac{x \in P \quad y \in P \quad z = x \cdot y}{z \in P}$$

definiert. Dann gilt

$$\forall x \in P\ \exists m \in \mathbb{N}\ \exists n \in \mathbb{N}\colon\ x = 3^m \cdot 5^n \wedge\ m + n > 0$$

Beweis Durch Induktion über die Definition von M.

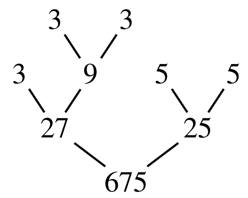
1. $3 = 3^1 \cdot 5^0$.
2. $5 = 3^0 \cdot 5^1$.
3. Seien $x, y \in P$ und $z = x \cdot y$. Nach Induktionsannahme gibt es $m_1, n_1, m_2, n_2 \in \mathbb{N}$ mit $x = 3^{m_1} \cdot 5^{n_1}$, $m_1 + n_1 > 0$, $y = 3^{m_2} \cdot 5^{n_2}$ und $m_2 + n_2 > 0$. Also

$$z = x \cdot y = 3^{m_1} \cdot 5^{n_1} \cdot 3^{m_2} \cdot 5^{n_2} = 3^{m_1+m_2} \cdot 5^{n_1+n_2}$$

und $m_1 + m_2 + n_1 + n_2 > 0$. □

Korrektheit der Regelinduktion

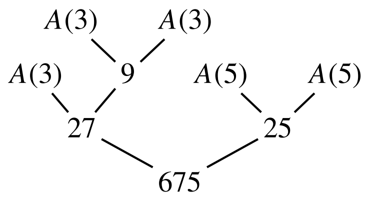
Wir überlegen uns jetzt, warum Regelinduktion eine korrekte Beweistechnik ist. Dazu betrachten wir nochmal den Ableitungsbaum



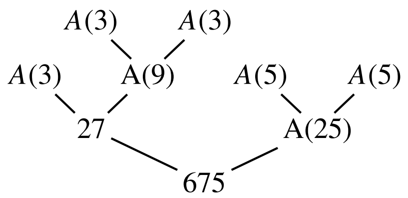
der zeigt, warum $675 \in P$ gilt. Wir nehmen an, dass wir durch Regelinduktion

$$\forall x \in P: \ A(x)$$

gezeigt haben. Die Beweise für die erste und zweite Regel zeigen, dass A für 3 und 5 gilt. Dieses Wissen markieren wir im Ableitungsbaum wie folgt:



Der Beweis für die dritte Regel liefert uns jetzt, dass A für 9 und 25 gilt:



Jetzt folgt mit dem Beweis für die dritte Regel, dass A für 27 gilt:

$$\begin{array}{ccccccc}
 & A(3) & & A(3) & & & \\
A(3) & & \mathrm{A}(9) & & A(5) & & A(5) \\
 & \mathrm{A}(27) & & & & \mathrm{A}(25) & \\
 & & & 675 & & &
\end{array}$$

Wieder mit dem Beweis für die dritte Regel bekommen wir schließlich

$$\begin{array}{ccccccc}
 & A(3) & & A(3) & & & \\
A(3) & & \mathrm{A}(9) & & A(5) & & A(5) \\
 & \mathrm{A}(27) & & & & \mathrm{A}(25) & \\
 & & & A(675) & & &
\end{array}$$

Damit ist klar, dass wir in jedem Ableitungsbaum die Gültigkeit von A von den Blättern bis zur Wurzel propagieren können. Da P genau die Zahlen enthält, die man als Wurzel eines Ableitungsbaum erhalten kann, gilt A offensichtlich für alle Elemente von P.

Der mathematisch geübte Leser wird erkennen, dass wir die Korrektheit der Regelinduktion durch Induktion über die Tiefe der Ableitungsbäume gezeigt haben. Die Korrektheit der Regelinduktion folgt also aus der Korrektheit der Induktion für natürliche Zahlen.

Induktion über natürliche Zahlen

Als Nächstes wollen wir zeigen, dass

$$P = \{\, 3^n \cdot 5^m \mid n, m \in \mathbb{N} \wedge n + m > 0 \,\}$$

gilt. Die Richtung „$\subseteq$“ wird durch die obige Proposition gezeigt. Um die andere Richtung zu zeigen benötigen wir wieder Induktion, diesmal allerdings über $\mathbb{N}^+$ (die postiven natürlichen Zahlen). Der Beweis ist mühsam, da wir drei Fälle betrachten müssen. Zunächst zeigen wir durch Induktion über n, dass

$$\forall n \in \mathbb{N}^+ :\; 3^n \in P$$

gilt. Dann zeigen wir auf analoge Weise, dass

$$\forall n \in \mathbb{N}^+ :\; 5^n \in P$$

gilt. Daraus folgt

$$\forall m \in \mathbb{N}^+ \; \forall n \in \mathbb{N}^+ :\; 3^m \cdot 5^n \in P$$

mit der dritten definierenden Regel für P. Zusammengenommen liefern die drei Teile

$$\{ 3^n \cdot 5^m \mid n, m \in \mathbb{N} \wedge n + m > 0 \} \subseteq P$$

Wir zeigen nun eine der Hilfbehauptungen.

Proposition 2.3.2 $\forall n \in \mathbb{N}^+ : \; 3^n \in P$

Beweis Durch Induktion über n.

1. Sei $n = 1$. Dann $3^n = 3 \in P$ wegen der ersten definierenden Regel für P.
2. Sei $n > 1$. Nach Induktionsannahme gilt $3^{n-1} \in P$. Da $3 \in P$ wegen Regel 1, folgt mit Regel 3 $3^n = 3 \cdot 3^{n-1} \in P$. □

2.4 Rekursive Definition der natürlichen Zahlen

Wir definieren eine Menge N wie folgt:

$$\frac{}{\emptyset \in N} \qquad \frac{x \in N}{\{x\} \in N}$$

Die Elemente von N entsprechen offensichtlich genau den natürlichen Zahlen:

1	$\emptyset$
2	$\{\emptyset\}$
3	$\{\{\emptyset\}\}$
4	$\{\{\{\emptyset\}\}\}$
...	

Offensichtlich kann man auf der Menge N mit Rekursion Funktionen definieren, die den Operation auf natürlichen Zahlen entsprechen. Diese Idee lässt sich am einfachsten in Standard ML demonstrieren:

```
datatype nat = N | S of nat

fun plus (N, y) = y
  | plus (S x, y) = S(plus(x,y))

plus(S(S N), S(S(S( N))))
S(S(S(S(S N)))) : nat
```

Kapitel 3

Syntax und Semantik

Wir zeigen am Beispiel von arithmetischen Ausdrücken, wie man eine einfache logische Sprache AA definiert.

Bei logischen Sprachen unterscheidet man zwischen Syntax und Semantik. Die *Syntax* einer Sprache definiert die *syntaktischen Objekte* der Sprache. Die *Semantik* einer Sprache definiert die *semantischen Objekte* der Sprache und eine oder mehrere *semantische Relationen*. Die syntaktischen Objekte einer Sprache sind Beschreibungen für die semantischen Objekte einer Sprache. Die semantischen Relationen stellen den Zusammenhang zwischen den syntaktischen und semantischen Objekten einer Sprache her.

Die Syntax einer Sprache definiert man durch eine *abstrakte Grammatik*. Für die Definition der semantischen Relationen gibt es zwei Techniken: die *operationale* und die *denotationale*.

Im Zusammenhang mit Syntax und denotationaler Semantik lernen wir *strukturelle Rekursion* und *strukturelle Induktion* kennen. Dabei handelt es sich um wichtige Spezialfälle von Regel-Rekursion und Regel-Induktion, die wir in der letzten Vorlesung kennengelernt haben.

Lesematerial

[Programmierung, Kapitel 11] und [Winskel, Kapitel 2–4].

3.1 Syntax

Wir betrachten eine *Beispielssprache AA*, deren syntaktische Objekte ganzzahlige arithmetische Ausdrücke darstellen sollen. Abbildung 3.1 zeigt eine *abstrakte*

$$
\begin{array}{lllll}
c & \in & \mathit{Kon} = \mathbb{Z} & & \textit{Konstante} \\
x & \in & \mathit{Var} & & \textit{Variable} \\
a & \in & \mathit{Aus} = & & \textit{Ausdruck} \\
 & & \quad c & & \quad \textit{Konstante} \\
 & & \mid\ x & & \quad \textit{Variable} \\
 & & \mid\ -a & & \quad \textit{Negation} \\
 & & \mid\ a_1 + a_2 & & \quad \textit{Summe} \\
 & & \mid\ a_1 \times a_2 & & \quad \textit{Produkt}
\end{array}
$$

Abbildung 3.1: Abstrakte Syntax von AA

Grammatik, die die Syntax unserer Beispielssprache AA definiert. Die Abbildung sollte Ihnen eine gute Idee von der Sprache AA geben, obwohl wir noch gar nicht erklärt haben, was eine abstrakte Grammatik sein soll.

AA ist nach Var parametrisiert

Die Syntax und die Semantik von AA sind relativ zu einer vorgegeben Menge *Var* definiert. Wir sagen, dass AA nach *Var* parametrisiert ist.

Die Grammatik vereinbart 3 Mengen von syntaktischen Objekten: Konstanten, Variablen und Ausdrücke. Die Menge der Variablen wird als vorgegeben angenommen. Dagegen werden die Mengen der Konstanten und Ausdrücke definiert. Die Menge der Ausdrücke ist rekursiv definiert und es werden 5 Ausdrucksvarianten vereinbart.

Es stellt sich nun die Frage, durch welche mathematischen Objekte die Ausdrücke von AA repräsentiert werden sollen. Wir wählen dafür Tupel, mit denen wir die Baumstruktur von Ausdrücken direkt wiederspiegeln können:

$$\mathit{Aus} = \mathit{Kon} \uplus \mathit{Var} \uplus \mathit{Aus} \uplus \mathit{Aus} \times \mathit{Aus} \uplus \mathit{Aus} \times \mathit{Aus}$$

Ausdrücke werden also als die Elemente einer Summe repräsentiert, wobei jeder Summand eine der 5 möglichen Ausdrucksvarianten darstellt. Noch expliziter lässt sich die Menge *Aus* durch Inferenzregeln beschreiben:

$$\frac{c \in \mathit{Kon}}{\langle 1, c\rangle} \qquad \frac{x \in \mathit{Var}}{\langle 2, x\rangle} \qquad \frac{a \in \mathit{Aus}}{\langle 3, a\rangle}$$

$$\frac{a_1 \in \mathit{Aus} \quad a_2 \in \mathit{Aus}}{\langle 4, \langle a_1, a_2\rangle\rangle} \qquad \frac{a_1 \in \mathit{Aus} \quad a_2 \in \mathit{Aus}}{\langle 5, \langle a_1, a_2\rangle\rangle}$$

Notation für syntaktische Objekte

Die direkte mathematische Notation für die gerade definierten Ausdrücke von AA ist unhandlich und schwer lesbar. Glücklicherweise legt die Grammatik jedoch auch eine lesbare Notation für die definierten Ausdrücke fest. Beispielsweise kann man für den Ausdruck

$$\langle 4, \langle\langle 1, 3\rangle\langle 4, \langle\langle 1, 5\rangle, \langle 3, \langle 1, 7\rangle\rangle\rangle\rangle\rangle\rangle$$

bequemer und lesbarer

$$3 + (5 + -(7))$$

schreiben. Entsprechend kann man für

$$\langle 4, \langle\langle 2, x_1\rangle\langle 4, \langle\langle 1, c\rangle, \langle 3, \langle 2, x_2\rangle\rangle\rangle\rangle\rangle\rangle$$

auch

$$x_1 + (c + -x_2)$$

schreiben.

Wir legen großen Wert darauf, die Objekte der abstrakten Syntax präzise als mathematische Objekte zu definieren. Dagegen sind bei der Festlegung und Verwendung unserer Notationen vergleichsweise nachlässig und vertrauen auf die mathematische Erfahrung des Lesers. Beispielsweise ist die Notation

$$1 + 2 + 3$$

für Ausdrücke in *Aus* unzulässig, da sie zwei Lesarten hat:

$$(1 + 2) + 3 \quad \text{oder} \quad 1 + (2 + 3)$$

Für die Notation

$$-3$$

gibt es in unserem Kontext drei Lesarten: die Zahl $-3 \in \mathit{Kon}$, der Ausdruck $\langle 1, -3\rangle \in \mathit{Aus}$ (Konstante), oder der Ausdruck $\langle 3, \langle 1, -3\rangle\rangle \in \mathit{Aus}$ (Negation). Generell ist der Gebrauch einer Notationen nur dann zulässig, wenn der Kontext dafür sorgt, dass es nur eine Lesart gibt.

Abstrakte und konkrete Syntax

Programmiersprachen legen für ihre syntaktischen Objekte eine textuelle Notation fest, die in jeder Hinsicht präzise definiert ist. Entsprechend unterscheidet man zwischen abstrakter und konkreter Syntax. Während die *abstrakte Syntax* die eigentlichen syntaktischen Objekte festlegt, definiert die *konkrete Syntax* eine textuelle Darstellung für diese Objekte.

Im Rahmen dieser Vorlesung interessieren wir uns vor allem für die abstrakte Syntax von Sprachen. Bei den Notationen für die Objekte der abstrakten Syntax halten wir uns an mathematische Gepflogenheiten und verzichten auf die präzise Definition einer konkreten Syntax. Wenn wir im Folgenden Syntax sagen, meinen wir stets abstrakte Syntax.

3.2 Strukturelle Rekursion

Wir haben die Menge der Ausdrücke von AA rekursiv definiert:

$$Aus = Kon \uplus Var \uplus Aus \uplus Aus \times Aus \uplus Aus \times Aus$$

Diese rekursive Definition hat eine spezielle Form, die man als strukturelle Rekursion bezeichnet:

1. Die definierte Menge ist eine Summe.
2. Für jede Variante der Summe gibt es genau eine Inferenzregel (siehe Abschnitt 3).

Die Deklaration eines Konstruktortyps in Standard ML entspricht der Definition einer Menge durch strukturelle Rekursion. Beispielsweise führt die Deklaration

```
datatype aus =
    Kon of kon
  | Var of var
  | Neg of aus
  | Add of aus * aus
  | Mul of aus * aus
```

einen Typ `aus` ein, der gerade der Menge *Aus* entspricht. Gegeben

```
type kon = int
```

können wir den Ausdruck

$$2 + -(3 \times 7)$$

in Standard ML durch

```
Add(Kon 2, Neg(Mul(Kon 3, Kon 7)))
```

darstellen. Die Variantennummern einer mathematischen Summe werden in Standard ML durch Konstruktoren mit frei wählbaren Namen ersetzt.

Für die Definition von totalen Funktionen auf Mengen, die durch strukturelle Rekursion definiert sind, gibt es eine spezielle Technik, die ebenfalls als strukturelle Rekursion bezeichnet wird. Als Beispiel betrachten wir die Funktion

$$V \in \mathit{Aus} \to \mathcal{P}(\mathit{Var})$$

die einem Ausdruck die Menge der in ihm vorkommenden Variablen zuordnet. Die Definition von V durch strukturelle Rekursion sieht wie folgt aus:

$$\begin{aligned} V(c) &= \emptyset \\ V(x) &= \{x\} \\ V(-a) &= V(a) \\ V(a_1 + a_2) &= V(a_1) \cup V(a_2) \\ V(a_1 \times a_2) &= V(a_1) \cup V(a_2) \end{aligned}$$

Sei M eine durch strukturelle Rekursion definierte Menge. Um eine Funktion f auf M durch strukturelle Rekursion zu defnieren, geht man wie folgt vor:

1. Definiere f durch eine Gleichungsregel für jede Variante der Summe M.
2. Beschränke rekursive Applikationen von f auf der rechten Seite einer Gleichungsregel auf Argumente, die echte Teilobjekte des Arguments auf der linken Seite sind.

Diese Definitionstechnik garantiert, dass f als totale Funktion auf M definiert wird. Natürlich müssen wir voraussetzen, dass die auf den rechten Seiten verwendeten Hilfsfunktionen die richtigen Eigenschaften haben.

Die ersten zwei Regeln der Definition von V sind in einer Kurzschreibweise aufgeschrieben. Eigentlich müssten wir

$$\begin{aligned} V(\langle 1, c\rangle) &= \emptyset \\ V(\langle 2, x\rangle) &= \{x\} \end{aligned}$$

schreiben, da V auf Ausdrücken definiert ist und Konstanten und Variablen als solche keine Ausdrücke sind. Andererseits ist der Kontext der Definition von V ausreichend, um die fehlenden Variantennummern automatisch zu ergänzen.

In Standard ML können wir eine Prozedur

```
fun var(Kon c)        = empty
  | var(Var x)        = add(empty, x)
  | var(Neg a)        = var a
  | var(Add(a1,a2))   = union(var a1, var a2)
  | var(Mul(a1,a2))   = union(var a1, var a2)
```

schreiben, die die Funktion V berechnet. Dabei setzen wir voraus, dass die Bezeichner

```
val empty : var set
val add   : var set * var -> var set
val union : var set * var set -> var set
```

geeignet gebunden sind.

Selbstverständlich kann man die Funktion V auch durch Inferenzregel definieren:

$$\frac{c \in Kon}{\langle\langle 1, c\rangle, \emptyset\rangle \in V} \qquad \frac{x \in Var}{\langle\langle 2, x\rangle, \{x\}\rangle \in V} \qquad \frac{\langle a, X\rangle \in V}{\langle -a, X\rangle \in V}$$

$$\frac{\langle a_1, X_1\rangle \in V \quad \langle a_2, X_2\rangle \in V \quad X = X_1 \cup X_2}{\langle a_1 + a_2, X\rangle \in V}$$

$$\frac{\langle a_1, X_1\rangle \in V \quad \langle a_2, X_2\rangle \in V \quad X = X_1 \cup X_2}{\langle a_1 \times a_2, X\rangle \in V}$$

Wir können die funktionale Definition von V durch Gleichungsregeln als eine Kurzschreibweise für die relationale Definition durch Inferenzregeln auffassen. Neben besserer Lesbarkeit hat die funktionale Definition mit struktureller Rekursion gegenüber der relationalen Definition den Vorteil, dass sie garantiert, dass V eine totale Funktion auf *Aus* ist.

3.3 Strukturelle Induktion

Um zu zeigen, dass die relationale Definition von V eine Funktion in $Aus \to \mathcal{P}(Var)$ definiert, müssen wir drei Eigenschaften zeigen:

(1) $\forall \langle a, X\rangle \in V\colon\ a \in Aus \ \wedge\ X \in \mathcal{P}(Var)$

(2) $\forall a \in Aus\ \exists X \in \mathcal{P}(Var)\colon\ \langle a, X\rangle \in V$

(3) $\forall a \in Aus\ \forall X_1, X_2 \in \mathcal{P}(Var)\colon\ \langle a, X_1\rangle \in V \wedge \langle a, X_2\rangle \in V \ \Rightarrow\ X_1 = X_2$

Die erste Eigenschaft können wir mit Regelinduktion über die Definition von V zeigen. Die Eigenschaften (2) und (3) können wir durch Induktion über die

Definition von *Aus* zeigen. Da *Aus* durch strukturelle Rekursion definiert ist, können wir den Induktionsbeweis in einer speziellen Form aufschreiben, die als strukturelle Induktion bezeichnet wird. Wir zeigen das am Beispiel der zweiten Eigenschaft.

Proposition 3.3.1 *Sei V durch die obigen Inferenzregeln definiert. Dann gilt:*

$$\forall a \in Aus\ \exists X \in \mathcal{P}(Var)\colon\ \langle a, X\rangle \in V$$

Beweis Durch strukturelle Induktion über $a \in Aus$.

1. Sei $\langle 1, c\rangle \in Aus$. Dann folgt $\langle\langle 1, c\rangle, \emptyset\rangle \in V$ mit der ersten Inferenzregel für V.
2. Sei $\langle 2, x\rangle \in Aus$. Dann folgt $\langle\langle 2, x\rangle, \{x\}\rangle \in V$ mit der zweiten Inferenzregel für V.
3. Sei $-a \in Aus$. Nach Induktionsannahme gibt es ein $X \in \mathcal{P}(Var)$ mit $\langle a, X\rangle \in V$. Also folgt $\langle -a, X\rangle \in V$ mit der dritten Inferenzregel für V.
4. Sei $a_1 + a_2 \in Aus$. Nach Induktionsannahme gibt es $X_1, X_2 \in \mathcal{P}(Var)$ mit $\langle a_1, X_1\rangle \in V$ und $\langle a_2, X_2\rangle \in V$. Also folgt $\langle a_1 + a_2, X_1 \cup X_2\rangle \in V$ mit der vierten Inferenzregel für V.
5. Sei $a_1 \times a_2 \in Aus$. Nach Induktionsannahme gibt es $X_1, X_2 \in \mathcal{P}(Var)$ mit $\langle a_1, X_1\rangle \in V$ und $\langle a_2, X_2\rangle \in V$. Also folgt $\langle a_1 \times a_2, X_1 \cup X_2\rangle \in V$ mit der fünften Inferenzregel für V.

$\square$

Ein Beweis durch strukturelle Induktion über eine durch strukturelle Rekursion definierte Menge M besteht also aus einem Teilbeweis für jede Variante von M. Strukturelle Induktion ist offensichtlich ein Spezialfall von Regelinduktion.

3.4 Operationale Semantik

Wir überlegen uns jetzt, welche semantischen Objekte wir den Ausdrücke von AA zuordnen wollen. Ausdrücken, die keine Variablen enthalten, könnten wir die Zahl zuordnen, die man durch ausrechnen bekommen kann. Für Ausdrücke mit Variablen müssen wir uns etwas mehr einfallen lassen, da wir diese erst ausrechnen können, wenn wir Werte für die Variablen vorgeben.

Zunächst definieren wir eine Menge

$$\sigma \in \Sigma = Var \to \mathbb{Z} \qquad \textit{Belegung}$$

von Funktionen, die wir *Belegungen* nennen. Eine Belegung gibt für jede Variable einen Wert vor.

Als Nächstes definieren wir eine Relation

$$OS = \Sigma \times Aus \times \mathbb{Z}$$

so, dass

$$\langle \sigma, a, c \rangle \in OS$$

genau dann gilt, wenn der Ausdruck a mit der Belegung σ zu der Zahl c auswertet. Bevor wir OS definieren, legen wir noch eine Notation fest:

$$\sigma \vdash a \Rightarrow c \iff \langle \sigma, a, c \rangle \in OS$$

Die Relation OS definieren wir jetzt durch Inferenzregeln:

$$\frac{\sigma \in \Sigma \quad c \in Kon}{\sigma \vdash c \Rightarrow c} \qquad \frac{\sigma \in \Sigma \quad x \in Var \quad c = \sigma(x)}{\sigma \vdash x \Rightarrow c}$$

$$\frac{\sigma \vdash a \Rightarrow c' \quad c = -c'}{\sigma \vdash -a \Rightarrow c}$$

$$\frac{\sigma \vdash a_1 \Rightarrow c_1 \quad \sigma \vdash a_2 \Rightarrow c_2 \quad c = c_1 + c_2}{\sigma \vdash a_1 + a_2 \Rightarrow c}$$

$$\frac{\sigma \vdash a_1 \Rightarrow c_1 \quad \sigma \vdash a_2 \Rightarrow c_2 \quad c = c_1 \cdot c_2}{\sigma \vdash a_1 \times a_2 \Rightarrow c}$$

Die Definition von OS bezeichnet man als eine operationale Semantik für AA. Als semantische Objekte haben wir Zahlen und Belegungen. OS ist eine sogenannte semantische Relation, die die syntaktischen mit den semantischen Objekten in Verbindung bringt.

Wir erwarten, dass die folgende Proposition für OS gilt:

Proposition 3.4.1 *Für jede Belegung $\sigma \in \Sigma$ und jeden Ausdruck $a \in Aus$ gibt es genau eine Zahl $c \in \mathbb{Z}$ mit $\sigma \vdash a \Rightarrow c$.*

Der Beweis dieser Proposition kann durch Induktion über $a \in Aus$ erfolgen. Einen ähnlichen Beweis haben wir im letzten Abschnitt für V gesehen. Wir werden die Proposition jedoch erst später mit einer anderen Technik beweisen.

3.5 Denotationale Semantik

Die obige Propostion garantiert uns, dass wir jedem Ausdruck genau eine Funktion in

$$\Sigma \to \mathbb{Z}$$

zuordnen können. Es gibt also eine Funktion

$$\mathcal{D} \in \mathit{Aus} \to (\Sigma \to \mathbb{Z})$$

die jedem Ausdruck genau ein semantisches Objekt zuordnet. Für $\mathcal{D}$ muss

$$\mathcal{D} = \lambda\, a \in \mathit{Aus} \,\textbf{.}\, \{\, \langle \sigma, c \rangle \mid \langle \sigma, a, c \rangle \in \mathit{OS} \,\}$$

gelten. Es besteht also die Möglichkeit, $\mathcal{D}$ mit Hilfe der operationalen Semantik *OS* zu definieren.

Mathematisch ist es jedoch viel eleganter, $\mathcal{D}$ direkt durch strukturelle Induktion über *Aus* zu definieren und als Semantik für AA zu definieren.

Wir vereinbaren zwei Notationen

$$\begin{aligned} \mathcal{D}[\![a]\!] &= \mathcal{D}(a) \\ \mathcal{D}[\![a]\!]\sigma &= (\mathcal{D}(a))(\sigma) \end{aligned}$$

und definieren damit $\mathcal{D}$ durch strukturelle Induktion über *Aus*:

$$\begin{aligned} \mathcal{D}[\![c]\!] &= \lambda\, \sigma \in \Sigma \,\textbf{.}\, c \\ \mathcal{D}[\![x]\!] &= \lambda\, \sigma \in \Sigma \,\textbf{.}\, \sigma(x) \\ \mathcal{D}[\![-a]\!] &= \lambda\, \sigma \in \Sigma \,\textbf{.}\, -(\mathcal{D}[\![a]\!]\sigma) \\ \mathcal{D}[\![a_1 + a_2]\!] &= \lambda\, \sigma \in \Sigma \,\textbf{.}\, \mathcal{D}[\![a_1]\!]\sigma + \mathcal{D}[\![a_2]\!]\sigma \\ \mathcal{D}[\![a_1 \times a_2]\!] &= \lambda\, \sigma \in \Sigma \,\textbf{.}\, \mathcal{D}[\![a_1]\!]\sigma \cdot \mathcal{D}[\![a_2]\!]\sigma \end{aligned}$$

Da wir $\mathcal{D}$ durch strukturelle Induktion über *Aus* definiert haben, wissen wir sofort das

$$\mathcal{D} \in \mathit{Aus} \to (\Sigma \to \mathbb{Z})$$

gilt.

Wenn wir die Definition von $\mathcal{D}$ in Standard ML schreiben, bekommen wir eine Prozedur, die $\mathcal{D}$ berechnet:

```
fun d (Kon c) s       = c
  | d (Var x) s       = s x
  | d (Neg a) s       = ~(d a s)
  | d (Add(a1,a2)) s = d a1 s + d a2 s
  | d (Mul(a1,a2)) s = d a1 s * d a2 s
```

Das einem Ausdruck $a \in Aus$ durch $\mathcal{D}$ zugeordnete semantische Objekt $\mathcal{D}[\![a]\!]$ bezeichnet man als die *Denotation von* a. Eine durch strukturelle Rekursion definierte Funktion, die so wie $\mathcal{D}$ syntaktische Objekte auf semantische Objekte abbildet, bezeichnet man als *denotationale Semantik*.

Zwei Ausdrücke heißen *äquivalent* genau dann, wenn sie die gleiche Denotation haben.

Machen Sie sich klar, dass für zwei Ausdrücke $a_1, a_2 \in Aus$ stets

$$a_1 + a_2 \neq a_2 + a_1 \iff a_1 \neq a_2$$

gilt. Dagegen sind $a_1 + a_2$ und $a_2 + a_1$ stets äquivalent, da für alle $\sigma \in \Sigma$

$$\mathcal{D}[\![a_1 + a_2]\!]\sigma = \mathcal{D}[\![a_1]\!]\sigma + \mathcal{D}[\![a_2]\!]\sigma = \mathcal{D}[\![a_2]\!]\sigma + \mathcal{D}[\![a_1]\!]\sigma = \mathcal{D}[\![a_2 + a_1]\!]\sigma$$

gilt.

3.6 Übereinstimmung von operationaler und denotationaler Semantik

Proposition 3.6.1 $\forall a \in Aus\ \forall \sigma \in \Sigma\ \forall c \in \mathbb{Z}\colon\ \mathcal{D}[\![a]\!]\sigma = c \ \Rightarrow\ \sigma \vdash a \Rightarrow c.$

Beweis Durch strukturelle Induktion über $a \in Aus$. Sei $\mathcal{D}[\![a]\!]\sigma = c$.

1. Sei $a = c'$. Dann $c = c'$ nach Definition von $\mathcal{D}$. Also $\sigma \vdash a \Rightarrow c$ mit der ersten Regel für *OS*.
2. Sei $a = x$. Dann $c = \sigma(x)$ nach Definition von $\mathcal{D}$. Also $\sigma \vdash x \Rightarrow c$ mit der zweiten Regel für *OS*.
3. Sei $a = -a'$. Dann $c = -\mathcal{D}[\![a']\!]\sigma$ nach Definition von $\mathcal{D}$. Also $\sigma \vdash a' \Rightarrow -c$ nach Induktionsanahme. Also $\sigma \vdash a \Rightarrow c$ mit der dritten Regel für *OS*.
4. Sei $a = a_1 + a_2$. Dann $c = \mathcal{D}[\![a_1]\!]\sigma + \mathcal{D}[\![a_2]\!]\sigma$ nach Definition von $\mathcal{D}$. Sei $c_1 = \mathcal{D}[\![a_1]\!]\sigma$ und $c_2 = \mathcal{D}[\![a_2]\!]\sigma$. Die Induktionsanahme liefert $\sigma \vdash a_1 \Rightarrow c_1$ und $\sigma \vdash a_2 \Rightarrow c_2$. Also $\sigma \vdash a_1 + a_2 \Rightarrow c$ mit der vierten Regel für *OS*.
5. Sei $a = a_1 \times a_2$. Dann folgt $\sigma \vdash a_1 \times a_2 \Rightarrow c$ analog zu (4). $\square$

Proposition 3.6.2 $\forall a \in Aus\, \forall \sigma \in \Sigma\, \forall c \in \mathbb{Z}\colon\ \sigma \vdash a \Rightarrow c \ \Rightarrow\ \mathcal{D}[\![a]\!]\sigma = c$.

Beweis Durch Regelinduktion über die Definition von *OS*.

1. Sei $\sigma \in \Sigma$ und $c \in Kon$. Dann $\mathcal{D}[\![c]\!]\sigma = c$ nach Definition von $\mathcal{D}$.
2. Sei $\sigma \in \Sigma$, $x \in Var$ und $c = \sigma(x)$. Dann $\mathcal{D}[\![x]\!]\sigma = c$ nach Definition von $\mathcal{D}$.
3. Sei $\sigma \vdash a \Rightarrow c'$ und $c = -c'$. Dann $\mathcal{D}[\![a]\!]\sigma = c'$ nach Induktionsanahme. Also $\mathcal{D}[\![-a]\!]\sigma = c$ nach Definition von $\mathcal{D}$.
4. Sei $\sigma \vdash a_1 \Rightarrow c_1$, $\sigma \vdash a_2 \Rightarrow c_2$ und $c = c_1 + c_2$. Dann $\mathcal{D}[\![a_1]\!]\sigma = c_1$ und $\mathcal{D}[\![a_2]\!]\sigma = c_2$ nach Induktionsanahme. Also $\mathcal{D}[\![a_1 + a_2]\!]\sigma = c$ nach Definition von $\mathcal{D}$.
5. Analog zu (4). $\square$

Proposition 3.6.3 $OS = \{\, \langle \sigma, a, c\rangle \mid \mathcal{D}[\![a]\!]\sigma = c \,\}$.

Beweis Die Richtung „$\subseteq$“ folgt aus Proposition 3.6.2. Die andere Richtung folgt aus Proposition 3.6.1. $\square$

Aus Proposition 3.6.3 folgt übrigens sofort Proposition 3.4.1.

Proposition 3.6.4 $\mathcal{D} = \lambda\, a \in Aus\,.\ \{\, \langle \sigma, c\rangle \mid \langle \sigma, a, c\rangle \in OS \,\}$.

Beweis Es genügt zu zeigen: $\forall a \in Aus\colon\ \mathcal{D}[\![a]\!] = \{\, \langle \sigma, c\rangle \mid \langle \sigma, a, c\rangle \in OS \,\}$.

Seien $a \in Aus$, $\sigma \in \Sigma$ und $c \in \mathbb{Z}$ beliebig gewählt. Dann folgt mit Proposition 3.6.3:

$$\langle \sigma, c\rangle \in \mathcal{D}[\![a]\!] \iff \mathcal{D}[\![a]\!]\sigma = c \iff \langle \sigma, a, c\rangle \in OS \qquad \square$$

Proposition 3.6.5 *Seien $a_1, a_2 \in Aus$. Dann gilt:*

$$\mathcal{D}[\![a_1]\!] = \mathcal{D}[\![a_2]\!] \iff \forall \sigma \in \Sigma\, \forall c \in \mathbb{Z}\colon\ (\sigma \vdash a_1 \Rightarrow c \iff \sigma \vdash a_2 \Rightarrow c)$$

3.7 Diskussion

Am Beispiel einer sehr einfachen Sprache AA haben wir grundlegende logische Methoden kennengelernt.

Zunächst haben wir gelernt, wie man Mengen von syntaktische Objekten mit struktureller Rekursion definiert. Zusammengesetzte syntaktische Objekte (zum Beispiel Ausdrücke) werden dabei als Tupel

$$\langle i, o_1, \ldots, o_n\rangle$$

repräsentiert, die aus einer Variantennummer und endlich vielen Teilobjekten $o_1, \ldots, o_n$ ($n \geq 0$) (den sogenannten *Konstituenten*) bestehen. Die für eine Sprache erforderlichen syntaktischen Mengen können zusammen mit passenden Notationen mittels einer abstrakten Grammatiken beschrieben werden.

Wir haben zwei Methoden, operationale und denotationale Semantik, kennengelernt, mit denen man die Semantik von Sprachen beschreiben kann. Die elegantere Methode ist die der denotationalen Semantik, bei der eine Denotationsfunktion den syntaktischen Objekten semantische Objekte zuordnet. Eine denotationale Semantik macht explizit, dass syntaktische Objekte semantische Objekte beschreiben. Die Denotationsfunktion muss dabei mit struktureller Rekursion bezüglich der Rekursionsstruktur der syntaktischen Objekte definiert werden. Diese Forderung garantiert wichtige mathematische Eigenschaften und stellt eine gute Grundlage für Äquivalenzbeweise dar. (Zwei syntaktische Objekte heißen äquivalent, wenn sie das gleiche semantische Objekt beschreiben.)

Die Methode der operationalen Semantik definiert semantische Relationen mit Hilfe von Inferenzregeln. Aufbauend auf den semantischen Relationen kann man auch Denotationsfunktionen definieren. Gegenüber der denotationalen Definitionsmethode hat die operationale Definitionsmethode den Nachteil, dass sie eine weniger geeignete Grundlage für das Beweisen von Eigenschaften der Denotationsfunktionen liefert.

Durch Induktionsbeweise haben wir gezeigt, dass die operationale und die denotationale Semantik für AA übereinstimmen. Das stärkt unser Vertrauen, dass wir bei den operationalen und den denotationalen Definitionen keine Fehler gemacht haben. (Machen Sie sich klar, dass Definitionen falsch sein können, weil sie etwas anderes definieren als die zugrundeliegenden Konzepte.) Die Konstuktion verschiedener Charakterisierungen desselben Konzepts und das Beweisen der Übereinstimmung sind typisches mathematisches Vorgehen. Jede neue Charakterisierung liefert eine neue Sicht auf das zugrundeliegende Konzept und eröffnet neue Beweismöglichkeiten.

Der Vorteil der operationalen Methode ist es, dass sie auch bei komplexen Programmiersprachen relativ leicht anwendbar ist. Beispielsweise wird Standard ML durch eine operationale Semantik definiert, da eine denotationale Semantik zu komplex wäre. Noch schwieriger ist die Situation bei nebenläufigen Sprachen (zum Beispiel Sprachen mit Threads), auf die die denotationale Methode (bisher) nicht anwendbar ist. Das Problem der denotationalen Methode ist also, dass sie nur auf verhältnismäßig einfache Sprachen anwendbar ist.

Wir haben gesehen, dass man die syntaktischen und semantische Konstruktionen für AA direkt mit Standard ML programmieren kann. Syntaktische Mengen lassen sich durch Konstruktortypen beschreiben. Denotationsfunktionen werden durch rekursive Prozeduren beschrieben. Dabei ist wichtig, dass Standard ML Funktionstypen höherer Ordnung wie zum Beispiel

$$Aus \to \Sigma \to \mathbb{Z}$$

direkt realisieren kann.

Kapitel 4

Rekursion und Fixpunktoperatoren

Rekursion ist eine wichtige Definitionstechnik. Im Rahmen dieser Vorlesung haben wir Rekursion benutzt, um die Syntax und die Semantiken der Sprache AA zu definieren.

Aus mathematischer Sicht sind allgemeine rekursive Definitionen problematisch, da nicht immer klar ist, was sie definieren. Wir müssen also zwischen zulässigen und unzulässigen rekursiven Definitionsversuchen unterscheiden.

In Standard ML benutzen wir Rekursion um Prozeduren und Typen zu definieren. Standard ML ist so konstruiert, dass rekursive Definitionen mathematisch immer dann zulässig sind, wenn sie gemäß der statischen Semantik zulässig sind. Damit kann die Zulässigkeit von rekursiven Definitionen in Standard ML automatisch durch ein Programmiersystem geprüft werden.

Wir entwickeln jetzt ein einfaches mathematisches Modell für zulässige rekursive Definitionen. Dieses Modell erklärt rekursive Definitionen durch nicht-rekursive Definitionen, deren Zulässigkeit offensichtlich ist. Viele der rekursiven Definitionen, die wir später in der Vorlesung benötigen, werden wir innerhalb dieses Modells ausführen.

Lesematerial

[Winskel, Kapitel 4.1, 4.4 und 5.4]

4.1 FP-Format und R-Strukturen

Sei X eine Menge und $f \in X \to X$ eine Funktion. Dann kann man versuchen, ein $x \in X$ als die Lösung der rekursiven Gleichung

$$x \stackrel{\text{def}}{=} f(x)$$

zu definieren. Wir sagen, dass ein solcher rekursiver Definitionsversuch im *FP-Format* vorliegt. Die Funktion f nennen wir das *Funktional* des Definitionsversuchs. Natürlich stellt sich sofort die Frage, ob es ein x mit

$$x = f(x)$$

gibt, und ob es mehrere solche x gibt. Eine rekursive Definition im FP-Format ist offensichtlich nur dann *zulässig*, wenn die eindeutige Existenz eines solchen x gesichert ist. Jedes $x \in X$ mit $x = f(x)$ nennt man einen *Fixpunkt* von f. Mit dieser Sprechweise ist eine rekursive Definition im FP-Format genau dann zulässig, wenn f genau einen Fixpunkt hat.

Das gerade vorgestellte Definitionsschema ist nicht sehr nützlich, da die in der Praxis auftretenden Funktionen f in der Regel mehr als einen Fixpunkt haben. Dieses Problem können wir aber beseitigen, indem wir einen Mechanismus hinzufügen, der für f genau einen Fixpunkt auswählt.

Definition 4.1.1 *Ein* R-Struktur *ist ein Tripel* $\langle X, F, \mathit{fix} \rangle$ *wie folgt:*

- *X ist eine Menge.*
- $F \subseteq X \to X$.
- $\mathit{fix} \in F \to X$ *so dass* $\forall f \in F\colon\ \mathit{fix}(f) = f(\mathit{fix}(f))$.

Die Funktion fix bezeichnen wir als den Fixpunktoperator der R-Struktur.

Sei $\langle X, F, \mathit{fix} \rangle$ eine R-Struktur und $f \in F$. Dann legen wir fest, dass eine rekursive Definition

$$x \stackrel{\text{def}}{=} f(x)$$

im FP-Format der nicht-rekursiven Definition

$$x \stackrel{\text{def}}{=} \mathit{fix}(f)$$

entspricht. Eine rekursive Definition im FP-Format wird also mithilfe einer passenden R-Struktur als Notation für eine nicht-rekursive Definition mit dem Fixpunktoperator der R-Struktur interpretiert. Wir werden sehen, dass dieses einfache Modell eine Vielzahl von rekursiven Definitionen erklären kann.

4.2 Ein Fixpunktoperator für Standard ML

Wir zeigen jetzt, dass wir rekursive Prozedurdeklarationen in Standard ML so umschreiben können, dass sie (fast) im FP-Format vorliegen. Weiter zeigen wir, dass wir in Standard ML einen Fixpunktoperator definieren können, mit dem wir rekursive Prozedurdeklarationen im FP-Format in nicht-rekursive Deklarationen umschreiben können.

Wir betrachten die rekursive Deklaration

```
fun fac n = if n<2 then 1 else n * fac (n-1)
fac : int -> int
```

einer Fakutätsprozedur. Diese können wir zunächst zu

```
val rec fac = fn n => if n<2 then 1 else n * fac (n-1)
```

umschreiben. Um diese Deklaration in FP-Format zu bringen, deklarieren wir zuerst das passende Funktional

```
val fac' = fn fac => fn n => if n<2 then 1 else n * fac (n-1)
fac' : (int -> int) -> (int -> int)
```

und deklarieren dann

```
val rec fac'' = fac' fac''
fac'' : int -> int
```

Die letzte Deklaration ist in Standard ML allerdings syntaktisch unzulässig. Deshalb weichen wir auf die zulässige Deklaration

```
val rec fac'' = fn x => fac' fac'' x
fac'' : int -> int
```

aus. Überzeugen Sie sich durch Experimente, dass `fac''` in der Tat dieselbe Funktion berechnet wie `fac`.

Wir können also zu jeder rekursiven Prozedurdeklaration

```
fun f x = a
f : t1 -> t2
```

ein Funktional

```
val f' = fn f => fn x => a
f' : (t1 -> t2) -> (t1 -> t2)
```

ohne Rekursion definieren. Mit dem Funktional können wir dann eine Prozedur

```
val rec f'' = fn x => f' f'' x
f'' : t1 -> t2
```

deklarieren, die dieselbe Funktion wie `f` berechnet. Das liefert uns die Idee für die Deklaration einer Prozedur

```
fun fix f' = let val rec f'' = fn x => f' f'' x in f'' end
fix : (('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)
```

mit der wir `f''` ohne sichtbare Rekursion

```
val f'' = fix f'
f'' : t1 -> t2
```

deklarieren können. Die für `f''` erforderliche Rekursion ist in der Prozedur `fix` versteckt.

Machen Sie sich klar, dass Sie mit der gerade definierten Prozedur `fix` in der Tat jede rekursive Prozedurdeklaration durch eine nicht-rekursive Deklarationen ersetzen können, in der Sie `fix` auf dass der rekursiven Prozedurdeklaration entsprechende Funktional anwenden. Beispielsweise können wir eine Prozedur, die die Fakultätsfunktion berechnet, wie folgt deklarieren:

```
val fac = fix (fn fac => fn n =>
                  if n<2 then 1 else n * fac (n-1))
```

Bei polymorphen Prozeduren muss man statt

```
val f'' = fix f'
```

die etwas komplexere Deklaration

```
val f'' = fn x => (fix f') x
```

schreiben, damit `f'` polymorph getypt werden kann (siehe [Programmierung, Kapitel 3.5, Transparenzbedingung]).

Gegeben zwei Typen t_1 und t_2, verhält sich das Triple

$$\langle t_1 \to t_2,\ (t_1 \to t_2) \to (t_1 \to t_2),\ \mathtt{fix}\rangle$$

wie eine R-Struktur. Durch Experimente können Sie sich davon überzeigen, dass für jede Prozedur

$$f:\ (t_1 \to t_2) \to (t_1 \to t_2)$$

die zwei Prozeduren $\mathtt{fix}\, f$ und $f(\mathtt{fix}\, f)$ dieselbe Funktion berechnen. Die Prozedur `fix` verhält sich also wie ein Fixpunktoperator.

Man kann den Fixpunktoperator noch etwas eleganter deklarieren als wir das oben getan haben:

```
fun fix f' = fn x => f' (fix f') x
```

4.3 Vollständige partielle Ordnungen

Wir haben gesehen, wie man in Standard ML eine polymorphe Prozedur deklarieren kann, die sich wie ein Fixpunktoperator verhält. Dabei haben wir ohne Beweise gearbeitet und uns auf unsere Intuition als Programmierer verlassen. Jetzt zeigen wir, wie man mit mathematischen Mitteln R-Strukturen konstruiert. Dazu gehört natürlich ein Beweis, dass es sich auch wirklich um eine R-Struktur mit einem Fixpunktoperator handelt.

Zunächst benötigen wir das Konzept einer vollständigen partiellen Ordnung, kurz VPO genannt. Eine VPO ist ein Paar $\langle X, \leq \rangle$, das aus einer Menge X und einer partiellen Ordnung $\leq$ auf X besteht. Zusätzlich muss eine VPO die Eigenschaft besitzen, dass jede aufsteigende Kette eine kleinste obere Schranke hat. Die genaue Definition von VPOs finden Sie in [Winskel, Kapitel 5.4].

Das Standardbeispiel für VPOs sind Potenzmengen mit der Inklusionsordnung.

Sei $\langle X, \leq \rangle$ eine VPO. Eine Funktion $f \in X \to X$ heißt *monoton* genau dann, wenn $\forall x, y \in X\colon\ x \leq y \Rightarrow f(x) \leq f(y)$.

Proposition 4.3.1 *Sei $\langle X, \leq \rangle$ eine VPO und $f \in X \to X$ monoton. Dann gilt für jede aufsteigende Kette $x_0 \leq x_1 \leq \dots$ in X:*

(1) $f(x_0) \leq f(x_1) \leq \dots$ *ist eine aufsteigende Kette in X.*

(2) $\bigsqcup_{i \in \mathbb{N}} f(x_i) \leq f(\bigsqcup_{i \in \mathbb{N}} x_i)$

Beweis Sei $x_0 \leq x_1 \leq \dots$ eine aufsteigende Kette in X. Da f monoton ist, gilt (1). Aus der Monotonie von f folgt auch, dass $f(x_n) \leq f(\bigsqcup_{i \in \mathbb{N}} x_i)$ für alle $n \in \mathbb{N}$ gilt. Also ist $f(\bigsqcup_{i \in \mathbb{N}} x_i)$ eine obere Schranke für die Kette $f(x_0) \leq f(x_1) \leq \dots$. Da $\bigsqcup_{i \in \mathbb{N}} f(x_i)$ die kleinste obere Schranke dieser Kette ist, folgt (2). □

Sei $\langle X, \leq \rangle$ eine VPO. Eine monotone Funktion $f \in X \to X$ heißt *stetig* genau dann, wenn für jede aufsteigende Kette $x_0 \leq x_1 \leq \dots$ in X die Ungleichung

$$f(\bigsqcup_{i \in \mathbb{N}} x_i) \leq \bigsqcup_{i \in \mathbb{N}} f(x_i)$$

gilt. Die Menge aller stetigen Funktionen in $X \to X$ bezeichnen wir mit $[X \to X]$.

Proposition 4.3.2 *Sei $\langle X, \leq\rangle$ eine VPO und $f \in X \to X$ stetig. Dann gilt für jede aufsteigende Kette $x_0 \leq x_1 \leq \ldots$ in X:*

$$f(\bigsqcup_{i\in\mathbb{N}} x_i) = \bigsqcup_{i\in\mathbb{N}} f(x_i)$$

Diese Eigenschaft bezeichnen wir als Distributivität.

Die Distributivität von stetigen Funktionen folgt aus Proposition 4.3.1 und aus der Definition von Stetigkeit.

Der Fixpunktsatz formuliert die wichtigste Eigenschaft von VPOs.

Satz 4.3.3 (Fixpunktsatz) *Sei $\langle X, \leq\rangle$ eine VPO mit einem kleinsten Element $\perp \in X$ (sprich bottom) und $f \in X \to X$ eine stetige Funktion. Dann hat f einen eindeutig bestimmten kleinsten Fixpunkt x_0. Zusätzlich gilt:*

1. $x_0 = \bigsqcup_{i\in\mathbb{N}} f^n(\perp)$
2. *x_0 ist das kleinste x mit $f(x) \leq x$.*

Sei $\langle X, \leq\rangle$ eine VPO mit einem kleinsten Element. Dann heißt die Funktion, die jeder Funktion in $[X \to X]$ ihren kleinsten Fixpunkt zuordnet, der *Fixpunktoperator* der VPO.

Korollar 4.3.4 *Sei $\langle X, \leq\rangle$ eine VPO mit einem kleinsten Element und sei fix der Fixpunktoperator der VPO. Dann ist*

$$\langle X,\ [X \to X],\ \mathsf{fix}\rangle$$

eine R-Struktur.

Die mit VPOs konstruierten Fixpunktoperatoren ermöglichen es, rekursive Definitionen

$$x \stackrel{\text{def}}{=} f(x)$$

im FP-Format als nicht-rekursive Definitionen der Form

$$x \stackrel{\text{def}}{=} \bigsqcup_{i\in\mathbb{N}} f^n(\perp)$$

zu interpretieren. Wir zeigen mit zwei Fallstudien, dass diese Technik wichtige Anwendungen hat.

4.4 Fallstudie: Rekursive Definition von Mengen

Wir zeigen jetzt, wie man die rekursive Definition einer Menge mithilfe von Inferenzregeln auf eine nicht-rekursive Definition zurückführen kann.

Bisher haben wir nicht genau erklärt, was wir unter einer Inferenzregel verstehen wollen. Um dies zu tun, müssten wir eine logische Sprache einführen, mit der die Prämissen und Konklusionen von Inferenzregeln formuliert werden können. Das ist nicht so einfach. Wir sind aber auch mit einfacheren Mittel in der Lage, mehr über die rekursive Definition von Mengen mit Inferenzregeln zu sagen.

Sei X eine Menge. Eine *X-Regel* ist ein Paar $\langle A, x\rangle$, dass aus einer endlichen Menge $A \subseteq X$ und einem $x \in X$ besteht. Beachten Sie, dass die Menge A endlich sein muss.

Stellen Sie sich nun eine Menge von Inferenzregeln vor, mit denen eine Teilmenge $I \subseteq X$ definiert werden soll. Dann können wir uns jede Inferenzregel als ein Schema vorstellen, dass eine Menge von X-Regeln definiert. Als Beispiel wählen wir

$$\frac{}{3 \in P} \qquad \frac{}{5 \in P} \qquad \frac{x \in P \quad y \in P \quad z = x \cdot y}{z \in P}$$

Hier ist $X = \mathbb{Z}$. Die erste und zweite Inferenzregel definieren je genau eine $\mathbb{Z}$-Regel, nämlich

$$\langle \emptyset, 3\rangle \quad \text{und} \quad \langle \emptyset, 5\rangle$$

Die dritte Inferenzregel definiert dagegen eine unendliche Menge von $\mathbb{Z}$-Regeln:

$$\{\, \langle \{x, y\}, z\rangle \mid x, y \in \mathbb{Z} \wedge z = x \cdot y \,\}$$

Beachten Sie, dass diese $\mathbb{Z}$-Regeln im Gegensatz zur Inferenzregel entweder eine oder zwei Prämissen haben.

Wir können jetzt die rekursive Definition einer Menge mit Inferenzregeln in zwei Schritten beschreiben. Im ersten Schritt leiten wir aus dem Kontext und den Inferenzregeln eine Menge X und eine Menge R von X-Regeln ab. Dazu braucht man mathematische Erfahrung, um die Inferenzregel richtig zu verstehen. Der zweite Schritt legt zu R eine Menge $I_R \subseteq X$ fest, die als die durch die Inferenzregel definierte Menge zu verstehen ist. Den zweiten Schritt können wir vollständig formal beschreiben. Dabei werden wir die Menge I_R ohne Rekursion definieren. Wir führen also eine rekursive Definition mit Inferenzregeln auf eine nicht-rekursive Definition zurück.

Sei X eine Menge und R eine Menge von X-Regeln. Dann definieren wir eine Funktion

$$\hat{R} \in \mathcal{P}(X) \to \mathcal{P}(X)$$
$$\hat{R}(A) = \{\, x \mid \exists B\colon\ \langle B, x\rangle \in R \wedge B \subseteq A \,\}$$

Die Menge $\hat{R}(A)$ enthält offensichtlich genau die Elemente, die man durch einmaliges Anwenden einer Regel in R aus den Elementen von A ableiten kann.

Offensichtlich ist $\hat{R}$ bezüglich der Inklusionsordnung auf $\mathcal{P}(X)$ monoton. Also haben wir

$$\emptyset \subseteq \hat{R}(\emptyset) \subseteq \hat{R}^2(\emptyset) \subseteq \ldots$$

Machen Sie sich jetzt klar, dass $\hat{R}^n(\emptyset)$ genau die Elemente enthält, für die man einem Ableitungsbaum mit einer Tiefe kleiner als n angeben kann.

Die Menge I_R können wir jetzt wie folgt definieren:

$$I_R \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} \hat{R}^i(\emptyset)$$

Offensichtlich enthält I_R genau die Elemente, die man mit den Regeln aus R ableiten kann.

Proposition 4.4.1 *Sei X eine Menge und R eine Menge von X-Regeln. Dann ist $\hat{R}$ eine stetige Funktion $\mathcal{P}(X) \to \mathcal{P}(X)$ (bezüglich der VPO $\langle \mathcal{P}(X), \subseteq\rangle$).*

Beweis Wir wissen bereits, dass $\hat{R}$ monoton ist. Sei $A_0 \subseteq A_1 \subseteq \ldots$ eine aufsteigende Kette in $\mathcal{P}(X)$. Wir müssen die folgende Ungleichung zeigen:

$$\hat{R}(\bigcup_{i \in \mathbb{N}} A_i) \subseteq \bigcup_{i \in \mathbb{N}} \hat{R}(A_i)$$

Sei $x \in \hat{R}(\bigcup_{i \in \mathbb{N}} A_i)$. Dann existiert eine Regel $(B, x) \in R$ mit $B \subseteq \bigcup_{i \in \mathbb{N}} A_i$. Da B endlich ist, gibt es ein n, so dass $B \subseteq A_n$. Also $x \in \hat{R}(A_n) \subseteq \bigcup_{i \in \mathbb{N}} \hat{R}(A_i)$. $\square$

Beachten Sie, dass der Beweis ausnützt, dass jede Regel nur endlich viele Prämissen hat.

Sei X eine Menge und R eine Menge von X-Regeln. Wir sagen dass eine Menge $A \subseteq X$ *unter R abgeschlossen* ist genau dann, wenn $\hat{R}(A) \subseteq A$.

Der Fixpunktsatz liefert uns jetzt die folgende Proposition.

Proposition 4.4.2 *Sei X eine Menge und R eine Menge von X-Regeln. Sei fix der Fixpunktoperator der VPO $\langle \mathcal{P}(X), \subseteq \rangle$. Dann gilt:*

1. $I_R = \mathit{fix}(\hat{R}) = \hat{R}(I_R)$.
2. *I_R ist die kleinste Teilmenge von X, die unter R abgeschlossen ist.*

4.5 Fallstudie: Rekursive Definition von partiellen Funktionen

Seien X und Y Mengen. Da es mathematisch gesehen einfacher ist, mit totalen statt mit partiellen Funktionen zu arbeiten, repräsentieren wir die partiellen Funktionen in

$$X \rightharpoonup Y$$

durch die totalen Funktionen in

$$X \to Y_\perp$$

wobei

$$Y_\perp \stackrel{\text{def}}{=} Y \cup \perp$$

und $\perp$ ein einmal gewähltes Objekt ist, das kein Element von Y ist. Eine partielle Funktion $f \in X \rightharpoonup Y$ repräsentieren wir dabei durch die totale Funktion

$$f' \in X \to Y_\perp$$
$$f'(x) = \text{if } \exists y \in Y\colon f(x) = y \text{ then } f(x) \text{ else } \perp$$

Das bedeutet, dass

$$f'(x) = \perp$$

genau dann gilt, wenn f auf x undefiniert ist.

Wir definieren jetzt auf $X \to Y_\perp$ eine binäre Relation wie folgt:

$$f \leq g \stackrel{\text{def}}{\iff} \forall x \in X\colon f(x) = \perp \vee f(x) = g(x).$$

Proposition 4.5.1 *$\langle X \to Y_\perp, \leq \rangle$ ist eine VPO mit dem kleinsten Element $\lambda x \in X.\, \perp$. Wenn $f_0 \leq f_1 \leq \ldots$ eine aufsteigende Kette in $X \to Y_\perp$ ist, dann ist*

$$f(x) \stackrel{\text{def}}{=} \begin{cases} y & \textit{falls } \exists i\colon\ f_i(x) = y \in Y \\ \perp & \textit{falls } \forall i\colon\ f_i(x) = \perp \end{cases}$$

die kleinste obere Schranke für die Kette.

Beweis Offensichtlich ist $\leq$ eine partielle Ordnung auf $X \to Y_\perp$. Es ist auch klar, dass $\lambda\, x \in X\,.\,\perp$ das kleinste Element ist. Sei

$$f_0 \leq f_1 \leq \dots$$

eine aufsteigende Kette in $X \to Y_\perp$. Die Definition der Ordnung erzwingt, dass sich die Glieder der Kette nur dadurch unterscheiden, dass spätere Glieder für ein $x \in X$ statt dem bisherigen $\perp$ ein $y \in Y$ liefern dürfen. Wenn ein Glied für ein $x \in X$ bereits ein $y \in Y$ liefert, müssen alle nachfolgenden Glieder für x genau dieses y liefern. Damit ist die Definition der Funktion f zulässig und es muss sich um die kleinste obere Schranke handeln. □

Stellen Sie sich jetzt vor, dass wir eine Funktion $f \in X \to Y_\perp$ durch eine rekursive Gleichung

$$f(x) \stackrel{\text{def}}{=} a$$

definieren wollen. Wir haben bereits am Beispiel von Standard ML gesehen, dass wir diese Definition mit dem Funktional

$$F \in (X \to Y_\perp) \to (X \to Y_\perp)$$
$$F(f) = \lambda\, x \in X\,.\, a$$

in eine rekursive Definition im FP-Format umschreiben können:

$$f \stackrel{\text{def}}{=} F(f)$$

Betrachten Sie nun die wie folgt definierten Funktionen $f_i \in X \to Y_\perp$:

$$f_1 = F(\lambda\, x \in X\,.\,\perp)$$
$$f_2 = F(f_1) = F^2(\lambda\, x \in X\,.\,\perp)$$
$$f_3 = F(f_2) = F^3(\lambda\, x \in X\,.\,\perp)$$
$$f_n = F(f_{n-1}) = F^n(\lambda\, x \in X\,.\,\perp)$$

Machen Sie sich klar, dass die rekursiv definierte Funktion f bis zur Rekursionstiefe $n-1$ genauso „rechnet“ wie die Funktion f_n. Ab der Rekursionstiefe n verwendet f_n jedoch die Funktion $\lambda\, x \in X\,.\,\perp$ anstelle von f. Wenn $f(x)$ mit einer Rekursionstiefe kleiner n „berechnet“ wird, gilt folglich $f_n(x) = f(x)$. Je größer wir also n wählen, desto besser approximiert f_n die Funktion f.

Wir werden nur rekursive Definitionen betrachten, für die das Funktional F monoton ist (bezüglich der VPO $\langle X \to Y_\perp, \leq\rangle$). Dann haben wir eine aufsteigende Kette

$$F^0(\lambda\, x \in X\,.\,\perp) \leq F^1(\lambda\, x \in X\,.\,\perp) \leq F^2(\lambda\, x \in X\,.\,\perp) \leq \dots$$

in $X \to Y_\perp$. Aus der obige Überlegung folgt jetzt, dass die kleinste obere Schranke dieser Kette genau die Funktion ist, die wir intuitiv mit der Definition

$$f(x) \stackrel{\text{def}}{=} a$$

definieren wollen.

Damit können wir die rekursive Definition

$$f(x) \stackrel{\text{def}}{=} a$$

durch die nicht-rekursive Definition

$$f \stackrel{\text{def}}{=} \bigsqcup_{i \in \mathbb{N}} F^i(\lambda\, x \in X \,\bullet\, \perp)$$

ersetzen. Dabei setzen wir voraus, dass das Funktional F monoton ist. Wenn wir zusätzlich verlangen, dass F stetig ist, liefert uns der Fixpunktsatz

$$f \stackrel{\text{def}}{=} \bigsqcup_{i \in \mathbb{N}} F^i(\lambda\, x \in X \,\bullet\, \perp) = \mathit{fix}(F) = F(\mathit{fix}(F))$$

Dabei ist *fix* der Fixpunktoperator der VPO $\langle X \to Y_\perp, \leq \rangle$.

4.6 Zusammenfassung

Wir haben drei Schemata kenngelernt, die rekursive Definitionen auf nicht-rekursive Definitionen zurückführen:

1. Rekursive Prozedurdeklarationen in Standard ML können mit einem Fixpunktoperator auf nicht-rekursive Deklarationen zurückgeführt werden. Allerdings wird der Fixpunktoperator als rekursive Prozedur deklariert.

2. Rekursive Definitionen von Mengen mit Inferenzregeln können auf nicht-rekursive Definitionen der Form

   $$I_R \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} \hat{R}^i(\emptyset)$$

   zurückgeführt werden. Dabei wird vorausgesetzt, dass eine Grundmenge X mit $\hat{R} \in \mathcal{P}(X) \to \mathcal{P}(X)$ a priori gegeben ist. $\hat{R}$ ist stetig und I_R ist der kleinste Fixpunkt von $\hat{R}$.

3. Rekursive Definitionen von Funktionen in $X \to Y_\perp$ können auf nicht-rekursive Definitionen der Form

$$f \stackrel{\text{def}}{=} \bigsqcup_{i \in \mathbb{N}} F^i(\lambda\, x \in X \,.\, \perp)$$

zurückgeführt werden. Dabei muss das Funktional F monoton sein. Wenn F zudem stetig ist, ist f der kleinste Fixpunkt von F.

Bei den Schemata (2) und (3) wir Rekursion auf die Iteration einer Funktion und die Existenz von kleinsten oberen Schranken zurückgeführt. Diese Iterationen kann man auch als eine primitive Form von Rekursion ansehen, die mathematisch gut beherrschbar ist.

Im Folgenden verwende wir 4 rekursive Definitionsprinzipien:

1. Definitionen von syntaktischen Mengen durch abstrakte Grammatiken. Diese Definitionen lassen sich im Prinzip auf Schema (2) zurückführen, allerdings muss man dann eine Grundmenge X a priori angeben, wofür man wieder Rekursion benötigt.

2. Definition von totalen Funktionen auf syntaktischen Mengen durch strukturelle Rekursion (siehe Abschnitt 3.2). Dieses Definitionsprinzip verwenden wir stets für denotationale Semantiken.

3. Definition von Mengen gemäß Schema (2). Dieses Definitionsprinzip verwenden wir stets für operationale Semantiken.

4. Definition von Funktionen gemäß Schema (3). Dieses Definitionsprinzip verwenden wir zusätzlich zu Prinzip (2) für denotationale Semantiken von Sprachen, die ein Rekursionskonstrukt enthalten (zum Beispiel rekursive Prozedurdeklarationen oder Schleifen).

Zu den Definitionsprinzipien (1) und (2) gehört die Beweistechnik strukturelle Induktion. Zu Definitionsprinzip (3) gehört Regelinduktion. Zu Definitionsprinzip (4) gehört natürliche Induktion (also Induktion für $\mathbb{N}$).

Kapitel 5

Programme mit Schleifen

Wir betrachten jetzt eine einfache imperative Programmiersprache IMP. IMP hat keine Prozeduren und nur den Typ `int`. Verglichen mit AA hat IMP ein wichtiges neues Konzept, nämlich Schleifen. Schleifen stellen eine einfache Form von Rekursion zu Verfügung, die man *Iteration* nennt. Die Syntax und die operationale Semantik von IMP lassen sich ohne Schwierigkeiten mit den Konzepten definieren, die wir bei AA kennengelernt haben. Dagegen erfordert die denotationale Semantik von IMP eine neue Idee, da die Denotation von Schleifen mithilfe von rekursiv definierten Funktionen definiert werden muss. Diese Definitionen haben wir in Abschnitt 4.5 gut vorbereitet. Es ist jetzt nicht mehr ganz so einfach, die Übereinstimmung der denotationalen und operationalen Semantik zu zeigen.

Lesematerial

[Winskel, Kapitel 2 und 5]

5.1 Syntax und operationale Semantik von IMP

Die Syntax von IMP definieren wir in Abbildung 5.1 bis auf Kleinigkeiten so wie Winskel. Wir wollen IMP als eine Teilsprache von C oder Java verstehen.

Für die Semantik von IMP benötigen wir genau wie bei AA Funktionen

$$\sigma \in \Sigma = Loc \to \mathbb{Z} \qquad \textit{Zustand}$$

die wir diesmal *Zustände* nennen.

Die operationale und die denotationale Semantik von arithmetischen und boolschen Ausdrücken wird analog zu AA definiert. Dabei ist zu beachten, dass die Semantik von boolschen Ausdrücken erst definiert werden kann, nach dem die

n	$\in$	$\mathbb{Z}$	*Zahl*
X	$\in$	*Loc*	*Lokation*
a	$\in$	*Aexp* $=$	*Arithmetischer Ausdruck*
		$n \mid x \mid -a \mid a_1 + a_2$	
b	$\in$	*Bexp* $=$	*Boolscher Ausdruck*
		$\mathsf{true} \mid \mathsf{false} \mid a_1 \leq a_2 \mid \neg b \mid$ $b_1 \wedge b_2$	
c	$\in$	*Com* $=$	*Kommando*
		$\mathsf{skip} \mid X := a \mid c_1; c_2 \mid$ $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid$ $\mathsf{while}\ b\ \mathsf{do}\ c$	

Abbildung 5.1: Syntax von IMP

Semantik von arithmetischen Ausdrücken definiert ist. Im folgenden werden wir lediglich die Denotationsfunktionen

$$\mathcal{A} \in \mathit{Aexp} \to \Sigma \to \mathbb{Z}$$
$$\mathcal{B} \in \mathit{Bexp} \to \Sigma \to \mathbb{B}$$

verwenden. Dabei ist $\mathbb{B} = \{0, 1\}$.

Die operationale Semantik von Kommandos definieren wir in Abbildung 5.2 als eine Relation

$$\sigma \vdash c \Rightarrow \sigma' \iff \langle \sigma, c, \sigma' \rangle \in \mathit{OS} \subseteq \Sigma \times \mathit{Com} \times \Sigma$$

die genau dann erfüllt sein soll, wenn die Ausführung des Kommandos c mit dem *Anfangszustand* σ mit dem *Endzustand* σ' terminiert. Mit Schleifen können wir offensichtlich divergierende Programme schreiben, zum Beispiel

```
while true do skip
```

Die Regeln in Abbildung 5.2 formulieren direkt unsere operationalen Vorstellungen über die Ausführung von Kommandos. Wir werden gleich sehen, dass die denotationale Semantik nicht so einfach zu verstehen ist. Die operationale Semantik eignet sich bei IMP also besonders gut als definierende Semantik.

$$\frac{\sigma \in \Sigma}{\sigma \vdash \mathsf{skip} \Rightarrow \sigma} \qquad \frac{\mathcal{A}[\![a]\!]\sigma = n \quad \sigma' = \sigma[n/X]}{\sigma \vdash X := a \Rightarrow \sigma'}$$

$$\frac{\sigma \vdash c_1 \Rightarrow \sigma' \quad \sigma' \vdash c_2 \Rightarrow \sigma''}{\sigma \vdash c_1; c_2 \Rightarrow \sigma''}$$

$$\frac{\mathcal{B}[\![b]\!]\sigma = 0 \quad \sigma \vdash c_2 \Rightarrow \sigma'}{\sigma \vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \Rightarrow \sigma'} \qquad \frac{\mathcal{B}[\![b]\!]\sigma = 1 \quad \sigma \vdash c_1 \Rightarrow \sigma'}{\sigma \vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \Rightarrow \sigma'}$$

$$\frac{\mathcal{B}[\![b]\!]\sigma = 0}{\sigma \vdash \mathsf{while}\ b\ \mathsf{do}\ c \Rightarrow \sigma}$$

$$\frac{\mathcal{B}[\![b]\!]\sigma = 1 \quad \sigma \vdash c \Rightarrow \sigma' \quad \sigma' \vdash \mathsf{while}\ b\ \mathsf{do}\ c \Rightarrow \sigma''}{\sigma \vdash \mathsf{while}\ b\ \mathsf{do}\ c \Rightarrow \sigma''}$$

Abbildung 5.2: Operationale Semantik von Kommandos

5.2 Denotationale Semantik von Kommandos mit Standard ML

Bevor wir die denotationale Semantik von Kommandos mathematisch definieren, werden wir sie zuerst in Standard ML implementieren. Das hat den Vorteil, dass wir mit einer vertrauten Notation arbeiten und die Semantik ausführen können. Zudem können wir uns durch Experimente von der computationalen Korrektheit der Semantik überzeugen.

Wir nehmen an, dass wir die Syntax und Semantik von Ausdrücken bereits implementiert haben:

```
type loc
type state = loc -> int
val update : state * loc * int -> state
type aexp
type bexp
val evala  : aexp -> state -> int
val evalb  : bexp -> state -> bool
```

Die Syntax von Kommandos implementieren wir wie folgt:

```
datatype com =
    SKIP
  | ASSIGN of loc * aexp
  | SEQ    of com * com
  | IF     of bexp * com * com
  | WHILE  of bexp * com
```

Wir schreiben jetzt eine Prozedur

```
evalc : com -> state -> state
```

deren Anwendung auf ein Kommando eine Prozedur liefert, die die Denotation des Kommandos berechnet. Die Denotationen von Kommandos verstehen wir vorerst als partielle Prozeduren

$$\Sigma \rightharpoonup \Sigma$$

Wie für eine denotationale Semantik erforderlich, definieren wir `evalc` durch strukturelle Rekursion über `com`:

```
fun while (beta,phi) =
    let
        fun iter s = if beta s then iter (phi s) else s
    in
        iter
    end

fun evalc SKIP           = (fn s => s)
  | evalc (ASSIGN(x,a)) = (fn s => update(s, x, evala a s))
  | evalc (SEQ(c1,c2))  = (fn s => evalc c2 (evalc c1 s))
  | evalc (IF(b,c1,c2)) = (fn s => if evalb b s then evalc c1 s
                                   else evalc c2 s)
  | evalc (WHILE(b,c))  = while (evalb b, evalc c)
```

Die Anwendung der Hilfsprozedur `while` liefert ein rekursive Prozedur `iter`, die die Denotation der entsprechenden Schleife berechnet.

Als Nächstes definieren wir `iter` mit dem Fixpunkt-Operator statt mit Rekursion:

```
fun while (beta,phi) =
   fix (fn iter => fn s => if beta s then iter (phi s) else s)
```

Jetzt schieben wir die Anwendung von `fix` in die Definition von `evalc`:

```
fun gamma (beta,phi) =
    fn iter => fn s => if beta s then iter (phi s) else s
```

```
fun evalc SKIP           = (fn s => s)
  | evalc (ASSIGN(x,a)) = (fn s => update(s, x, evala a s))
  | evalc (SEQ(c1,c2))  = (fn s => evalc c2 (evalc c1 s))
  | evalc (IF(b,c1,c2)) = (fn s => if evalb b s then evalc c1 s
                                   else evalc c2 s)
  | evalc (WHILE(b,c))  = fix (gamma (evalb b, evalc c))
```

Damit haben wir eine gute Vorlage für die mathematische Definition der denotationale Semantik von Kommandos.

5.3 Denotationale Semantik von Kommandos mathematisch

Da es mathematisch gesehen einfacher ist, mit totalen statt mit partiellen Funktionen zu arbeiten, repräsentieren wir die Denotationen von Kommandos jetzt durch Funktionen

$$\phi, \psi \in \Sigma \to \Sigma_\perp$$

wobei

$$\Sigma_\perp = \Sigma \cup \perp$$

und $\perp$ ein einmal gewähltes Objekt ist, das kein Element von Σ ist. Dabei bedeutet

$$\phi\sigma = \perp$$

dass die Ausführung des Kommandos mit der Denotation ϕ für den Anfangszustand σ divergiert.

In Abschnitt 4.5 haben wir eine partielle Ordnung auf $\Sigma \to \Sigma_\perp$ definiert, mit der $\Sigma \to \Sigma_\perp$ eine VPO bildet. Sei

$$\mathit{fix} \in [(\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)] \to (\Sigma \to \Sigma_\perp)$$

die Funktion, die jeder stetige Funktion ihren kleinsten Fixpunkt zuordnet. Damit haben wir eine R-Struktur

$$\langle \Sigma \to \Sigma_\perp,\ [(\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)],\ \mathit{fix} \rangle$$

Abbildung 5.3 definiert die denotationale Semantik von Kommandos mathematisch. Die Definition folgt der Implementierung in Standard ML, behandelt aber Divergenz explizit. Damit die Definition zulässig ist, müssen wir natürlich noch zeigen, dass wir den Fixpunkt-Operator *fix* tatsächlich auf eine stetige Funktion anwenden.

$$\mathcal{C} \in Com \to (\Sigma \to \Sigma_\perp)$$

$$\begin{aligned}
\mathcal{C}[\![\mathsf{skip}]\!] &= \lambda\,\sigma\,.\,\sigma\\
\mathcal{C}[\![X := a]\!] &= \lambda\,\sigma\,.\,\sigma[\mathcal{A}[\![a]\!]\sigma/X]\\
\mathcal{C}[\![c_1; c_2]\!] &= \lambda\,\sigma\,.\,\text{if } \mathcal{C}[\![c_1]\!]\sigma = \perp \text{ then } \perp \text{ else } \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)\\
\mathcal{C}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= \lambda\,\sigma\,.\,\text{if } \mathcal{B}[\![b]\!]\sigma = 0 \text{ then } \mathcal{C}[\![c_2]\!]\sigma \text{ else } \mathcal{C}[\![c_1]\!]\sigma\\
\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] &= \mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))
\end{aligned}$$

$$\Gamma \in (\Sigma \to \mathbb{B}) \times (\Sigma \to \Sigma_\perp) \to ((\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp))$$

$$\Gamma(\beta, \phi) = \lambda\,\psi\,.\,\lambda\,\sigma\,.\,\text{if } \beta\sigma = 0 \text{ then } \sigma \text{ else if } \phi\sigma = \perp \text{ then } \perp \text{ else } \psi(\phi\sigma)$$

Abbildung 5.3: Denotationale Semantik von Kommandos

Proposition 5.3.1 *Sei $\beta \in \Sigma \to \mathbb{B}$ und $\phi \in \Sigma \to \Sigma_\perp$. Dann ist*

$$\gamma \in (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$$
$$\gamma\psi = \lambda\,\sigma\,.\,\text{if } \beta\sigma = 0 \text{ then } \sigma \text{ else if } \phi\sigma = \perp \text{ then } \perp \text{ else } \psi(\phi\sigma)$$

stetig.

Beweis Die Monotonie von γ ist offensichtlich. Wir müssen also nur noch die Distributivität von γ zeigen. Sei $\phi_0 \leq \phi_1 \leq \dots$ eine aufsteigende Kette und $\sigma \in \Sigma$. Es genügt zu zeigen, dass

$$\gamma(\bigsqcup_{i\in\mathbb{N}} \phi_i)\sigma = \perp \quad\vee\quad \gamma(\bigsqcup_{i\in\mathbb{N}} \phi_i)\sigma = (\bigsqcup_{i\in\mathbb{N}} \gamma\phi_i)\sigma$$

gilt. Sei

$$\gamma(\bigsqcup_{i\in\mathbb{N}} \phi_i)\sigma \neq \perp \tag{$*$}$$

Wir unterscheiden zwei Fälle:

1. $\beta\sigma = 0$. Dann $\gamma\psi\sigma = \sigma$ für alle ψ. Also

   $$\gamma(\bigsqcup_{i\in\mathbb{N}} \phi_i)\sigma = \sigma = (\bigsqcup_{i\in\mathbb{N}} \gamma\phi_i)\sigma$$

2. $\beta\sigma = 1$. Dann folgt $\phi\sigma \neq \perp$ mit $(*)$. Also $\gamma\psi\sigma = \psi(\phi\sigma)$ für alle ψ. Also gibt es ein n mit

   $$\perp \neq \gamma(\bigsqcup_{i\in\mathbb{N}} \phi_i)\sigma = (\bigsqcup_{i\in\mathbb{N}} \phi_i)(\phi\sigma) = \phi_n(\phi\sigma) = \gamma\phi_n\sigma = (\bigsqcup_{i\in\mathbb{N}} \gamma\phi_i)\sigma \qquad \square$$

Schauen Sie sich diesen Beweis genau an. Er benutzt viele wesentlichen Eigenschaften der VPO $\langle \Sigma \to \Sigma_\perp, \leq \rangle$.

Da wir die Denotation von Schleifen auch mit

$$\mathcal{C}[\![\textsf{while } b \textsf{ do } c]\!] = \bigsqcup_{i \in \mathbb{N}} (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^i (\lambda\, \sigma \in \Sigma \,.\, \perp)$$

beschreiben können, ist der Einsatz der Konzepte Stetigkeit und Fixpunktoperator nicht zwingend. Es genügt, die VPO $\langle \Sigma \to \Sigma_\perp, \leq \rangle$ zu verstehen. Winskel wählt diese mathematisch elementarere Beschreibung für seine denotationale Semantik von IMP. Zudem verwendet er statt der VPO $\langle \Sigma \to \Sigma_\perp, \leq \rangle$ die VPO $\langle \Sigma \rightharpoonup \Sigma, \subseteq \rangle$ der partiellen Funktionen von Σ nach Σ. Unsere Verwendung von totalen Funktionen von Σ nach $\Sigma_\perp$ und des Fixpunktoperators erfordert etwas mehr mathematisches Rüstzeug, liefert dafür aber eine Formulierung, die für Beweise besser geeignet ist. Hier ist eine Kostprobe.

Proposition 5.3.2 *Sei $b \in Bexp$ und $c \in Com$. Dann*

$$\mathcal{C}[\![\textsf{while } b \textsf{ do } c]\!] = \mathcal{C}[\![\textsf{if } b \textsf{ then } c; \textsf{while } b \textsf{ do } c \textsf{ else skip}]\!]$$

Beweis Sei $\sigma \in \Sigma$. Dann:

$$\begin{aligned}
\mathcal{C}[\![\textsf{while } b \textsf{ do } c]\!]\sigma &= \mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ \sigma \\
&= (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ (\mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!])))\ \sigma \\
&= (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ (\mathcal{C}[\![\textsf{while } b \textsf{ do } c]\!])\ \sigma \\
&= \text{if } \mathcal{B}[\![b]\!]\sigma = 0 \text{ then } \sigma \\
&\ \ \text{else if } \mathcal{C}[\![c]\!]\sigma = \perp \text{ then } \perp \\
&\ \ \qquad \text{else } (\mathcal{C}[\![\textsf{while } b \textsf{ do } c]\!])(\mathcal{C}[\![c]\!]\sigma) \\
&= \text{if } \mathcal{B}[\![b]\!]\sigma = 0 \text{ then } \mathcal{C}[\![\textsf{skip}]\!]\sigma \\
&\ \ \text{else } \mathcal{C}[\![c; \textsf{while } b \textsf{ do } c]\!]\sigma \\
&= \mathcal{C}[\![\textsf{if } b \textsf{ then } c; \textsf{while } b \textsf{ do } c \textsf{ else skip}]\!]\sigma
\end{aligned}$$

Die zweite Gleichung folgt aus der Fixpunkteigenschaft von *fix*, alle anderen Gleichungen folgen direkt aus den Definitionen. □

Die Implementierung der denotationalen Semantik in Standard ML unterscheidet sich gegenüber der mathematischen Formulierung dadurch, dass die Prozedur, die die Denotation eines Kommandos berechnet, divergiert, falls das Kommando divergiert. Es stellt sich die Frage, ob man die denotationale Semantik so implementieren kann, dass die für ein Kommando berechnete Prozedur immer terminiert, also für den Fall, dass das Kommando divergiert, einen Wert liefert, der $\perp$ repräsentiert. Wir werden später sehen, dass dies unmöglich ist, da das Halteproblem von IMP unentscheidbar ist.

5.4 Übereinstimmung der operationalen und denotationalen Semantik

Da wir die denotationale Semantik von IMP mit einigem neuerworbenen mathematischen Rüstzeug konstruiert haben, ist die Übereinstimmung der denotationalen mit der operationalen Semantik jedenfalls nicht offensichtlich. Wir führen den Beweis in zwei Teilen.

Proposition 5.4.1 $\forall \sigma \in \Sigma\ \forall c \in \mathit{Com}\ \forall \sigma' \in \Sigma:\ \sigma \vdash c \Rightarrow \sigma' \Rightarrow \mathcal{C}[\![c]\!]\sigma = \sigma'$

Beweis Durch Regelinduktion über die Definition der operationalen Semantik. Nur die Beweise für die Regeln für Schleifen zeigen verglichen mit AA Neues. Daher geben wir nur die Beweise für diese beiden Regeln und die Regel für Sequentialisierung an.

1. *Regel für Sequentialisierung.* Wir nehmen an:

   $$\mathcal{C}[\![c_1]\!]\sigma = \sigma' \ \wedge\ \mathcal{C}[\![c_2]\!]\sigma' = \sigma''$$

   Wir müssen zeigen:

   $$\mathcal{C}[\![c_1; c_2]\!]\sigma = \sigma''$$

   Das folgt mit der Definition von $\mathcal{C}$ und den Annahmen:

   $$\begin{aligned}\mathcal{C}[\![c_1; c_2]\!]\sigma &= \text{if } \mathcal{C}[\![c_1]\!]\sigma = \bot \text{ then } \bot \text{ else } \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)\\ &= \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)\\ &= \sigma''\end{aligned}$$

2. *Erste Regel für Schleifen.* Wir nehmen an:

   $$\mathcal{B}[\![b]\!]\sigma = 0$$

   Wir müssen zeigen:

   $$\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma = \sigma$$

   Das folgt mit den Definitionen von $\mathcal{C}$, der Fixpunkteigenschaft der Definition von Γ und der Annahme:

   $$\begin{aligned}\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma &= \mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ \sigma\\ &= (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ (\mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!])))\ \sigma\\ &= \text{if } \mathcal{B}[\![b]\!]\sigma = 0 \text{ then } \sigma \text{ else } \cdots\\ &= \sigma\end{aligned}$$

3. *Zweite Regel für Schleifen.* Wir nehmen an:

   $$\mathcal{B}[\![b]\!]\sigma = 1 \ \wedge\ \mathcal{C}[\![c]\!]\sigma = \sigma' \ \wedge\ \mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma' = \sigma''$$

   Wir müssen zeigen:

   $$\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma = \sigma''$$

   Das folgt mit den Definitionen von $\mathcal{C}$, der Fixpunkteigenschaft der Definition von Γ und den Annahmen:

   $$\begin{aligned}\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma &= \mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ \sigma\\ &= (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ (\mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!])))\ \sigma\\ &= \text{if } \mathcal{B}[\![b]\!]\sigma = 0 \text{ then } \sigma\\ &\quad\ \text{else if } \mathcal{C}[\![c]\!]\sigma = \bot \text{ then } \bot\\ &\qquad\quad \text{else } (\mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!])))\ (\mathcal{C}[\![c]\!]\sigma)\\ &= \mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ \sigma'\\ &= \mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma'\\ &= \sigma''\end{aligned}$$ □

Beachten Sie, dass dieser Beweis die Fixpunkteigenschaft verwendet. Wir profitieren also davon, dass wir gezeigt haben, dass Γ stetige Funktionen liefert (Proposition 5.3.1).

Proposition 5.4.2 $\forall \sigma \in \Sigma\ \forall c \in \mathit{Com}\ \forall \sigma' \in \Sigma :\ \mathcal{C}[\![c]\!]\sigma = \sigma' \Rightarrow \sigma \vdash c \Rightarrow \sigma'$

Beweis Durch strukturelle Induktion über *Com*. Wer benötigen also einen Beweisteil für jede Gleichung der denotationalen Semantik. Interessant ist der Beweisteil für Schleifen, die restlichen Beweisteile sind einfach und analog zu AA. Wir zeigen die Beweisteile für Sequentialsierung und Schleifen.

1. Sei $\mathcal{C}[\![c_1; c_2]\!]\sigma = \sigma'$. Es genügt zu zeigen, dass ein $\sigma'' \in \Sigma$ existiert mit

   $$\sigma \vdash c_1 \Rightarrow \sigma'' \ \wedge\ \sigma'' \vdash c_2 \Rightarrow \sigma'$$

   Mit der Annahme und der Definition von $\mathcal{C}$ folgt:

   $$\begin{aligned}\sigma' &= \mathcal{C}[\![c_1; c_2]\!]\sigma\\ &= \text{if } \mathcal{C}[\![c_1]\!]\sigma = \bot \text{ then } \bot \text{ else } \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)\\ &= \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)\end{aligned}$$

   Also existiert ein σ'' mit

   $$\mathcal{C}[\![c_1]\!]\sigma = \sigma'' \ \wedge\ \mathcal{C}[\![c_2]\!]\sigma'' = \sigma'$$

   Jetzt liefert die Induktionsannahme

   $$\sigma \vdash c_1 \Rightarrow \sigma'' \ \wedge\ \sigma'' \vdash c_2 \Rightarrow \sigma'$$

2. Sei $\mathcal{C}[\![\textsf{while}\ b\ \textsf{do}\ c]\!]\sigma = \sigma'$. Wir müssen zeigen, dass

$$\sigma \vdash \textsf{while}\ b\ \textsf{do}\ c \Rightarrow \sigma'$$

gilt. Zuerst bekommen wir mit der Annahme sowie der Definition von $\mathcal{C}$ und des Fixpunktoperators:

$$\begin{aligned}\sigma' &= \mathcal{C}[\![\textsf{while}\ b\ \textsf{do}\ c]\!]\sigma \\ &= \mathit{fix}\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ \sigma \\ &= (\bigsqcup_{i\in\mathbb{N}}(\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^i(\lambda\,\sigma\,.\,\bot))\ \sigma\end{aligned}$$

Folglich gibt es ein $n \in \mathbb{N}$ mit

$$\sigma' = (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^n(\lambda\,\sigma\,.\,\bot))\ \sigma$$

Also genügt es zu zeigen

$$\begin{aligned}&\forall n \in \mathbb{N}\ \forall \sigma \in \Sigma\ \forall \sigma' \in \Sigma: \\ &\qquad \sigma' = (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^n(\lambda\,\sigma\,.\,\bot))\ \sigma \ \Rightarrow\ \sigma \vdash \textsf{while}\ b\ \textsf{do}\ c \Rightarrow \sigma'\end{aligned}$$

Dies Behauptung zeigen wir durch Induktion über $n \in \mathbb{N}$.

a) *Sei* $n = 0$. Dann ist die Implikation trivialerweise erfüllt, da $\sigma' \neq \bot$.

b) *Sei* $n > 0$. Dann

$$\begin{aligned}\sigma' &= (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^n(\lambda\,\sigma\,.\,\bot))\ \sigma \\ &= (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))\ (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^{n-1}(\lambda\,\sigma\,.\,\bot))\ \sigma\end{aligned}$$

Wir unterscheiden jetzt zwei Fälle:

i. *Sei* $\mathcal{B}[\![b]\!]\sigma = 0$. Dann $\sigma = \sigma'$ nach Definition von Γ. Also folgt $\sigma \vdash \textsf{while}\ b\ \textsf{do}\ c \Rightarrow \sigma'$ mit der ersten Regel für Schleifen.

ii. *Sei* $\mathcal{B}[\![b]\!]\sigma = 1$. Dann folgt aus der Definition von Γ, dass ein σ'' existiert mit

$$\sigma'' = \mathcal{C}[\![c]\!]\sigma$$

und

$$\sigma' = (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^{n-1}(\lambda\,\sigma\,.\,\bot)\ \sigma''$$

Mit der äußeren Induktionsannahme folgt

$$\sigma \vdash c \Rightarrow \sigma''$$

Mit der inneren Induktionsannahme folgt

$$\sigma'' \vdash \textsf{while } b \textsf{ do } c \Rightarrow \sigma'$$

Also haben wir $\sigma \vdash \textsf{while } b \textsf{ do } c \Rightarrow \sigma'$ mit der zweiten Regel für Schleifen.

□

Dieser Beweis kommt ohne die Fixpunkteigenschaft aus. Wir benötigen lediglich die Konstruktion des kleinsten Fixpunktes.

Aus der Übereinstimmung der Semantiken bekommen wir sofort eine wichtige Eigenschaft der operationalen Semantik.

Korollar 5.4.3 *Für jedes Kommando $c \in Com$ und jeden Zustand $\sigma \in \Sigma$ gibt es höchstes einen Zustand $\sigma' \in \Sigma$ mit $\sigma \vdash c \Rightarrow \sigma'$.*

Kapitel 6

Formeln mit Quantoren

Wir betrachten jetzt eine logische Sprache ASSN, mit deren Formeln man Aussagen über ganze Zahlen formulieren kann. Hier ist ein Beispiel:

$$\forall x \in \mathbb{Z}\ \forall y \in \mathbb{Z}\ \exists z \in \mathbb{Z}\colon\ x + z = y$$

Die wichtigsten Konzepte, die wir am Beispiel von ASSN kennenlernen, sind Quantifizierung, Abkürzungen, Expressivität und Substitution. Außerdem führen wir die in der Logik gebräuchliche Unterscheidung zwischen Objekt- und Metaebene ein. Schließlich definieren wir die in der Logik gebräuchlichen $\models$-Notationen und Sprechweisen ein. ASSN ist ein Vertreter der Familie der prädikatenlogischen Sprachen, die wir später genauer betrachten werden.

Lesematerial

[Winskel, Kapitel 6]

6.1 Definition von ASSN

Abbildung 6.1 zeigt die Definition von ASSN. Die Definition erfolgt relativ zu einer Menge *Loc*, deren Elemente wir im Kontext von ASSN als Variablen bezeichnen. Die Syntax unterscheidet zwischen Termen und Formeln, die den arithmetischen und booleschen Ausdrücken von IMP entsprechen. Neu ist die existenzielle Quantifizierung. Die Belegungen von ASSN entsprechen den Zuständen von IMP. Die Semantik von Termen und Formel wird denotational definiert.

Die Formeln von ASSN sind alte Bekannte. Sie formalisieren Notationen, die wir routinemäßig benutzen, wenn wir mathematische Aussagen aufschreiben. Neu ist

$$
\begin{array}{lll}
n & \in & \mathit{Kon} = \mathbb{Z} & \textit{Konstante} \\
X, Y, Z & \in & \mathit{Loc} & \textit{Variable} \\
a & \in & \mathit{Ter} = & \textit{Term} \\
 & & \quad n & \quad \textit{Konstante} \\
 & & \mid X & \quad \textit{Variable} \\
 & & \mid a_1 + a_2 & \quad \textit{Summe} \\
 & & \mid a_1 * a_2 & \quad \textit{Produkt} \\
A, B, C & \in & \mathit{Assn} = & \textit{Formel} \\
 & & \quad a_1 \leq a_2 & \quad \textit{Vergleich} \\
 & & \mid A_1 \wedge A_2 & \quad \textit{Konjunktion} \\
 & & \mid \neg A & \quad \textit{Negation} \\
 & & \mid \exists X\, A & \quad \textit{existenzielle Quantifizierung}
\end{array}
$$

$$
\begin{array}{llll}
\sigma & \in & \Sigma = \mathit{Loc} \to \mathbb{Z} & \textit{Belegung} \\
 & & \mathbb{B} = \{0, 1\} & \textit{boolescher Wert}
\end{array}
$$

$\mathcal{T} \in \mathit{Ter} \to \Sigma \to \mathbb{Z}$

$$
\begin{aligned}
\mathcal{T}[\![n]\!]\sigma &= n \\
\mathcal{T}[\![X]\!]\sigma &= \sigma X \\
\mathcal{T}[\![a_1 + a_2]\!]\sigma &= \mathcal{T}[\![a_1]\!]\sigma + \mathcal{T}[\![a_2]\!]\sigma \\
\mathcal{T}[\![a_1 * a_2]\!]\sigma &= \mathcal{T}[\![a_1]\!]\sigma \cdot \mathcal{T}[\![a_2]\!]\sigma
\end{aligned}
$$

$\mathcal{D} \in \mathit{Assn} \to \Sigma \to \mathbb{B}$

$$
\begin{aligned}
\mathcal{D}[\![a_1 \leq a_2]\!]\sigma &= \text{if } \mathcal{T}[\![a_1]\!]\sigma \leq \mathcal{T}[\![a_2]\!]\sigma \text{ then } 1 \text{ else } 0 \\
\mathcal{D}[\![A_1 \wedge A_2]\!]\sigma &= \min\{\mathcal{D}[\![A_1]\!]\sigma,\ \mathcal{D}[\![A_2]\!]\sigma\} \\
\mathcal{D}[\![\neg A]\!]\sigma &= 1 - \mathcal{D}[\![A]\!]\sigma \\
\mathcal{D}[\![\exists X\, A]\!]\sigma &= \max\{\, \mathcal{D}[\![A]\!](\sigma[n/X]) \mid n \in \mathbb{Z}\,\}
\end{aligned}
$$

Abbildung 6.1: Syntax und Semantik der Sprache ASSN

$$\begin{array}{rcl}
-a & \mapsto & -1 * a \\
\mathsf{true} & \mapsto & 0 \leq 1 \\
\mathsf{false} & \mapsto & 1 \leq 0 \\
a_1 = a_2 & \mapsto & a_1 \leq a_2 \wedge a_2 \leq a_1 \\
a_1 < a_2 & \mapsto & a_1 \leq a_2 \wedge \neg(a_1 = a_2) \\
A_1 \vee A_2 & \mapsto & \neg(\neg A_1 \wedge \neg A_2) \\
A_1 \Rightarrow A_2 & \mapsto & \neg A_1 \vee A_2 \\
A_1 \iff A_2 & \mapsto & A_1 \Rightarrow A_2 \wedge A_2 \Rightarrow A_1 \\
\forall X A & \mapsto & \neg\exists X(\neg A)
\end{array}$$

Abbildung 6.2: Abkürzungen für ASSN

nur, dass wir diese Notationen jetzt als logische Sprache formalisiert haben. In mathematischer Notation würden wir statt

$$\exists X(5 + X = 3)$$

allerdings

$$\exists x \in \mathbb{Z}:\ 5 + x = 3$$

schreiben, um den Wertebereich der quantifizierten Variable x anzugeben. Bei den Formeln von ASSN ist dies nicht erforderlich, da der Wertebereich $\mathbb{Z}$ in ASSN eingebaut ist.

Mit ASSN können wir mathematische Aussagen als Formeln formulieren. Man spricht von einer Formulierung auf *Objektebene*. Die direkte Formulierung von mathematischen Aussagen mit den üblichen mathematischen Notationen bezeichnet man dagegen als Formulierung auf der *Metaebene*.

6.2 Abkürzungen und Expressivität

Abbildung 6.2 zeigt einige Abkürzungen für ASSN. Diese Abkürzungen werden für die konkrete Syntax von ASSN vereinbart. Wenn wir beispielsweise die Notation

$$\forall X(\forall Y(\exists Z(X + Z = Y)))$$

als eine Formel von ASSN interpretieren, meinen wir damit stets die Formel

$$\neg\exists X(\neg\neg\exists Y(\neg\exists Z(X + Z = Y)))$$

Alternativ können wir die Abkürzungen in Abbildung 6.2 auch direkt als Terme und Formeln von ASSN definieren. Das hat aber den Nachteil, dass alle auf ASSN aufbauenden Definitionen und Beweise komplizierter werden, da sie die zusätzlichen Varianten behandeln müssen. Die zusätzlichen Varianten bringen aber für unsere Zwecke nichts Neues, da sich ihre Denotationen entsprechend den Abkürzungsregeln auch mit den Grundvarianten ausdrücken lassen.

Betrachten Sie die Menge

$$DA \stackrel{\text{def}}{=} \{\, \mathcal{D}[\![A]\!] \mid A \in \mathit{Assn} \,\}$$

die aus allen Funktionen in $\Sigma \to \mathbb{B}$ besteht, die sich mit Formeln aus *Assn* beschreiben lassen. Aus mathematischer Sicht sind die Elemente von *DA* das Wesentliche und die Formeln in *Assn* sind lediglich eine Hilfskonstruktion, mit der sich die Elemente von *DA* beschreiben lassen. Wenn die Elimination einer syntaktischen Variante nicht dazu führt, dass *DA* kleiner wird, ist diese Variante aus mathematischer Sicht redundant. Die syntaktischen Varianten von ASSN (siehe Abbildung 6.1) sind in dem Sinne minimal gewählt, dass das Weglassen einer Variante stets dazu führt, dass *DA* echt kleiner wird. Die Menge *DA* kann man als *Expressivität* der Sprache ASSN bezeichnen. Weglassen von syntaktischen Varianten kann also die Expressivität verkleinern, Hinzufügen kann sie vergrößern (gemäß der Inklusionsordnung auf $\mathcal{P}(\Sigma \to \mathbb{B})$).

6.3 Notationen und Sprechweisen

Für die Formeln von ASSN verwenden wir die folgenden Notationen und Sprechweisen:

$$\begin{array}{rclcl}
\sigma \models A & \stackrel{\text{def}}{\Longleftrightarrow} & \sigma \textit{ erfüllt } A & \stackrel{\text{def}}{\Longleftrightarrow} & \mathcal{D}[\![A]\!]\sigma = 1 \\
\models A & \stackrel{\text{def}}{\Longleftrightarrow} & A \textit{ gültig} & \stackrel{\text{def}}{\Longleftrightarrow} & \forall \sigma \in \Sigma\colon\; \sigma \models A \\
A \models B & \stackrel{\text{def}}{\Longleftrightarrow} & A \textit{ stärker als } B & \stackrel{\text{def}}{\Longleftrightarrow} & \forall \sigma \in \Sigma\colon\; \sigma \models A \Rightarrow \sigma \models B \\
A \models\!\!\dashv B & \stackrel{\text{def}}{\Longleftrightarrow} & A \textit{ äquivalent } B & \stackrel{\text{def}}{\Longleftrightarrow} & \mathcal{D}[\![A]\!] = \mathcal{D}[\![B]\!]
\end{array}$$

Machen Sie sich klar, dass $\models$ eine reflexive und transitive Relation auf *Assn* ist, und dass $\models\!\!\dashv$ eine Äquivalenzrelation auf *Assn* ist. Offensichtlich gilt für beliebige $A, B \in$ ASSN:

$$A \models B \iff\; \models A \Rightarrow B \iff \forall \sigma \in \Sigma\colon\; \mathcal{D}[\![A]\!]\sigma \leq \mathcal{D}[\![B]\!]\sigma$$

Sei $A \in \mathit{Assn}$ und sei α die A entsprechende Aussage auf der Metaebene (also in normaler mathematischer Notation). Wir haben die Semantik von ASSN gerade so definiert, dass α genau dann gilt, wenn $\models A$ gilt. Die Semantik von ASSN formalisiert also das Offensichtliche.

Angesichts dieser Tatsache werden Sie sich vielleicht fragen, warum wir überhaupt den Aufwand treiben, ASSN formal als logische Sprache zu definieren. Ein Grund dafür ist, dass wir in den nächsten Kapiteln einige wichtige Eigenschaften von ASSN zeigen, die alles andere als intuitiv klar sind. Für diese Beweise benötigen wir eine solide Grundlage. Außerdem ist es so, dass der Vorrat an mathematischen Notationen sehr groß ist, und gerade für den Anfänger nur bis zu einem gewissen Grad präzise definiert ist. Dagegen haben wir mit ASSN eine kleine Sprache zur Verfügung, deren Syntax und Semantik streng formal bis ins letzte Detail definiert sind.

Die Metavariable V wird im Folgenden für eine Menge von Formeln stehen:

$$V \in \mathcal{P}(Assn)$$

Wir werden die folgenden Notationen und Sprechweisen benutzen:

$$\begin{array}{rcl@{\quad}c@{\quad}l}
\sigma \models V & \overset{\text{def}}{\iff} & \sigma \textit{ erfüllt } V & \overset{\text{def}}{\iff} & \forall A \in V:\ \sigma \models A \\
\models V & \overset{\text{def}}{\iff} & V \textit{ gültig} & \overset{\text{def}}{\iff} & \forall \sigma \in \Sigma:\ \sigma \models V \\
V_1 \models V_2 & \overset{\text{def}}{\iff} & V_1 \textit{ stärker als } V_2 & \overset{\text{def}}{\iff} & \forall \sigma \in \Sigma:\ \sigma \models V_1 \Rightarrow \sigma \models V_2 \\
V_1 \models\!\!\dashv V_2 & \overset{\text{def}}{\iff} & V_1 \textit{ äquivalent } V_2 & \overset{\text{def}}{\iff} & V_1 \models V_2 \wedge V_1 \models V_2
\end{array}$$

Machen Sie sich klar, dass $\models$ eine reflexive und transitive Relation auf $\mathcal{P}(Assn)$ ist, und dass $\models\!\!\dashv$ eine Äquivalenzrelation auf $\mathcal{P}(Assn)$ ist.

6.4 Bindungsstruktur

Quantifizierung in Formeln führt eine Bindungsstruktur zwischen Variablen ein, die wir bereits am Beispiel von Programmiersprachen kennengelernt haben. Lesen Sie dazu [Programmierung, Kapitel 11.2]. Man unterscheidet zwischen *definierenden* und *benutzenden* Auftreten von Variablen. In ASSN werden definierende Auftreten durch eine existenzielle Quantifizierung

$$\exists X\, A$$

eingeführt. Das Auftreten von X nach dem Existenzquantor $\exists$ ist definierend und bindet alle freie Auftreten von X in A. Freie Auftreten sind benutzende Auftreten, die noch durch keine Quantifizierung gebunden sind.

Die Bindungsstruktur einer Formel kann man durch Annotationen explizit machen. Zum Beispiel können wir die Bindungsstruktur der Formel

$$\exists X(X \leq Y \land \exists Y(X \leq Y \land \exists X(X \leq Y)))$$

wie folgt explizieren:

$$\exists \overline{X}_1(X_1 \leq Y \land \exists \overline{Y_1}(X_1 \leq Y_1 \land \exists \overline{X_2}(X_2 \leq Y_1)))$$

Dabei werden alle definierende Auftreten von Variablen überstrichen. Zusätzlich wird jedes definierende Auftreten einer Variablen mit einem für diese Variable eindeutigen Index versehen. Alle benutzenden Auftreten, die durch ein definierendes Auftreten gebunden sind, werden mit dem Index dieses definierenden Auftretens annotiert. Freie Variablenauftreten erkennt man nach dieser Annotation daran, dass sie keinen Index tragen. In der obigen Formel gibt es also genau ein freies Auftreten einer Variablen. Dabei handelt es sich um das erste Auftreten der Variablen Y.

Syntaktische Varianten, die ein definierendes Auftreten einer Variable einführen, heißen *Variablenbinder*. ASSN hat genau einen Variablenbinder, nähmlich existenzielle Quantifizierung. Programmiersprachen haben in der Regel viele Variablenbinder. IMP ist eine triviale Programmiersprache, die keinen Variablenbinder hat.

Durch strukturelle Rekursion kann man eine Funktion

$$VT \in Ter \to \mathcal{P}_{fin}(Loc)$$

definieren, die zu jedem Term a die Menge aller Variablen liefert, die in a auftreten (vergleiche die Definition von V in Kapitel 3.2). Mit der gleichen Vorgehensweise kann man eine Funktion

$$VA \in Assn \to \mathcal{P}_{fin}(Loc)$$

definieren, die zu jeder Formel A die Menge aller Variablen liefert, die in A auftreten (gebunden oder ungebunden). Wir definieren jetzt eine Funktion FV, die zu jeder Formel A die Menge aller Variablen liefert, die in A frei auftreten:

$$FV \in Assn \to \mathcal{P}_{fin}(Loc)$$

$$\begin{aligned} FV(a_1 \leq a_2) &= VT(a_1) \cup VT(a_2) \\ FV(A_1 \land A_2) &= FV(A_1) \cup FV(A_2) \\ FV(\neg A) &= FV(A) \\ FV(\exists X A) &= FV(A) - \{X\} \end{aligned}$$

Schließlich definieren wir eine Funktion BV, die zu jeder Formel A die Menge aller Variablen liefert, die in A ein definierendes Auftreten haben:

$$BV \in Assn \to \mathcal{P}_{fin}(Loc)$$

$$\begin{aligned} BV(a_1 \leq a_2) &= \emptyset \\ BV(A_1 \wedge A_2) &= BV(A_1) \cup BV(A_2) \\ BV(\neg A) &= BV(A) \\ BV(\exists X A) &= BV(A) \cup \{X\} \end{aligned}$$

Eine Formel A heißt *bereinigt* genau dann, wenn die folgenden Bedingungen erfüllt sind:

1. Keine Variable hat in A mehr als ein definierendes Auftreten.
2. Keine Variable hat in A sowohl ein definierendes als auch ein freies Auftreten.

6.5 Substitution

Sei $A \in Assn$ eine Formel, $a \in Ter$ ein Term und $X \in Loc$ eine Variable. Dann bezeichnen wir mit

$$A[a/X]$$

die Formel, die wir aus A erhalten, indem wir alle freien Auftreten der Variable X durch den Ausdruck a ersetzen (wir sagen auch *substituieren*). Hier ist ein Beispiel:

$$\begin{aligned} A = {} & X * Y \leq Y \\ A[Y-1/Y] = {} & X * (Y-1) \leq (Y-1) \end{aligned}$$

Für Sprachen ohne Variablenbinder ist Substitution ein sehr einfaches Konzept. Für Sprachen mit Variablenbindern ist Substitution jedoch etwas komplizierter, da dann das Problem des Kaperns auftritt. Betrachten Sie dazu die Formel

$$A = \exists Y (Y + 1 \leq X)$$

und nehmen Sie an, wir wollen alle freien Auftreten von X durch Y ersetzen, also $A[Y/X]$ bestimmen. Wenn wir naiv vorgehen, bekommen wir die Formel

$$A = \exists Y (Y + 1 \leq Y)$$

Diese Substitution ist jedoch unzulässig, da das von außen kommende Auftreten von Y durch den Quantor der Formel *gekapert* wird. Der Grund, dass man Substitution ohne Kapern definieren will, liegt darin, dass für Substitution aus prinzipiellen Gründen immer die sogenannte *Substitutionseigenschaft* gelten soll:

$$\begin{array}{l} \forall A \in Assn\ \forall a \in Ter\ \forall X \in Loc\ \forall \sigma \in \Sigma: \\ \qquad \mathcal{D}[\![A[a/X]]\!]\sigma = \mathcal{D}[\![A]\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X]) \end{array}$$

Für das obige unzulässige Beispiel ist die Substitutionseigenschaft verletzt, da für alle $\sigma \in \Sigma$:

$$\mathcal{D}[\![\exists Y(Y+1 \leq Y)]\!]\sigma = 0 \neq 1 = \mathcal{D}[\![\exists Y(Y+1 \leq X)]\!](\sigma[\mathcal{T}[\![Y]\!]\sigma/X])$$

Wir definieren Substitution zuerst mittels struktureller Rekursion für Terme. Das ist einfach, da hier das Problem des Kaperns nicht auftritt.

$$\begin{array}{rcl} n[a/X] & = & n \\ Y[a/X] & = & \text{if } Y = X \text{ then } a \text{ else } Y \\ (a_1 + a_2)[a/X] & = & a_1[a/X] + a_2[a/X] \\ (a_1 \times a_2)[a/X] & = & a_1[a/X] \times a_2[a/X] \end{array}$$

Proposition 6.5.1 *Seien $a, a' \in Ter$, $X \in Loc$ und $\sigma \in \Sigma$. Dann:*

$$\mathcal{T}[\![a[a'/X]]\!]\sigma = \mathcal{T}[\![a]\!](\sigma[\mathcal{T}[\![a']\!]\sigma/X])$$

Beweis Durch strukturelle Induktion über $a \in Ter$. Übung! □

Jetzt definieren wir mittels struktureller Rekursion eine vorläufige Substitutionsfunktion für Formeln wie folgt:

$$S \in Assn \times Ter \times Loc \rightarrow Assn$$

$$\begin{array}{rcl} S(a_1 \leq a_2, a, X) & = & a_1[a/X] \leq a_2[a/X] \\ S(A_1 \wedge A_2, a, X) & = & S(A_1, a, X) \wedge S(A_2, a, X) \\ S(\neg A, a, X) & = & \neg S(A, a, X) \\ S(\exists Y A, a, X) & = & \text{if } Y = X \text{ then } \exists Y A \text{ else } \exists Y (S(A, a, X)) \end{array}$$

Proposition 6.5.2 *Seien $A \in Assn$, $a \in Ter$, $X \in Loc$ und $\sigma \in \Sigma$. Dann:*

$$BV(A) \cap VT(a) = \emptyset \;\Rightarrow\; \mathcal{D}[\![S(A, a, X)]\!]\sigma = \mathcal{D}[\![A]\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X])$$

Beweis Durch strukturelle Induktion über $A \in Assn$. Wir zeigen nur den Beweisteil für existenzielle Quantifizierung.

Sei $A = \exists Y A'$ und $BV(A) \cap VT(a) = \emptyset$. Wir müssen

$$\mathcal{D}[\![S(A, a, X)]\!]\sigma = \mathcal{D}[\![A]\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X])$$

zeigen. Dazu unterscheiden wir zwei Fälle.

Sei $X = Y$. Dann:

$$\begin{array}{ll}
\mathcal{D}[\![S(A,a,X)]\!]\sigma & \\
= \mathcal{D}[\![A]\!]\sigma & \text{Def. von } S \\
= \text{if } \exists n \in \mathbb{Z}\colon\ \mathcal{D}[\![A']\!](\sigma[n/Y]) \text{ then } 1 \text{ else } 0 & \text{Def. von } \mathcal{D} \\
= \text{if } \exists n \in \mathbb{Z}\colon\ \mathcal{D}[\![A']\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X][n/Y]) \text{ then } 1 \text{ else } 0 & \text{da } X = Y \\
= \mathcal{D}[\![\exists Y\, A']\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X]) & \text{Def. von } \mathcal{D} \\
= \mathcal{D}[\![A]\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X]) &
\end{array}$$

Sei $X \neq Y$. Dann:

$$\begin{array}{ll}
\mathcal{D}[\![S(A,a,X)]\!]\sigma & \\
= \mathcal{D}[\![\exists Y(S(A',a,X))]\!]\sigma & \text{Def. von } S \\
= \text{if } \exists n \in \mathbb{Z}\colon\ \mathcal{D}[\![S(A',a,X)]\!](\sigma[n/Y]) \text{ then } 1 \text{ else } 0 & \text{Def. von } \mathcal{D} \\
= \text{if } \exists n \in \mathbb{Z}\colon\ \mathcal{D}[\![A']\!](\sigma[n/Y][\mathcal{T}[\![a]\!](\sigma[n/Y])/X]) \text{ then } 1 \text{ else } 0 & \text{Induktionsannahme} \\
= \text{if } \exists n \in \mathbb{Z}\colon\ \mathcal{D}[\![A']\!](\sigma[n/Y][\mathcal{T}[\![a]\!]\sigma/X]) \text{ then } 1 \text{ else } 0 & Y \notin VT(a) \\
= \text{if } \exists n \in \mathbb{Z}\colon\ \mathcal{D}[\![A']\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X][n/Y]) \text{ then } 1 \text{ else } 0 & X \neq Y \\
= \mathcal{D}[\![\exists Y\, A']\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X]) & \text{Def. von } \mathcal{D} \\
= \mathcal{D}[\![A]\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X]) & \square
\end{array}$$

Wir haben jetzt eine Substitutionsfunktion für Formeln, die unter gewissen Vorbedingungen die Substitionseigenschaft erfüllt. Aus technischen Gründen ist es aber wünschenswert, eine Substitutionsfunktion zu haben, für die die Substitutionseigenschaft immer gilt.

Das Problem läßt sich mittels konsistenter Umbenennung von gebundenen Variablen lösen. Beispielsweise ist

$$\exists Z(Z + 1 \leq Y)$$

eine konsistente Umbenennung von

$$\exists X(X + 1 \leq Y)$$

Wir werden zeigen, dass Formel, die bis auf konsistente Umbenennung von gebundenen Variablen gleich sind, die gleiche Denotation haben. Wenn wir also einer Substitutionsfunktion erlauben, bei Bedarf gebundene Variablen konsistent umzubenennen, kann sie die Substitutioneigenschaft uneingeschränkt erfüllen. Allerdings müssen wir noch voraussetzen, dass es unendlich viele Variablen gibt.

Wir definieren jetzt eine Relation *CR* (consistent renaming)

$$A \sim A' \overset{\text{def}}{\iff} A \textit{ Variante} \text{ von } A' \overset{\text{def}}{\iff} \langle A, A'\rangle \in CR \subseteq \mathit{Assn} \times \mathit{Assn}$$

durch die folgenden Inferenzregeln:

$$\frac{S(A, Y, X) = A' \quad Y \notin VA(A)}{\exists X A \sim \exists Y A'}$$

$$\frac{A_1 \sim A_1' \quad A_2 \sim A_2'}{A_1 \wedge A_2 \sim A_1' \wedge A_2'} \qquad \frac{A \sim A'}{\neg A \sim \neg A'} \qquad \frac{A \sim A'}{\exists X A \sim \exists X A'}$$

$$\frac{}{A \sim A} \qquad \frac{A' \sim A}{A \sim A'} \qquad \frac{A \sim A'' \quad A'' \sim A'}{A \sim A'}$$

Die Relation „$A \sim A'$“ formalisiert unsere intuitive Idee von konsistenter Umbenennung in dem Sinne, dass $A \sim A'$ genau dann gilt, wenn A und A' bis auf konsistente Umbenennung von gebundenen Variablen gleich sind. Das gilt allerdings nur dann, wenn es unendlich viele Variablen gibt.

Proposition 6.5.3 *Die Relation „$A \sim A'$“ ist eine Äquivalenzrelation. Für alle A, $A' \in Assn$ gilt:*

$$A \sim A' \Rightarrow \mathcal{D}[\![A]\!] = \mathcal{D}[\![A']\!]$$

Beweis Die letzten drei Inferenzregeln erzwingen, dass „$A \sim A'$“ eine Äquivalenzrelation ist. Die zweite Behauptung folgt mit Regelinduktion und Proposition 6.5.2. □

Definition 6.5.4 *Eine* Substitutionsfunktion *für ASSN ist eine Funktion*

$$s \in Assn \times Ter \times Loc \to Assn$$

die die folgenden Bedingungen erfüllt:

$$\begin{aligned} &\forall A \in Assn\ \forall a \in Ter\ \forall X \in Loc\ \exists A' \in Assn: \\ &\qquad A \sim A' \wedge BV(A') \cap VT(a) = \emptyset \ \Rightarrow\ s(A, a, X) \sim S(A', a, X) \end{aligned}$$

Proposition 6.5.5 *Wenn Loc eine unendliche Menge ist, gibt es eine Substitutionsfunktion für ASSN. Jede Substitutionsfunktion s für ASSN erfüllt die Substitutionseigenschaft:*

$$\begin{aligned} &\forall A \in Assn\ \forall a \in Ter\ \forall X \in Loc\ \forall \sigma \in \Sigma: \\ &\qquad \mathcal{D}[\![s(A, a, X)]\!]\sigma = \mathcal{D}[\![A]\!](\sigma[\mathcal{T}[\![a]\!]\sigma/X]) \end{aligned}$$

Wir werden im Folgenden voraussetzen, dass wir eine Substitutionsfunktion $A[a/X]$ für ASSN festgelegt haben. Welche das ist, spielt keine Rolle. Beispiele können wir wie folgt schreiben:

$$(\exists Y(Y + 1 \leq X))[Y/X] \sim \exists Z(Z + 1 \leq Y)$$

Übrigens umgeht Winskel in seinem Buch das Kaper-Problem für Substitutionen dadurch, dass er mit zwei Arten von Variablen arbeitet. Quantifizieren darf man nur über die eine Art von Variable, und die zu substituierenden Terme dürfen nur die zweite Art von Variablen enthalten.

Kapitel 7

Programmverifikation

Eine *Spezifikation* sagt, *was* gemacht werden soll. Eine *Implementierung* sagt, *wie* etwas gemacht werden soll. Unter einer *Verifikation* versteht man eine Argumentation, die zeigt, dass eine Implementierung macht was eine Spezifikation vorschreibt.

Diese grundlegende Idee, die auf Hardware und Software angewendet werden kann, machen wir jetzt am Beispiel der Programmiersprache IMP konkret. Dabei sind Implementierungen stets Kommandos gemäß IMP. Spezifikationen formulieren wir mit den Formeln von ASSN. Gegeben IMP und ASSN können wir definieren, was es heißt, dass eine Implementierung (ein Kommando in IMP) eine Spezifikation (ein syntaktisches Objekt von ASSN) erfüllt. Damit haben wir eine Grundlage für Verifikationstechniken, mit denen wir beweisen können, dass eine Implementierung eine Spezifikation erfüllt.

Da wesentliche Ideen dieser Konstruktion von C.A.R. Hoare entwickelt wurden (um 1970), bezeichnet man sie auch als Hoare-Logik.

Lesematerial

[Winskel, Kapitel 6 und 7]

7.1 Spezifikationen und Korrektheit

Für IMP verwenden wir Spezifikationen, die aus einer *Vorbedingung* und einer *Nachbedingung* bestehen. Hier ist ein Beispiel:

Vorbedingung:	$X \geq 1 \land Y \geq 0$
Nachbedingung:	$Z = X + Y$

Sei das Kommando c ein Kandidat für die Implementierung dieser Spezifikation. Die Vorbedingung legt fest, auf welche Zustände c angewendet werden soll. Die Nachbedingung legt fest, welche Eigenschaften die Endzustände haben sollen, die die Ausführung von c mit einem Anfangszustand gemäß der Vorbedingung liefert. Wenn wir annehmen, dass c die Lokationen X und Y nicht ändert, legt die obige Spezifikation gerade fest, dass c die Summe $X + Y$ berechnen und in der Lokation Z ablegen soll.

Es ist zweckmäßig, zwischen Termination und partieller Korrektheit zu trennen. Partielle Korrektheit läßt zu, dass ein Kommando für zulässige Anfangszustände divergiert. Nur für den Fall, dass der Anfangszustand die Vorbedingung erfüllt und das Kommando für diesen Zustand terminiert, muss der gelieferte Endzustand die Nachbedingung erfüllen. Unter totaler Korrektheit versteht man partielle Korrektheit plus Termination für alle Anfangszustände, die die Vorbedingung erfüllen. Hoare-Logik behandelt partielle Korrektheit und ignoriert Termination.

Im Kontext von Verifikation bezeichnet man die Formeln von ASSN auch als *Zusicherungen* oder *Assertionen*. Für Zusicherungen werden wir die folgenden Metavariablen verwenden:

$$A, B, C, I \in Assn$$

Aus technischen Gründen ist es sinnvoll, darauf zu bestehen, dass die folgenden Beziehungen gelten:

1. $Aexp \subseteq Ter$ und $\forall a \in Ter\colon\ \mathcal{A}[\![a]\!] = \mathcal{T}[\![a]\!]$.
2. $Bexp \subseteq Assn$ und $\forall b \in Bexp\colon\ \mathcal{B}[\![b]\!] = \mathcal{D}[\![b]\!]$.

Bisher haben wir die Ausdrücke von IMP jedoch so definiert, dass die obigen Beziehungen nicht gelten. Dieses Problem lässt sich jedoch leicht beheben, indem wir die Syntax von IMP an ASSN anpassen:

$$\begin{aligned} a \in Aexp &= n \mid\ X \mid\ a_1 + a_2 \mid\ a_1 * a_2 \\ b \in Bexp &= a_1 \leq a_1 \mid\ A_1 \wedge A_2 \mid\ \neg A \end{aligned}$$

In der Praxis ist es sinnvoll, für Zusicherungen weitere arithmetische Operationen (zum Beispiel Exponentiation n^k) zur Verfügung zu stellen. Es ist einfach, ASSN entsprechend zu erweitern. Alle Ergebnisse dieses Kapitels gelten auch für entsprechende Erweiterungen von *ASSN*.

Eine *Spezifikation* ist ein Paar $\langle A, B\rangle$ aus zwei Zusicherungen $A, B \in Assn$.

Definition 7.1.1 *Ein Kommando $c \in Com$* erfüllt *eine Spezifikation $\langle A, B\rangle$ genau dann, wenn:*

$$\forall \sigma, \sigma' \in \Sigma\colon\ (\sigma \models A\ \wedge\ \mathcal{C}[\![c]\!]\sigma = \sigma')\ \Rightarrow\ \sigma' \models B$$

Wir führen eine Notation ein:

$$\models \{A\}c\{B\}\ \overset{\text{def}}{\iff}\ \{A\}c\{B\}\ \textit{ist gültig}\ \overset{\text{def}}{\iff}\ c\ \textit{erfüllt}\ \langle A, B\rangle$$

Die Tripel aus $Assn \times Com \times Assn$ nennen wir *Hoare-Tripel* und schreiben sie als $\{A\}c\{B\}$.

Um die Gültigkeit von Hoare-Tripeln kompakter charakterisieren zu können, definieren wir eine Relation auf $\Sigma_\perp \times Assn$ wie folgt:

$$\begin{aligned}&\forall s \in \Sigma_\perp\ \forall A \in Assn\colon\\ &\qquad s \models A\ \overset{\text{def}}{\iff}\ (s \neq \perp \Rightarrow \mathcal{D}[\![A]\!]s = 1)\end{aligned}$$

Diese Relation stimmt auf $\Sigma \times Assn$ mit der für *ASSN* eingeführten Relation „$\sigma \models A$“ überein. Jetzt können wir die Gültigkeit von Hoare-Tripeln wie folgt charakterisieren:

$$\begin{aligned}&\forall A \in Assn\ \forall c \in Com\ \forall B \in Assn\colon\\ &\qquad \models \{A\}c\{B\} \iff (\forall \sigma \in \Sigma\colon\ \sigma \models A \Rightarrow \mathcal{C}[\![c]\!]\sigma \models B)\end{aligned}$$

Wir können jetzt die Begriffe partielle und totale Korrektheit definieren. Ein Kommando c heißt *partiell korrekt* für eine Spezifikation $\langle A, B\rangle$ genau dann, wenn $\models \{A\}c\{B\}$. Ein Kommando c heißt *total korrekt* für eine Spezifikation $\langle A, B\rangle$ genau dann, wenn $\models \{A\}c\{B\}$ und

$$\forall \sigma \in \Sigma\colon\ \sigma \models A\ \Rightarrow\ \mathcal{C}[\![c]\!]\sigma \neq \perp$$

Expressivität von Spezifikationen

Sei c ein Kommando mit

$$\models \{\mathsf{true}\}c\{\mathsf{false}\}$$

Da die Nachbedingung false von keinem Zustand erfüllt wird, bedeutet dies, dass c für alle Zustände divergiert. Wir haben also

$$\forall c \in Com\colon\ (\forall \sigma \in \Sigma\colon\ \mathcal{C}[\![c]\!]\sigma = \perp) \iff\ \models \{\mathsf{true}\}c\{\mathsf{false}\}$$

Nehmen Sie jetzt für einen Augenblick an, dass die Menge *Loc* endlich ist. Dann können wir für jeden Zustand σ eine Zusicherung A_σ angeben, so dass σ der einzige Zustand ist, der A erfüllt. Sei beispielsweise $Loc = \{X, Y\}$ und

$$\sigma = \{X \mapsto 1,\ Y \mapsto 2\}$$

Dann können wir

$$A_\sigma = (X = 1 \land Y = 2)$$

wählen. Damit gilt für $c \in Com$ und $\sigma, \sigma' \in \Sigma$:

$$\mathcal{C}[\![c]\!]\sigma = \bot \iff \models \{A_\sigma\}c\{\mathsf{false}\}$$
$$\mathcal{C}[\![c]\!]\sigma = \sigma' \iff \models \{A_\sigma\}c\{A_{\sigma'}\} \ \land\ \neg \models \{A_\sigma\}c\{\mathsf{false}\}$$

Da Kommandos immer nur endlich viele Lokationen enthalten, bekommen wir diese Äquivalenzen auch bei unendlichem *Loc*. Allerdings müssen wir A_σ dann relativ zu der endlichen Menge von Lokationen konstruieren, die in c vorkommen.

7.2 Hoare-Regeln

Wir definieren jetzt mithilfe von Inferenzregeln, die als Hoare-Regeln bezeichnet werden, eine Menge von Hoare-Tripeln. Wir werden zeigen, dass diese Menge genau die Menge der gültigen Hoare-Tripel ist. An dieser Definition ist interessant, dass sie unabhängig von den bisherigen Semantiken für IMP erfogt. Da man mit der Menge der gültigen Hoare-Tripeln die Semantik von IMP beschreiben kann (siehe oben), kann man die Hoare-Regeln als eine weitere Semantik für IMP ansehen. Man spricht von einer *axiomatischen Semantik*.

Wir definieren also eine Menge *AS*

$$\vdash \{A\}c\{B\} \overset{\text{def}}{\iff} \langle A, c, B\rangle \in AS \subseteq Assn \times Com \times Assn$$

rekursiv durch die folgenden Inferenzregeln (Hoare-Regeln):

$$\frac{}{\vdash \{A\}\mathsf{skip}\{A\}} \qquad \frac{}{\vdash \{B[a/X]\}X := a\{B\}}$$

$$\frac{\vdash \{A\}c_1\{C\} \quad \vdash \{C\}c_2\{B\}}{\vdash \{A\}c_1; c_2\{B\}} \qquad \frac{\vdash \{A \land b\}c_1\{B\} \quad \vdash \{A \land \neg b\}c_2\{B\}}{\vdash \{A\}\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\{B\}}$$

$$\frac{\vdash \{I \land b\}c\{I\}}{\vdash \{I\}\mathsf{while}\ b\ \mathsf{do}\ c\{I \land \neg b\}} \qquad \frac{A \models A' \quad \vdash \{A'\}c\{B'\} \quad B' \models B}{\vdash \{A\}c\{B\}}$$

Diese Definition erfolgt nach Schema (3) in Abschnitt 4.6. Da wir AS als Teilmenge der Grundmenge $Assn \times Com \times Assn$ definieren, verzichten wir auf Prämissen der Bauart $A \in Assn$.

Die Zusicherung I in der Regel für Schleifen nennt man *Invariante*. Die zuletzt angegebene Hoare-Regel heißt *Abschwächungsregel*.

Satz 7.2.1 (Korrektheit der Hoare-Regeln)

$$\forall c \in Com\ \forall A, B \in Assn:\ \ \vdash \{A\}c\{B\} \ \Rightarrow \models \{A\}c\{B\}$$

Beweis Wir zeigen

$$\begin{aligned}&\forall c \in Com\ \forall A, B \in Assn:\\ &\qquad \vdash \{A\}c\{B\} \ \Rightarrow\ (\forall \sigma, \sigma' \in \Sigma:\ \sigma \models A \ \wedge\ \mathcal{C}[\![c]\!]\sigma = \sigma' \ \Rightarrow\ \sigma' \models B)\end{aligned}$$

durch Regelinduktion über $\vdash$.

1. *Regel für Sequentialisierung*. Sei $\models \{A\}c_1\{C\}$ und $\models \{C\}c_2\{B\}$ (Induktionsannahmen). Weiter sei $\mathcal{C}[\![c_1; c_2]\!]\sigma = \sigma'$ und $\sigma \models A$. Wir müssen $\sigma' \models B$ zeigen. Mit der Definition von $\mathcal{C}$ haben wir ein σ'' mit $\mathcal{C}[\![c_1]\!]\sigma = \sigma''$ und $\mathcal{C}[\![c_2]\!]\sigma'' = \sigma'$. Mit der ersten Induktionsannahme folgt $\sigma'' \models C$. Mit der zweiten Induktionsannahme folgt $\sigma' \models B$.

2. *Regel für Schleifen*. Sei $\models \{I \wedge b\}c\{I\}$ (Induktionsannahme). Weiter sei $\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma = \sigma'$ und $\sigma \models I$. Wir müssen $\sigma' \models I \wedge \neg b$ zeigen. Nach Definition von $\mathcal{C}$ genügt es zu zeigen:

   $$\begin{aligned}&\forall n \in \mathbb{N}\ \forall \sigma \in \Sigma:\\ &\qquad \sigma' = (\Gamma(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]))^n(\lambda\sigma\,.\,\bot))\ \sigma\ \wedge\ \sigma \models I\ \Rightarrow\ \sigma' \models I \wedge \neg b\end{aligned}$$

   Dies Behauptung zeigt man leicht durch Induktion über $n \in \mathbb{N}$. Übung!

3. Die Beweisteile für die restlichen Regeln sind leicht. Übung! □

7.3 Verifikationsbedingungen

Wir entwickeln jetzt eine Methode, mit der man Kommandos verifizieren kann. Wir geben einen Algorithmus an, der zu einem Kommando c und einer Spezifikation $\langle A, B\rangle$ eine endliche Menge V von Zusicherungen berechnet, so dass

$$\models V \ \Rightarrow \vdash \{A\}c\{B\}$$

gilt. Die Zusicherungen in V nennt man *Verifikationsbedingungen*. Den Algorithmus nennt man *Verifikationsbedingungsgenerator*. Der Algorithmus funktioniert

$$
\begin{array}{lcl}
\multicolumn{3}{l}{VC' \in Com \times Assn \to Assn \times \mathcal{P}_{fin}(Assn)} \\
VC'(\textsf{skip},\ B) & = & \langle B, \emptyset \rangle \\
VC'(X := a,\ B) & = & \langle B[a/X], \emptyset \rangle \\
VC'(c_1; c_2,\ B) & = & \text{let } \langle A_2, V_2 \rangle = VC'(c_2, B) \\
 & & \quad\ \langle A_1, V_1 \rangle = VC'(c_1, A_2) \\
 & & \text{in } \langle A_1, V_1 \cup V_2 \rangle \\
VC'(\textsf{if } b \textsf{ then } c_1 \textsf{ else } c_2,\ B) & = & \text{let } \langle A_1, V_1 \rangle = VC'(c_1, B) \\
 & & \quad\ \langle A_2, V_2 \rangle = VC'(c_2, B) \\
 & & \text{in } \langle b \wedge A_1 \vee \neg b \wedge A_2,\ V_1 \cup V_2 \rangle \\
VC'(\{I\}\textsf{while } b \textsf{ do } c,\ B) & = & \text{let } \langle A, V \rangle = VC'(c, I) \\
 & & \text{in } \langle I,\ V \cup \{I \wedge b \Rightarrow A,\ I \wedge \neg b \Rightarrow B\} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{VC \in Assn \times Com \times Assn \to \mathcal{P}_{fin}(Assn)} \\
VC(A, c, B) & = & \text{let } \langle A', V \rangle = VC'(c, B) \text{ in } V \cup \{A \Rightarrow A'\}
\end{array}
$$

Abbildung 7.1: Verifikationsbedingungsgenerator für IMP

allerdings nur, wenn man jede Schleife von Hand mit einer Invariante annotiert. Annotierte Schleifen schreiben wir als

$$\{I\}\textsf{while } b \textsf{ do } c$$

Zu einem Kommando c mit n Schleifen berechnet der Algorithmus $2n + 1$ Verifikationsbedingungen. Für jede Schleife bekommt man genau zwei Verifikationsbedingungen.

Sei *ComA* die Menge der annotierten Kommandos. Um unsere Notation einfach zu halten, benutzen wir die Metavariable c sowohl für einfache wie für annotierte Kommandos. Zudem lassen wir annotierte Kommandos auch überall da zu, wo einfache Kommandos verlangt werden. Dabei werden die annotierten Invarianten automatisch gelöscht. Dieser notationale Trick spart uns viel Schreibarbeit.

Abbildung 7.1 zeigt den Verifikationsbedingungsgenerator für IMP und *Assn*.

Satz 7.3.1 (Korrektheit von VC') *Sei* $VC'(c, B) = \langle A, V \rangle$ *und* $\models V$. *Dann* $\vdash \{A\}c\{B\}$.

Beweis Durch strukturelle Induktion über $c \in Com$. Sei $VC'(c, B) = \langle A, V \rangle$ und $\models V$. Wir geben nur die Beweisteile für Konditionale und Schleifen an.

1. Sei $c = \textsf{if } b \textsf{ then } c_1 \textsf{ else } c_2$. Sei $VC'(c_1, B) = \langle A_1, V_1 \rangle$ und

$VC'(c_2, B) = \langle A_2, V_2\rangle$. Dann $A = (b \wedge A_1 \vee \neg b \wedge A_2)$ und $V = V_1 \cup V_2$ (Definition von VC'). Also:

$\vdash \{A_1\}c_1\{B\}$	Induktionsannahme
$\vdash \{A_2\}c_2\{B\}$	Induktionsannahme
$\vdash \{A \wedge b\}c_1\{B\}$	Abschwächungsregel
$\vdash \{A \wedge \neg b\}c_2\{B\}$	Abschwächungsregel
$\vdash \{A\}c\{B\}$	Regel für Konditionale

2. Sei $c = \{I\}\textsf{while}\ b\ \textsf{do}\ c'$. Sei $VC'(c', I) = \langle A', V'\rangle$. Dann $A = I$ und $V = V' \cup \{I \wedge b \Rightarrow A',\ I \wedge \neg b \Rightarrow B\}$ (Definition von VC'). Also:

$\vdash \{A'\}c'\{I\}$	Induktionsannahme
$\vdash \{I \wedge b\}c'\{I\}$	Abschwächungsregel
$\vdash \{I\}c\{I \wedge \neg b\}$	Schleifenregel
$\vdash \{A\}c\{B\}$	Abschwächungsregel

□

Korollar 7.3.2 (Korrektheit von VC) *Sei* $\models VC(A, c, B)$. *Dann* $\vdash \{A\}c\{B\}$.

Beweis Satz 7.3.1. □

7.4 VC-Methode

Wir können jetzt die Gültigkeit eines Hoare-Tripels wie folgt zeigen:

1. Annotiere alle Schleifen des Kommandos mit Invarianten.
2. Zeige, dass alle Verifikationsbedingungen gültig sind.

Wir bezeichnen dieses Vorgehen als *VC-Methode*. Als Beispiel verifizieren wir das Hoare-Tripel:

$$\begin{array}{ll} A = & \{Y \geq 0\} \\ & N := Y; \\ & Z := 1; \\ & \textsf{while}\ N \geq 1\ \textsf{do}\ (N; = N - 1;\ \ Z := Z * X) \\ B = & \{Z = X^Y\} \end{array}$$

Für dieses Beispiel haben wir die Terme von ASSN um Exponentiation erweitert. Als Invariante wählen wir:

$$I = (X^Y = Z * X^N \wedge N \geq 0)$$

Jetzt zeigen wir die Gültigkeit der drei Verifikationsbedingungen:

1. $I \wedge \neg(N \geq 1) \models B$. Gilt, da:

$$I \wedge \neg(N \geq 1) \;\;\models\!\!\!\dashv\;\; X^Y = Z * X^N \wedge N = 0 \;\models\; B$$

2. $I \wedge N \geq 1 \models I[Z * X/Z][N-1/N]$. Gilt, da:

$$\begin{aligned} I[Z * X/Z][N-1/N] &\models\!\!\!\dashv (X^Y = (Z * X) * X^N \wedge N \geq 0)[N-1/N] \\ &\models\!\!\!\dashv X^Y = (Z * X) * X^{N-1} \wedge N - 1 \geq 0 \\ &\models\!\!\!\dashv X^Y = Z * X^N \wedge N \geq 1 \end{aligned}$$

3. $A \models I[1/Z][Y/N]$. Gilt, da:

$$\begin{aligned} I[1/Z][Y/N] &\models\!\!\!\dashv (X^Y = 1 * X^N \wedge N \geq 0)[Y/N] \\ &\models\!\!\!\dashv X^Y = 1 * X^Y \wedge Y \geq 0 \\ &\models\!\!\!\dashv Y \geq 0 \end{aligned}$$

7.5 Vollständigkeit der VC-Methode

Satz 7.5.1 (Monotonie von VC') *Sei* $VC'(c, B_1) = \langle A_1, V_1 \rangle$, $VC'(c, B_2) = \langle A_2, V_2 \rangle$, *und* $B_1 \models B_2$. *Dann* $A_1 \models A_2$ *und* $V_1 \models V_2$.

Beweis Durch strukturelle Induktion über $c \in Com$. Sei $VC'(c, B_1) = \langle A_1, V_1 \rangle$, $VC'(c, B_2) = \langle A_2, V_2 \rangle$, und $B_1 \models B_2$. Wir geben nur die Beweisteile für Konditionale und Schleifen an.

1. Sei $c = \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. Sei

$$\begin{aligned} (A_{11}, V_{11}) &= VC'(c_1, B_1) \\ (A_{12}, V_{12}) &= VC'(c_1, B_2) \\ (A_{21}, V_{21}) &= VC'(c_2, B_1) \\ (A_{22}, V_{22}) &= VC'(c_2, B_2) \end{aligned}$$

Nach Induktionsannahme gilt $A_{11} \models A_{12}$ und $V_{11} \models V_{12}$ sowie $A_{21} \models A_{22}$ und $V_{21} \models V_{22}$. Außerdem liefert die Definition von VC':

$$\begin{aligned} (A_1, V_1) &= (b \wedge A_{11} \vee \neg b \wedge A_{21},\;\; V_{11} \cup V_{21}) \\ (A_2, V_2) &= (b \wedge A_{12} \vee \neg b \wedge A_{22},\;\; V_{12} \cup V_{22}) \end{aligned}$$

Also $A_1 \models A_2$ und $V_1 \models V_2$.

2. Sei $c = \{I\}\mathsf{while}\ b\ \mathsf{do}\ c'$. Sei $VC'(c', I) = \langle A, V\rangle$. Die Definition von VC' liefert:

$$(A_1, V_1) = (I,\ V \cup \{I \wedge b \Rightarrow A,\ I \wedge \neg b \Rightarrow B_1\})$$
$$(A_2, V_2) = (I,\ V \cup \{I \wedge b \Rightarrow A,\ I \wedge \neg b \Rightarrow B_2\})$$

Also $A_1 \models A_2$ und $V_1 \models V_2$.

□

Satz 7.5.2 (Vollständigkeit von VC') *Sei* $\vdash \{A\}c\{B\}$. *Dann kann man c so mit Invarianten notieren, dass gilt:*

$$\forall A'\ \forall V:\ VC'(c, B) = \langle A', V\rangle \ \Rightarrow\ (A \models A' \wedge \models V)$$

Beweis Durch Regelinduktion über die Hoare-Regeln. Wir geben nur die Beweisteile für die Regeln für Sequentialisierung, Schleifen und Abschwächung an. Dabei benötigen wir die gerade gezeigte Monotonie-Eigenschaft von VC'.

1. Regel für $c_1; c_2$. Sei $\vdash \{A\}c_1\{C\}$ und $\vdash \{C\}c_2\{B\}$. Wir annotieren c_1 und c_2 gemäß der Induktionsannahme. Sei $VC'(c_1, C) = \langle A_1, V_1\rangle$ und $VC'(c_2, B) = \langle A_2, V_2\rangle$. Nach Induktionsannahme gilt $\models V_1 \cup V_2$ sowie $A \models A_1$ und $C \models A_2$. Sei $VC'(c_1, A_2) = \langle A_1', V_1'\rangle$. Dann

   $$VC'(c_1; c_2, B) = \langle A_1', V_1' \cup V_2\rangle$$

   Nach Satz 7.5.1 gilt $A_1 \models A_1'$ und $V_1 \models V_1'$. Damit folgt $A \models A_1'$ und $\models V_1' \cup V_2$.

2. Regel für $\mathsf{while}\ b\ \mathsf{do}\ c$. Sei $\vdash \{I \wedge b\}c\{I\}$. Wir annotieren c gemäß der Induktionsannahme. Sei $VC'(c, I) = \langle A, V\rangle$. Dann

   $$VC'(\{I\}\mathsf{while}\ b\ \mathsf{do}\ c,\ I \wedge \neg b) = (I, V \cup \{I \wedge b \Rightarrow A,\ I \wedge \neg b \Rightarrow I \wedge \neg b\})$$

   Damit gilt die Behauptung, da die Induktionsannahme für $VC'(c, I) = \langle A, V\rangle$ liefert, dass $I \wedge b \models A$ und $\models V$.

3. Abschwächungsregel. Sei $A \models A'$, $\vdash \{A'\}c\{B'\}$ und $B' \models B$. Wir annotieren c gemäß der Induktionsannahme. Sei $VC'(c, B') = (A'', V)$ und $VC'(c, B) = (A''', V')$. Nach Satz 7.5.1 gilt $A'' \models A'''$ und $V \models V'$. Damit gilt die Behauptung, da die Induktionsannahme für $VC'(c, B') = (A'', V)$ liefert, dass $A' \models A''$ und $\models V$.

□

Korollar 7.5.3 (Vollständigkeit von VC) *Sei* $\vdash \{A\}c\{B\}$. *Dann kann man c so mit Invarianten annotieren, dass* $\models VC(A, c, B)$.

Wir wissen jetzt, dass wir der VC-Methode genau die Hoare-Tripel verifizieren können, die wir mit den Hoare-Regeln ableiten können.

7.6 Schwächste Vorbedingungen

Ein $W \in \mathit{Assn}$ heißt *schwächste Vorbedingung (SVB)* für $c \in \mathit{Com}$ und $B \in \mathit{Assn}$ genau dann, wenn:

$$\forall \sigma \in \Sigma\colon\ (\sigma \models W \iff \mathcal{C}[\![c]\!]\sigma \models B)$$

Eine *SVB-Funktion* ist eine Funktion

$$w \in \mathit{Com} \to \mathit{Assn} \to \mathit{Assn}$$

für die gilt:

$$\forall c \in \mathit{Com}\ \forall B \in \mathit{Assn}\ \forall \sigma \in \Sigma\colon\ (\sigma \models wcB \iff \mathcal{C}[\![c]\!]\sigma \models B)$$

Proposition 7.6.1 *Sei w eine SVB-Funktion. Dann gilt für alle $c, c' \in \mathit{Com}$ und $A, B, B' \in \mathit{Assn}$:*

1. $\models \{A\}c\{B\} \iff A \models wcB$.
2. $(\mathcal{C}[\![c]\!] = \mathcal{C}[\![c']\!] \wedge \mathcal{D}[\![B]\!] = \mathcal{D}[\![B']\!]) \Rightarrow \mathcal{D}[\![wcB]\!] = \mathcal{D}[\![wc'B']\!]$.

Die Behauptungen der Proposition folgen direkt aus den Definitionen.

Wir werden zeigen, dass eine SVB-Funktion existiert. Der Beweis dieser Tatsache ist nicht einfach. Ein entsprechendes Resultat wurde erstmals 1978 von Stephen A. Cook publiziert. Cook verwendete dabei Techniken, die der Logiker Kurt Gödel um 1930 entwickelt hat.

Proposition 7.6.2 *Seien w und w' zwei SVB-Funktionen. Dann:*

$$\forall c \in \mathit{Com}\ \forall B \in \mathit{Assn}\colon\ wcB \models\!\!\!\!\!= w'cB$$

Proposition 7.6.3 *Sei w eine SVB-Funktion. Dann gilt für alle $c, c_1, c_2 \in \mathit{Com}$, $X \in \mathit{Loc}$, $a \in \mathit{Aexp}$ und $B \in \mathit{Assn}$:*

1. $w(\textsf{skip})B \models\!\!\!\!\!= B$.
2. $w(X := a)B \models\!\!\!\!\!= B[a/X]$.
3. $w(c_1; c_2)B \models\!\!\!\!\!= w(c_1)(wc_2B)$.
4. $w(\textsf{if}\ b\ \textsf{then}\ c_1\ \textsf{else}\ c_2)B \models\!\!\!\!\!= b \wedge wc_1B \vee \neg b \wedge wc_2B$.
5. $w(\textsf{while}\ b\ \textsf{do}\ c)B \models\!\!\!\!\!= b \wedge wc(w(\textsf{while}\ b\ \textsf{do}\ c)B) \vee \neg b \wedge B$.

Beweis Jede Behauptung folgt durch einfaches Rechnen. Wir zeigen hier nur die Teilbehauptungen (2), (3) und (5). Sei $\sigma \in \Sigma$.

Behauptung (2)

$$
\begin{array}{llll}
\sigma \models w(X := a)B & \iff & \mathcal{C}[\![X := a]\!]\sigma \models B & \text{Definition von } w \\
 & \iff & \sigma[\mathcal{A}[\![a]\!]\sigma/X] \models B & \text{Definition von } \mathcal{C} \\
 & \iff & \sigma \models B[a/X] & \text{Substitutions-Lemma}
\end{array}
$$

Behauptung (3)

$$
\begin{array}{lll}
\multicolumn{2}{l}{\sigma \models w(c_1; c_2)B} & \\
\iff & \mathcal{C}[\![c_1; c_2]\!]\sigma \models B & \text{Def. von } w \\
\iff & \mathcal{C}[\![c_1]\!]\sigma \neq \bot \Rightarrow \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma) \models B & \text{Def. von } \mathcal{C} \\
\iff & \mathcal{C}[\![c_1]\!]\sigma \neq \bot \Rightarrow \mathcal{C}[\![c_1]\!]\sigma \models wc_2B & \text{Def. von } w \\
\iff & \mathcal{C}[\![c_1]\!]\sigma \models wc_2B & \\
\iff & \sigma \models wc_1(wc_2B) & \text{Def. von } w
\end{array}
$$

Behauptung (5)

$$
\begin{array}{lll}
\multicolumn{2}{l}{\sigma \models w(\textsf{while } b \textsf{ do } c)B} & \\
\iff & \sigma \models w(\textsf{if } b \textsf{ then } c; \textsf{while } b \textsf{ do } c \textsf{ else skip})B & \text{Prop 7.6.1 und 5.3.2} \\
\iff & \sigma \models b \wedge w(c; \textsf{while } b \textsf{ do } c)B \vee \neg b \wedge w(\textsf{skip})B & \text{Teil (4)} \\
\iff & \sigma \models b \wedge w(c; \textsf{while } b \textsf{ do } c)B \vee \neg b \wedge B & \text{Teil (1)}
\end{array}
$$

□

Satz 7.6.4 *Sei w eine SVB-Funktion. Dann:*

$$\forall c \in \textsf{Com}\, \forall B \in \textsf{Assn}: \ \vdash \{wcB\}c\{B\}$$

Beweis Durch Induktion über $c \in Com$. Wir zeigen die Beweisteile für Zuweisung, Sequentialisierung und Schleifen.

$c = (x := a)$

$$
\begin{array}{ll}
\vdash \{B[a/X]\}c\{B\} & \text{Regel für :=} \\
\vdash \{w(X := a)B\}c\{B\} & \text{Prop. 7.6.3}
\end{array}
$$

$c = (c_1; c_2)$

$$
\begin{array}{ll}
\vdash \{wc_2B\}c_2\{B\} & \text{Induktionsannahme} \\
\vdash \{wc_1(wc_2B)\}c_1\{wc_2B\} & \text{Induktionsannahme} \\
\vdash \{wc_1(wc_2B)\}c_1; c_2\{B\} & \text{Regel für ;} \\
\vdash \{w(c_1; c_2)B)\}c_1; c_2\{B\} & \text{Prop 7.6.3 und Abschwächungsregel}
\end{array}
$$

$c = (\textsf{while}\ b\ \textsf{do}\ c')$

$I \stackrel{\text{def}}{=} wcB$	
$b \wedge I \models\!\!\!\dashv b \wedge wc'I$	Prop 7.6.3
$\neg b \wedge I \models\!\!\!\dashv \neg b \wedge B$	Prop 7.6.3
$\vdash \{wc'I\}c'\{I\}$	Induktionsannahme
$\vdash \{I \wedge b\}c'\{I\}$	Abschwächungsregel
$\vdash \{I\}c\{I \wedge \neg b\}$	Regel für $\textsf{while}$
$\vdash \{wcB\}c\{B\}$	Abschwächungsregel

□

Korollar 7.6.5 *Wenn eine SVB-Funktion existiert, dann:*

$$\forall A \in \mathit{Assn}\, \forall c \in \mathit{Com}\, \forall B \in \mathit{Assn}\colon \quad \models \{A\}c\{B\} \;\Rightarrow \vdash \{A\}c\{B\}$$

Beweis Sei w eine SVB-Funktion und $\models \{A\}c\{B\}$. Dann $A \models wcB$ mit Proposition 7.6.1. Mit dem obigen Satz folgt $\vdash \{wcB\}c\{B\}$. Also folgt $\vdash \{A\}c\{B\}$ mit der Abschwächungsregel. □

Im Folgenden verwenden wir die Funktion

$$LC \in Com \to \mathcal{P}_{fin}(Loc)$$

die uns zu jedem Kommando c die Menge aller Lokationen liefert, die in c vorkommen.

Ein *Transformator* für ein Kommando $c \in Com$ ist eine Funktion

$$\tau \in Assn \to Assn$$

für die gilt:

1. $\forall B \in Assn\colon\ \tau B$ ist SVB für c, B.
2. $\forall B \in Assn\colon\ FV(\tau B) \subseteq FV(B) \cup LC(c)$.

Ein *Schleifenkombinator* ist eine Funktion

$$\omega \in Bexp \times (Assn \to Assn) \to (Assn \to Assn)$$

für die gilt:

$$\begin{array}{l} \forall c \in Com\ \forall \tau \in Assn \to Assn\ \forall b \in Bexp\colon \\ \qquad \tau \text{ Transformator für } c \;\Rightarrow\; \omega(b, \tau) \text{ Transformator für } \textsf{while}\ b\ \textsf{do}\ c \end{array}$$

Satz 7.6.6 *Sei ω ein Schleifenkombinator und sei w wie folgt definiert:*

$$
\begin{aligned}
&w \in \mathit{Com} \to \mathit{Assn} \to \mathit{Assn}\\
&\qquad\qquad w(\mathsf{skip})B &&= B\\
&\qquad\qquad w(X := a)B &&= B[a/X]\\
&\qquad\qquad w(c_1; c_2)B &&= wc_1(wc_2B)\\
&w(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)B &&= b \wedge wc_1B \vee \neg b \wedge wc_2B\\
&\qquad w(\mathsf{while}\ b\ \mathsf{do}\ c)B &&= \omega(b,\ wc)B
\end{aligned}
$$

Dann ist ωc für alle $c \in \mathit{Com}$ ein Transformator für c.

Beweis Machen Sie sich zuerst klar, dass w ordnungsgemäß mittels struktureller Rekursion über $c \in \mathit{Com}$ definiert ist. Wir beweisen durch strukturelle Induktion über $c \in \mathit{Com}$, dass

$$\forall c \in \mathit{Com}\ \forall B \in \mathit{Assn}\ \forall \sigma \in \Sigma\colon\ (\sigma \models wcB \iff \mathcal{C}[\![c]\!]\sigma \models B)$$

Wir beschränken uns auf die Teilbeweise für Zuweisungen, Sequentialisierung und Schleifen:

$c = (X := a)$

$$
\begin{aligned}
\sigma \models wcB &\iff \sigma \models B[a/X] && \text{Definition von } w\\
&\iff \sigma[\mathcal{A}[\![a]\!]\sigma/X] \models B && \text{Substitutionseigenschaft}\\
&\iff \mathcal{C}[\![c]\!]\sigma \models B && \text{Definition von } \mathcal{C}
\end{aligned}
$$

$c = (c_1; c_2)$

$$
\begin{aligned}
&\sigma \models wcB\\
&\iff \sigma \models wc_1(wc_2B) && \text{Def. von } w\\
&\iff \mathcal{C}[\![c_1]\!]\sigma \models wc_2B && \text{Induktionsannahme für } c_1\\
&\iff \mathcal{C}[\![c_1]\!]\sigma \neq \bot \Rightarrow \mathcal{C}[\![c_1]\!]\sigma \models wc_2B\\
&\iff \mathcal{C}[\![c_1]\!]\sigma \neq \bot \Rightarrow \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma) \models B && \text{Induktionsannahme für } c_2\\
&\iff \mathcal{C}[\![c]\!]\sigma \models B && \text{Definition von } \mathcal{C}
\end{aligned}
$$

$c = (\mathsf{while}\ b\ \mathsf{do}\ c')$

$$
\begin{aligned}
\sigma \models wcB &\iff \sigma \models \omega(b, wc')B && \text{Def. von } w\\
&\iff \mathcal{C}[\![c]\!]\sigma \models B && \text{Induktionsannahme für } c'
\end{aligned}
$$

□

7.7 Konstruktion eines Schleifenkombinators

Im Folgenden nehmen wir an, dass eine endliche Menge

$$L \in \mathcal{P}_{\mathit{fin}}(\mathit{Loc})$$

von Lokationen und eine bijektive Funktion

$$\lambda \in L \to \{1, \dots, |L|\}$$

gegeben sind. Dabei ist $|L|$ die Anzahl der Lokationen in L. Wir benötigen die folgende Notationen:

$$\begin{aligned}
\Sigma_L &\stackrel{\text{def}}{=} \{\, \sigma \in \Sigma \mid \forall X \in \mathit{Loc} - L\colon\ \sigma X = 0 \,\} \\
\mathit{Com}_L &\stackrel{\text{def}}{=} \{\, c \in \mathit{Com} \mid c \text{ enthält nur Lokationen in } L \,\} \\
\mathit{Assn}_L &\stackrel{\text{def}}{=} \{\, A \in \mathit{Assn} \mid FV(A) \subseteq L \,\} \\
\mathit{Bexp}_L &\stackrel{\text{def}}{=} \{\, b \in \mathit{Bexp} \mid FV(b) \subseteq L \,\}
\end{aligned}$$

Wir definieren eine Funktion

$$\begin{aligned}
&Z \in \Sigma_L \to \mathit{Assn}_L \\
&Z\sigma = (\bigwedge_{X \in L} X = \sigma X)
\end{aligned}$$

Diese Definition ist etwas schlampig, da wir die Glieder der Konjunktion in einer bestimmten Reihenfolge anordnen müssen. Das ist aber kein Problem, da λ eine totale Ordnung auf L induziert, die wir für die Anordnung benützen können.

Proposition 7.7.1 $\forall \sigma, \sigma' \in \Sigma_L\colon\ \sigma = \sigma' \iff \sigma' \models Z\sigma$.

Proposition 7.7.2 *Sei τ ein Transformator für $c \in \mathit{Com}_L$. Dann:*

$$\forall \sigma, \sigma' \in \Sigma_L\colon\ \mathcal{C}[\![c]\!]\sigma = \sigma' \iff\ \models (Z\sigma \Rightarrow (\tau(Z\sigma') \wedge \neg\tau(\mathsf{false})))$$

Lemma 7.7.3 (Generatorfunktion) *Es gibt eine Funktion $g \in \mathbb{Z}^3 \to \mathbb{Z}$ so dass:*

$$\forall k \geq 1\ \forall \langle x_1, \dots, x_k \rangle \in \mathbb{Z}^k\ \exists u, v \in \mathbb{Z}\ \forall i \in \{1, \dots, k\}\colon\ x_i = g(u, v, i)$$

Mithilfe einer Generatorfunktion kann jedes Tupel von Zahlen durch nur zwei Zahlen u und v repräsentiert werden (*** eine Zahl würde reichen ***). Die Konstruktion einer Generatorfunktion findet man in [Winskel, Kapitel 7.2]. Sie basiert auf Gödels β-Prädikat. Im Folgenden nehmen wir an, dass eine Generatorfunktion g gegeben ist.

Jetzt definieren wir eine Funktion

$$\begin{aligned}
&S \in \mathbb{Z}^3 \to \Sigma_L \\
&S(u, v, i)X = \text{if } X \in L \text{ then } g(u, v, (i-1)|L| + \lambda X) \text{ else } 0
\end{aligned}$$

Proposition 7.7.4 *Sei $k \geq 1$ und $\langle\sigma_1, \ldots, \sigma_k\rangle \in {\Sigma_L}^k$. Dann existieren $u, v \in \mathbb{Z}$ so dass $\sigma_i = S(u, v, i)$ für alle $i \in \{1, \ldots, k\}$.*

Die Proposition sagt, dass sich jedes Tupel von Zuständen in Σ_L durch zwei Zahlen repräsentieren lässt.

Wir definieren eine Funktion

$$T \in \mathbb{Z}^3 \to Assn_L$$
$$T(u, v, i) = \bigwedge_{X \in L} X = g(u, v, (i-1)|L| + \lambda X)$$

Proposition 7.7.5 *Seien $u, v, i \in \mathbb{Z}$ und $\sigma \in \Sigma_L$. Dann:*

1. $\sigma = S(u, v, i) \iff \sigma \models T(u, v, i)$.
2. $Z(S(u, v, i)) \models\!\!\!\dashv T(u, v, i)$.
3. $\forall A \in Assn_L:\ S(u, v, i) \models A \iff\ \models (T(u, v, i) \Rightarrow A)$.

Der Ausgangspunkt für die Konstruktion des Schleifenkombinators ist:

Lemma 7.7.6 *Seien $b \in Bexp_L$, $c \in Com_L$, $B \in Assn_L$ und $\sigma \in \Sigma$. Dann*

$$\mathcal{C}[\![\textsf{while}\ b\ \textsf{do}\ c]\!]\sigma \models B$$

genau dann, wenn

$$\begin{array}{l}
\forall k \geq 1\ \forall \langle\sigma_1, \ldots, \sigma_k\rangle \in \Sigma_L^k: \\
\quad (\ \sigma \models Z\sigma_1 \wedge \\
\quad\ \ \forall i \in \{1, \ldots, k-1\}: \\
\quad\quad \sigma_i \models b\ \wedge\ \mathcal{C}[\![c]\!]\sigma_i = \sigma_{i+1}\ ) \\
\quad \Rightarrow\ \sigma_k \models b \vee B
\end{array}$$

Beweis Die zwei Richtungen der Äquivalenz müssen getrennt gezeigt werden. Die Richtung „⇒“ zeigt man durch Induktion über die Fixpunktkonstruktion (siehe den Beweisteil für Schleifen bei Proposition 5.4.2). Die andere Richtung zeigt man durch Induktion über k. □

Wir formen jetzt die Aussage auf der rechten Seite der Äquivalenz des Lemmas schrittweise um. Schließlich werden wir eine Zusicherung $A \in Assn_L$ konstruieren, so dass die Aussage genau dann gilt, wenn $\sigma \models A$ gilt. Damit ist A eine schwächste Vorbedingung für while b do c und B.

Als erstes machen wir Gebrauch von der Generatorfunktion und eliminieren damit die Quantifizierung über die Zustände σ_i.

$$\begin{array}{l} \forall k \geq 1\ \forall u, v \in \mathbb{Z}: \\ \quad [\ \sigma \models Z(S(u, v, 1)) \wedge \\ \quad\quad \forall i \in \{1, \ldots, k-1\}: \\ \quad\quad\quad S(u, v, i) \models b \ \wedge\ \mathcal{C}[\![c]\!](S(u, v, i)) = S(u, v, i+1)\ ] \\ \quad \Rightarrow\ S(u, v, k) \models b \vee B \end{array}$$

Um die Aussage besser lesbar zu machen, verwenden wir auch eckige Klammern. Sei τ ein Transformator für c. Mithilfe der Propositionen 7.7.5 und 7.7.2 bekommen wir:

$$\begin{array}{l} \forall k \geq 1\ \forall u, v \in \mathbb{Z}: \\ \quad [\ \sigma \models T(u, v, 1) \wedge \\ \quad\quad \forall i \in \{1, \ldots, k-1\}: \\ \quad\quad\quad \models T(u, v, i) \Rightarrow b \wedge\ \models (T(u, v, i) \Rightarrow (\tau(T(u, v, i+1)) \wedge \neg\tau(\mathsf{false})))\ ] \\ \quad \Rightarrow\ \models T(u, v, k) \Rightarrow (b \vee B) \end{array}$$

Diese Aussage lässt sich noch etwas vereinfachen:

$$\begin{array}{l} \forall k \geq 1\ \forall u, v \in \mathbb{Z}: \\ \quad [\ \sigma \models T(u, v, 1) \wedge \\ \quad\quad \forall i \in \{1, \ldots, k-1\}: \\ \quad\quad\quad \models (T(u, v, i) \Rightarrow (b \wedge \tau(T(u, v, i+1)) \wedge \neg\tau(\mathsf{false})))\ ] \\ \quad \Rightarrow\ \models T(u, v, k) \Rightarrow (b \vee B) \end{array}$$

Als nächstes wollen wir die Metaquantifizierung für i durch eine Objektquantifizierung ersetzen. Dazu nehmen wir an, dass wir eine Funktion

$$\tilde{T} \in \mathit{Ter}^3 \to \mathit{Assn}_L$$

konstruieren können, für die gilt:

$$\forall a_1, a_2, a_3 \in \mathit{Ter}:\ T(\mathcal{A}[\![a_1]\!]\sigma, \mathcal{A}[\![a_2]\!]\sigma, \mathcal{A}[\![a_3]\!]\sigma) \models\!\!\!\!\!\!= \tilde{T}(a_1, a_2, a_3)$$

Die Konstruktion von $\tilde{T}$ gelingt mit Gödels β-Prädikat (siehe [Winskel, Kapitel 7.2]).

Jetzt wählen wir eine Variable $X_1 \in \mathit{Loc} - L$ und eliminieren damit die Metaquantifizierung für i:

$$
\begin{aligned}
&\forall k \geq 1\ \forall u, v \in \mathbb{Z}: \\
&\quad [\ \sigma \models T(u, v, 1) \land \\
&\quad\ \models \forall X_1\ ((1 \leq X_1 \land X_1 \leq k-1 \land \tilde{T}(u, v, X_1)) \\
&\qquad\qquad\qquad \Rightarrow (b \land \tau(\tilde{T}(u, v, X_1 + 1)) \land \neg\tau(\mathsf{false})))\ ] \\
&\quad \Rightarrow\ \models T(u, v, k) \Rightarrow (b \lor B)
\end{aligned}
$$

Die in den Objektformeln frei auftretetenden Variablen sind alle in L. Wir schreiben $\forall L$ als Abkürzung für

$$\forall Y_1\ \ldots\ \forall Y_m$$

wobei $L = \{Y_1, \ldots, Y_m\}$ gelten soll. Damit können wir die obige Aussage wie folgt umschreiben:

$$
\begin{aligned}
&\forall k \geq 1\ \forall u, v \in \mathbb{Z}: \\
&\quad \sigma \models (\ [\ T(u, v, 1) \land \\
&\qquad\qquad\quad \forall L\ \forall X_1\ (\ (1 \leq X_1 \land X_1 \leq k-1 \land \tilde{T}(u, v, X_1)) \\
&\qquad\qquad\qquad\qquad\qquad \Rightarrow (b \land \tau(\tilde{T}(u, v, X_1 + 1)) \land \neg\tau(\mathsf{false}))\ )\ ] \\
&\qquad\quad \Rightarrow\ \forall L\ T(u, v, k) \Rightarrow (b \lor B)\ )
\end{aligned}
$$

Wenn wir jetzt noch die Metaquantifizierungen für k, u, v wie vorher für i auf die Objektebene schieben, haben wir eine schwächste Vorbedingung $A \in Assn_L$ für $\mathsf{while}\ b\ \mathsf{do}\ c$ und B konstruiert. Also haben wir das folgende Lemma bewiesen:

Lemma 7.7.7 *Seien $b \in Bexp_L$, $c \in Com_L$, $B \in Assn_L$ und τ ein Transformator für c. Dann kann man eine Formel $A \in Assn_L$ konstruieren, so dass A eine schwächste Vorbedingung für $\mathsf{while}\ b\ \mathsf{do}\ c$ und B ist.*

Satz 7.7.8 *Man kann einen Schleifenkombinator konstruieren.*

7.8 Zusammenfassung

Wir haben gesehen, das wir mit den Formeln von ASSN Spezifikationen für die Kommandos von IMP schreiben können. Spezifikationen bestehen dabei aus einer Vor- und Nachbedingung. Den Begriff der partiellen Korrektheit haben wir basierend auf der denotationalen Semantik von IMP wie folgt definiert:

$$\models \{A\}c\{B\} \iff (\forall \sigma:\ \sigma \models A \Rightarrow \mathcal{C}[\![c]\!]\sigma \models B)$$

Mit den Hoare-Regeln haben wir unabhängig von den beiden Semantiken für IMP eine Relation

$$\vdash \{A\}c\{B\}$$

definiert. Wir haben gezeigt, dass

$$\models \{A\}c\{B\} \iff \vdash \{A\}c\{B\}$$

gilt. Die Richtung von rechts nach links heißt Korrektheit und ist einfach zu zeigen. Die andere Richtung heißt Vollständigkeit und ist nicht einfach zu zeigen. Die Schwierigkeit kommt daher, dass wir die Semantik von Schleifen (ein Rekursionskonstrukt) mithilfe einer Sprache (ASSN) charakterisieren müssen, die kein Rekursionskonstrukt hat. Die Charakterisierung der Semantik von Schleifen gelingt mithilfe von Quantifizierung.

Für die Vollständigkeit der Hoare-Regeln haben wir gezeigt, dass man zu jedem Kommando c und jeder Nachbedingung B eine schwächste Vorbedingung A konstruieren kann, die wie folgt charakterisiert ist:

$$\forall\sigma\colon\ \sigma \models A \iff \mathcal{C}[\![c]\!]\sigma \models B$$

Mithilfe von schwächsten Vorbedingungen kann man zu c, σ, σ' stets eine Formel A von IMP konstruieren (Proposition 7.7.2), für die gilt:

$$\mathcal{C}[\![c]\!]\sigma = \sigma' \iff \models A$$

Man kann also die Sematik von IMP mithilfe der Formeln von ASSN beschreiben.

Um die Verifikation der Gültigkeit von Hoare-Tripeln praktikabel zu machen, haben wir einen Verifikationsbedingungsgenerator VC angegeben, der zu einem mit Schleifeninvarianten annotierten Kommando c und zu Formel A, B endlich viele Formeln berechnet, so dass gilt:

$$\models \{A\}c\{B\} \iff \exists\text{Annotierung von } c\colon\quad \models VC(A, c, B)$$

Kapitel 8

Berechenbarkeit

Auf den ersten Blick sieht es nicht so aus, dass man mit IMP sehr viel berechnen kann. Dieser Eindruck ist falsch. In der Tat kann man mit IMP alles berechnen, was überhaupt berechenbar ist. Das liegt im Wesentlichen daran, dass IMP mit beliebig großen Zahlen rechnen kann und dass Zahlen genügen, um den Zustand beliebiger Datenstrukturen zu codieren.

Mithilfe von IMP können wir die grundlegenden Konzepte der Berechbarkeitstheorie formal definieren. Wir zeigen, dass das Halteproblem für IMP unentscheidbar ist und dass die Menge der gültigen Formeln von ASSN nicht testbar ist (Gödels Unvollständigkeitssatz).

Lesematerial

[Winskel, Anhang A]

8.1 Gödelisierung

Wir zeigen jetzt, wie man Paare von Zahlen als Zahlen codieren kann. Dazu geben wir 4 Funktionen

$$\begin{aligned} \mathit{pair} &\in \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}^+ \\ \mathit{first} &\in \mathbb{Z} \to \mathbb{Z} \\ \mathit{second} &\in \mathbb{Z} \to \mathbb{Z} \\ \mathit{ispair} &\in \mathbb{Z} \to \mathbb{B} \end{aligned}$$

an, die die folgenden Eigenschaften erfüllen:

(1) $\forall n_1, n_2 \in \mathbb{Z}\colon\ \mathit{first}(\mathit{pair}(n_1, n_2)) = n_1$
(2) $\forall n_1, n_2 \in \mathbb{Z}\colon\ \mathit{second}(\mathit{pair}(n_1, n_2)) = n_2$
(3) $\forall n \in \mathbb{Z}\colon\ \mathit{ispair}(n) = 1 \iff (\exists n_1, n_2 \in \mathbb{Z}\colon\ n = \mathit{pair}(n_1, n_2))$

Wir definieren *pair* wie folgt:

$$\mathit{pair}(n_1, n_2) = 2^{sg(n_1)} \cdot 3^{|n_1|} \cdot 5^{sg(n_2)} \cdot 7^{|n_2|}$$

wobei

$$sg(n) = \text{if } n \geq 0 \text{ then } 1 \text{ else } 0$$

Wegen der Eindeutigkeit der Primzahlzerlegung können wir jetzt die restlichen 3 Funktionen wie gewünscht definieren. Man überzeugt sich leicht, dass alle 4 Funktionen mit IMP berechnet werden können.

Mit derselben Technik können wir auch Tupel mit mehr als zwei Komponenten codieren. Auch Listen lassen sich als Zahlen codieren. Dazu ordnen wir der leeren Liste die Zahl 0 zu und erinnern uns daran, dass nichtleere Listen Paare sind.

Die gerade vorgestellte Codierungstechnik bezeichnet man als *Gödelisierung* (nach dem Logiker Kurt Gödel, der diese Technik entdeckt hat). Die einem Objekt o durch die Codierung zugeordnete Zahl bezeichnen wir mit $\#o$ und nennen sie die *Gödelnummer* von o.

In Kapitel 3 haben wir syntaktische Objekte durch Tupel repräsentiert, die man ausgehend von Zahlen und den Elementen von vorgegeben Mengen wie *Var* oder *Loc* erhalten kann. Wenn wir $\mathit{Loc} = \mathbb{N}$ festlegen, sind alle syntaktischen Objekte von IMP und ASSN Tupel, die man ausgehend von Zahlen konstruieren kann. Das bedeutet, dass wir alle syntaktischen Objekte von IMP und ASSN mithilfe von Gödelisierung als Zahlen codieren können. Wir verwenden die folgenden Notationen:

$$\#\mathit{Com} = \{\#c \mid c \in \mathit{Com}\}$$
$$\#\mathit{Assn} = \{\#A \mid A \in \mathit{Assn}\}$$

8.2 Berechenbare Funktionen

Da man immer gödelisieren kann, genügt es, Programme (das heißt Kommandos von IMP) zu betrachten, die zu einer Eingabezahl eine Ausgabezahl berechnen (oder divergieren). Damit wir diese Idee formal fassen können, legen wir eine Lokation

$$X_0 \in Loc$$

fest und nehmen an, dass $\perp$ so gewählt wurde, dass $\perp \notin \mathbb{Z}$. Wir definieren

$$\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$$
$$\sigma_0 = \lambda\, x \in Loc\,.\, 0$$

$$\mathcal{F} \in Com \to \mathbb{Z} \to \mathbb{Z}_\perp$$
$$\mathcal{F}[\![c]\!]n = \text{if } \mathcal{C}[\![c]\!](\sigma_0[n/X_0]) \neq \perp \text{ then } (\mathcal{C}[\![c]\!](\sigma_0[n/X_0]))X_0 \text{ else } \perp$$

Definition 8.2.1 *Eine Funktion $f \in \mathbb{Z} \to \mathbb{Z}_\perp$ heißt* berechenbar *genau dann, wenn es ein $c \in Com$ gibt mit $f = \mathcal{F}[\![c]\!]$.*

Satz 8.2.2 (Universelles Kommando) *Es gibt ein Kommando $U \in Com$, so dass:*

$$\forall c \in Com\ \forall n \in \mathbb{Z}\colon\ \mathcal{F}[\![U]\!](\#(\#c, n)) = \mathcal{F}[\![c]\!]n$$

Beweis Bei U handelt es sich offensichtlich um einen Interpreter für IMP, der in IMP geschrieben ist. Die Konstruktion von IMP gelingt mit Standardtechniken aus der Implementierung von Programmiersprachen. □

Beachten Sie, dass das universelle Kommando wie jedes Kommando nur endlich viele Lokationen enthält. Das bedeutet, dass die Menge der berechenbaren Funktionen nicht kleiner wird, wenn man für *Loc* eine hinreichend große aber endliche Teilmenge von $\mathbb{N}$ verwendet.

Das Standardmodell für die Formalisierung von Berechenbarkeit sind Turing-Maschinen. Alle bisher bekannten Berechnungsmodelle lassen sich durch Turing-Maschinen simulieren. Das gilt auch für Berechnugsmodelle, die realen Computern entsprechen. Daher geht man davon aus, dass alles was berechenbar ist, durch Turing-Maschinen berechenbar ist (sogenannte *Churchsche These*). Berechnungsmodelle, die beliebige Turing-Maschinen simulieren können, nennt man *Turing-vollständig*.

Wir überzeugen uns jetzt davon, dass IMP Turing-vollständig ist. Sei also eine beliebige Turing-Maschine gegeben. Wir können annehmen, dass die Symbole der Turing-Maschine Zahlen sind, da es für die Berechnung einer Turing-Maschine keine Rolle spielt, was für mathematische Objekte die Symbole sind. Das Band repräsentieren wir durch zwei Listen *xs* und *ys*, so dass $rev(xs)@ys$ dem Band entspricht und der Kopf auf das erste Element von *xs* zeigt. Mithilfe von Gödelisierung ist es nun einfach, ein Kommando von IMP anzugeben, dass die Turing-Maschine simuliert. Damit haben wir:

Satz 8.2.3 *Es gibt eine endliche Menge $M \subseteq \mathbb{Z}$, so dass IMP mit Loc $= M$ Turing-vollständig ist.*

8.3 Abzählbare Mengen

Seien X und Y Mengen. Eine Funktion $f \in X \to Y$ heißt

- *injektiv* genau dann, wenn für alle $x_1, x_2 \in X$ gilt:

  $$x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$$

- *surjektiv* genau dann, wenn für alle $y \in Y$ ein $x \in X$ existiert mit $f(x) = y$.

Eine Menge M heißt *abzählbar* genau dann, wenn es eine surjektive Funktion $\alpha \in \mathbb{N} \to M$ gibt. Eine abzählbare Menge enthält also mindestens ein Element. Offensichtlich ist jede nichtleere Teilmenge von $\mathbb{Z}$ abzählbar.

Proposition 8.3.1 *Eine nichtleere Menge M ist genau dann abzählbar, wenn es eine injektive Funktion in $M \to \mathbb{N}$ gibt.*

Mithilfe von Gödelisierung bekommen wir injektive Funktionen in *Com* $\to \mathbb{N}$ und *Assn* $\to \mathbb{N}$. Folglich sind *Com* und *Assn* abzählbar. Da *Com* abzählbar ist, ist auch die Menge der berechenbaren Funktionen abzählbar.

Proposition 8.3.2 *Die Menge der berechenbaren Funktionen ist abzählbar.*

Proposition 8.3.3 *Die Menge $\mathbb{Z} \to \mathbb{Z}_\perp$ ist nicht abzählbar.*

Beweis Durch Widerspruch. Sei $\alpha \in \mathbb{N} \to (\mathbb{Z} \to \mathbb{Z}_\perp)$ surjektiv. Wir definieren

$$f \in \mathbb{Z} \to \mathbb{Z}_\perp$$
$$f(n) = \text{if } (\alpha n)n = 0 \text{ then } 1 \text{ else } 0$$

Offensichtlich gilt

$$\forall n \in \mathbb{N}\colon\ (\alpha n)n \neq f(n)$$

Das ist ein Widerspruch zu der Annahme, dass α surjektiv ist. □

Die gerade verwendete Beweistechnik wird als *Cantors Diagonalargument* bezeichnet.

Wir wissen jetzt, dass es unendlich viele Funktionen in $\mathbb{Z} \to \mathbb{Z}_\perp$ gibt, die nicht berechenbar sind.

8.4 Entscheidbare und testbare Mengen

Sei $M \subseteq \mathbb{Z}$. Ein Kommando $c \in Com$ heißt

- *Entscheider für M* genau dann, wenn gilt:

$$\mathcal{F}[\![c]\!] = \lambda\, n \in \mathbb{Z}\,\boldsymbol{.}\ \text{if } n \in M \text{ then } 1 \text{ else } 0$$

- *Tester für M* genau dann, wenn gilt:

$$M = \{\, n \in \mathbb{Z} \mid \mathcal{F}[\![c]\!]n \neq \bot \,\}$$

Eine Menge $M \subseteq \mathbb{Z}$ heißt

1. *entscheidbar* genau dann, wenn es einen Entscheider für M gibt.
2. *unentscheidbar* genau dann, wenn es keinen Entscheider für M gibt.
3. *testbar* genau dann, wenn es einen Tester für M gibt. Statt testbar sagt man auch *rekursiv aufzählbar*.

Für eine unentscheidbare Menge M können wir kein Programm schreiben, dass für beliebige $N \in \mathbb{Z}$ terminiert und genau dann 1 liefert, wenn $n \in M$. Für eine nicht testbare Menge M können wir kein Programm schreiben, dass für beliebige $N \in \mathbb{Z}$ genau dann terminiert, wenn $n \in M$. Wir werden zeigen, dass es testbare Mengen gibt, die unentscheidbar sind.

Proposition 8.4.1 *Die Mengen #Com und #Assn sind entscheidbar.*

Proposition 8.4.2 *$\mathcal{P}(\mathbb{Z})$ ist nicht abzählbar.*

Proposition 8.4.3 *Es gibt Teilmengen von $\mathbb{Z}$, die nicht testbar sind.*

Proposition 8.4.4 *Seien M_1 und M_2 entscheidbare Mengen. Dann sind die Mengen $M_1 \cup M_2$, $M_1 \cap M_2$ und $M_1 - M_2$ entscheidbar.*

Ein *steuerbarer Tester* für eine Menge $M \subseteq \mathbb{Z}$ ist ein Kommando $c \in Com$ wie folgt:

(1) $\forall n \in \mathbb{Z}\colon\ \mathcal{F}[\![c]\!]n \neq \bot$

(2) $\forall n \in \mathbb{Z}\colon\ n \in M \iff \exists k \in \mathbb{N}\colon\ \mathcal{F}[\![c]\!](\#(n,k)) = 1$

(3) $\forall n \in \mathbb{Z}\ \forall k \in \mathbb{N}\colon\ \mathcal{F}[\![c]\!](\#(n,k)) = 1 \Rightarrow \forall m \geq k\colon\ \mathcal{F}[\![c]\!](\#(n,m)) = 1$

Proposition 8.4.5 *Sei $c \in Com$ ein Tester für M. Dann kann man aus c einen steuerbaren Tester für M konstruieren.*

Beweis Den steuerbaren Tester erhält man dadurch, dass man einen Zähler einbaut, der die Anzahl der Schleifendurchläufe zählt. Zusätzlich werden die Schleifenbedingungen so verstärkt, dass nach Überschreiten der durch das zweite Argument vorgegebenen Maximalanzahl keine weiteren Schleifendurchläufe mehr möglich sind.

Seien Z und S zwei Lokationen, die nicht in c vorkommen. Sei c' das Kommando, das man aus c wie folgt erhält:

1. Ersetze jede Schleifenbedingung b durch $b \wedge Z \leq S$.
2. Ersetze jeden Schleifenrumpf c'' durch $Z := Z + 1\ ;\ c''$.

Der steuerbare Tester sieht jetzt wie folgt aus:

$$\begin{array}{l}
\textsf{if } \mathit{ispair}(X_0) \textsf{ then } S := \mathit{second}(X_0);\ \ X_0 := \mathit{first}(X_0) \textsf{ else } S := -1; \\
Z := 0; \\
c'; \\
\textsf{if } Z \leq S \textsf{ then } X_0 := 1 \textsf{ else } X_0 := 0
\end{array}$$

□

Proposition 8.4.6 *Seien M_1 und M_2 testbare Mengen. Dann sind die Mengen $M_1 \cup M_2$ und $M_1 \cap M_2$ testbar.*

Beweis Aus steuerbaren Testern für M_1 und M_2 kann man Tester für $M_1 \cup M_2$ und $M_1 \cap M_2$ konstruieren. Dabei erhöht man mit einer Schleife schrittweise die maximale Anzahl von möglichen Schleifendurchläufen und führt jeweils beide steuerbaren Tester aus. □

Das Komplement einer Menge $M \subseteq \mathbb{Z}$ bezüglich $\mathbb{Z}$ bezeichnen wir mit

$$\overline{M} \stackrel{\text{def}}{=} \mathbb{Z} - M$$

Proposition 8.4.7 *Eine Menge $M \subseteq \mathbb{Z}$ ist genau dann entscheidbar, wenn M und $\overline{M}$ testbar sind.*

Proposition 8.4.8 *Sei $M \subseteq E \subseteq \mathbb{Z}$ und sei E entscheidbar. Dann:*

$$\overline{M} \textit{ testbar} \iff \overline{M} \cap E \textit{ testbar}$$

Beweis Gilt da $\overline{M} = \overline{E} \cup (\overline{M} \cap E)$. □

8.5 Unentscheidbarkeit des Halteproblems

Wir definieren zwei Mengen:

$$\#H \stackrel{\text{def}}{=} \{\,\#c \mid c \in Com \wedge \mathcal{F}[\![c]\!](\#c) \neq \bot\,\}$$

$$\#H_0 \stackrel{\text{def}}{=} \{\,\#c \mid c \in Com \wedge \mathcal{F}[\![c]\!](0) \neq \bot\,\}$$

Wir werden zeigen, dass beide Menge unentscheidbar sind. Diese Tatsache bezeichnet man als die Unentscheidbarkeit des Halteproblems.

Proposition 8.5.1 *Die Mengen $\#H$ und $\#H_0$ sind testbar.*

Beweis Folgt aus Proposition 8.4.1 und Satz 8.2.2. □

Proposition 8.5.2 *Die Menge $\overline{\#H}$ ist nicht testbar.*

Beweis Durch Widerspruch. Sei T ein Tester für $\overline{\#H}$. Dann haben wir den folgenden Widerspruch:

$$\begin{aligned}
\mathcal{F}[\![T]\!](\#T) = \bot \quad &\iff \#T \notin \#H && \text{Definition von } \#H\\
&\iff \#T \in \overline{\#H}\\
&\iff \mathcal{F}[\![T]\!](\#T) \neq \bot && T \text{ Tester für } \overline{\#H}
\end{aligned}$$

□

Satz 8.5.3 (Halteproblem) *Die Menge $\overline{\#H_0}$ ist nicht testbar.*

Beweis Durch Widerspruch. Sei T ein Tester für $\overline{\#H_0}$. Wir definieren:

$$g \in \mathbb{Z} \to \mathbb{Z}_\bot$$

$$g(n) = \begin{cases} \#(X_0 := \#c\ ;\ c) & \text{falls } c \in Com \text{ und } n = \#c\\ \bot & \text{sonst} \end{cases}$$

Sei G ein Kommando, dass die Funktion g berechnet. Wir wählen G so, dass bei Termination alle von X_0 verschiedenen Lokationen, die in T vorkommen, auf 0 gesetzt werden.

Für alle $c \in Com$ gilt:

$$\begin{aligned}
\#c \in \overline{\#H} \quad &\iff \mathcal{F}[\![c]\!](\#c) = \bot && \text{Definition von } \#H\\
&\iff \mathcal{F}[\![X_0 := \#c\ ;\ c]\!]0 = \bot\\
&\iff \#(X_0 := \#c\ ;\ c) \in \overline{\#H_0} && \text{Definition von } \#H_0\\
&\iff \mathcal{F}[\![T]\!](\#(X_0 := \#c\ ;\ c)) \neq \bot && T \text{ Tester für } \overline{\#H_0}\\
&\iff \mathcal{F}[\![T]\!](\mathcal{F}[\![G]\!](\#c)) \neq \bot && \text{Definition von } G\\
&\iff \mathcal{F}[\![G; T]\!](\#c) \neq \bot
\end{aligned}$$

Also ist $G; T$ ein Tester für $\overline{\#H} \cap \#Com$. Also ist $\overline{\#H}$ wegen Proposition 8.4.8 testbar. Das widerspricht Proposition 8.5.2. □

8.6 Gödels Unvollständigkeitssatz

Sei $M \subseteq \mathbb{Z}$ eine testbare Menge. Dann ist ein Tester für M eine *endliche* Repräsentation von M. Diese Eigenschaft ist interessant, wenn M eine unendliche Menge ist. Am Anfang des zwanzigsten Jahrhunderts vertraten viele Logiker die Ansicht, dass die Menge

$$\#G \stackrel{\text{def}}{=} \{\#A \mid A \in Assn \wedge \models A\}$$

testbar und damit endlich repräsentierbar ist. Der Logiker Kurt Gödel zeigte um 1930, dass dies nicht der Fall ist. Dieses für die Logik fundamentale Ergebnis ist als Gödels Unvollständigkeitssatz bekannt. Aus Gödels Unvollständigkeitssatz folgt, dass kein Computerprogramm das Wissen über die Zahlen, das man mit den gültigen Formeln von ASSN formulieren kann, vollständig repräsentieren kann.

Satz 8.6.1 (Gödels Unvollständigkeitssatz) *Die Menge $\#G$ ist nicht testbar.*

Beweis Durch Widerspruch. Sei $T \in Com$ ein Tester für $\#G$. Weiter sei w eine SVB-Funktion für IMP. Wir definieren:

$$\begin{aligned} &z \in Com \to Com \\ &z(c) = \text{let } \{Y_1, \ldots, Y_n\} = LC(c) \text{ in } (Y_1 := 0;\ \ldots\ ;\ Y_n = 0) \end{aligned}$$

Dabei liefert $LC(c)$ alle Lokationen, die in c vorkommen. Dann gilt für alle $c \in Com$:

$$\begin{array}{llll} \#c \in \overline{\#H_0} & \iff & \mathcal{F}[\![c]\!]0 = \bot & \text{Definition von } \overline{\#H_0} \\ & \iff & \forall \sigma \in \Sigma:\ \mathcal{C}[\![z(c); c]\!]\sigma = \bot & \text{Definition von } \mathcal{F} \\ & \iff & \models w(z(c); c)(\mathsf{false}) & w \text{ ist SVB-Funktion} \\ & \iff & \mathcal{F}[\![T]\!](\#(w(z(c); c)(\mathsf{false}))) \neq \bot & \end{array}$$

In Kapitel 7 haben wir gezeigt, dass man für jedes Paar (c, B) eine schwächste Vorbedingung konstruieren kann (in dem Sinne, dass sich die Konstruktion beispielsweise mit einem in Standard ML geschriebenen Programm ausführen lässt). Daraus folgt, dass man $\#(w(z(c); c)(\mathsf{false}))$ im formalen Sinne aus $\#c$ berechnen kann. Damit haben wir einen Tester für $\overline{\#H_0} \cap \#Com$. Also ist $\overline{\#H_0}$ wegen Proposition 8.4.8 testbar. Das widerspricht Satz 8.5.3. □

Korollar 8.6.2 *Die Menge $\overline{\#G}$ ist nicht testbar.*

Beweis Durch Widerspruch. Sei T ein Tester für $\overline{\#G}$. Dann gilt für alle

$A \in Assn$:

$$\begin{array}{lll} \#A \in \#G & \iff \models A & \text{Definition von } \#G \\ & \iff \models \forall A & \\ & \iff \not\models \neg\forall A & \\ & \iff \#(\neg\forall A) \in \overline{\#G} & \text{Definition von } \#G \\ & \iff \mathcal{F}[\![T]\!](\#(\neg\forall A)) \neq \bot & T \text{ Tester für } \overline{\#G} \end{array}$$

Dabei bezeichnet $\forall A$ eine Formel $\forall X_1 \ldots \forall X_n \colon A$ mit $FV(A) = \{X_1, \ldots, X_n\}$. Die obigen Äquivalenzen geben uns einen Tester für $\#G \cap \#Assn$. Also ist $\#G$ wegen Proposition 8.4.8 testbar. Das widerspricht Satz 8.6.1. $\square$

Kapitel 9

Aussagenlogik

In diesem Kapitel behandeln wir für Techniken zum Lösen von Booleschen Gleichungen (zum Beispiel $x \wedge y = x \vee y$). Boolesche Gleichungen sind dadurch ausgezeichnet, dass ihre Variablen nur zwei Werte annehmen können. Das Wissensgebiet, dass sich mit Booleschen Gleichungen beschäftigt, wird als Aussagenlogik bezeichnet. Aussagenlogik hat in der Informatik viele Anwendungen. Ein prominentes Beispiel ist die Beschreibung von Schaltkreisen.

Die Sprache der Aussagenlogik ist eine Teilsprache jeder prädikatenlogischen Sprache (zum Beispiel ASSN). Daher bilden die Ergebnisse und Methoden der Aussagenlogik eine wichtige Grundlage der prädikatenlogischen Sprachen.

Lesematerial

[Schöning, Kapitel 1], [Fitting, Kapitel 2+3]

9.1 Zweiwertige Boolesche Algebra

Unser Ausgangspunkt ist die bereits eingeführte zweielementige Menge

$$\mathbb{B} = \{0, 1\}$$

Im Kontext der Aussagenlogik nennen wir die Elemente von $\mathbb{B}$ *Boolesche Werte* und die Funktionen in $\mathbb{B}^n \to \mathbb{B}$ *Boolesche Funktionen* ($n \geq 1$). Hier sind

Beispiele für bekannte Boolesche Funktionen:

$\neg \in \mathbb{B} \to \mathbb{B}$ *Negation*
$(\neg x) = 1 - x$

$\wedge \in \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ *Konjunktion*
$(x \wedge y) = \min\{x, y\}$

$\vee \in \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ *Disjunktion*
$(x \vee y) = \max\{x, y\}$

$\Rightarrow \in \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ *Implikation*
$(x \Rightarrow y) = \max\{1 - x, y\}$

Das Tupel $\langle \mathbb{B}, \neg, \wedge, \vee \rangle$ bezeichnet man als *die zweiwertige Boolesche Algebra*.

Die Wahl der Zahlen 0 und 1 als Boolsche Werte ist willkürlich. Man kann auch beliebige andere mathematische Objekte als Boolesche Werte wählen. Wichtig ist nur, dass $\mathbb{B}$ eine zweielementige Menge ist. Die Booleschen Werte 0 und 1 bezeichnet man auch als die „Wahrheitswerte" „wahr" und „falsch".

Abbildung 9.1 zeigt einige gültige Boolesche Gleichungen. Offensichtlich verhalten sich Konjunktion und Disjunktion symmetrisch. Diese Symmetrie bezeichnet man auch als *Dualität*.

Um die Gültigkeit der Gleichungen in Abbildung 9.1 zu zeigen, rechnet man sie am besten für alle Werte der in ihnen vorkommenden Variablen nach. Dieses Nachrechnen lässt sich mithilfe sogenannter *Wahrheitstafeln* übersichtlich darstellen. Wir zeigen dies am Beispiel der ersten de Morganschen Gleichung:

x	y	$\neg(x \wedge y)$	$\neg x \vee \neg y$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Damit wir nicht so viele Klammern schreiben müssen, priorisieren wir die Boolschen Infixoperatoren wie folgt: $\wedge, \vee, \Rightarrow$. Damit können wir beispielsweise

$$((\neg x \wedge y) \vee z) \Rightarrow (x \wedge y)$$

ohne Klammern schreiben:

$$\neg x \wedge y \vee z \Rightarrow x \wedge y$$

$$
\begin{aligned}
(x \wedge y) \wedge z &= x \wedge (y \wedge z) && \textit{(Assoziativität)}\\
(x \vee y) \vee z &= x \vee (y \vee z) &&\\[1ex]
x \wedge y &= y \wedge x && \textit{(Kommutativität)}\\
x \vee y &= y \vee x &&\\[1ex]
x \wedge x &= x && \textit{(Idempotenz)}\\
x \vee x &= x &&\\[1ex]
x \wedge (y \vee z) &= (x \wedge y) \vee (x \wedge z) && \textit{(Distributivität)}\\
x \vee (y \wedge z) &= (x \vee y) \wedge (x \vee z) &&\\[1ex]
x \wedge (x \vee y) &= x && \textit{(Absorption)}\\
x \vee (x \wedge y) &= x &&\\[1ex]
\neg(x \wedge y) &= \neg x \vee \neg y && \textit{(de Morgan)}\\
\neg(x \vee y) &= \neg x \wedge \neg y &&\\[1ex]
\neg\neg x &= x && \textit{(Doppelnegation)}\\[1ex]
x \Rightarrow y &= \neg x \vee y && \textit{(Implikation)}\\[1ex]
x \wedge \neg x &= 0 && (0 \textit{ und } 1)\\
x \vee \neg x &= 1 &&\\
x \wedge 1 &= x &&\\
x \vee 0 &= x &&
\end{aligned}
$$

Abbildung 9.1: Einige gültige Boolesche Gleichungen ($z, y, z \in \mathbb{B}$)

9.2 Eine Denksportaufgabe

Mit Booleschen Gleichungen kann man einfache Sachverhalte repräsentieren. Wir zeigen dies am Beispiel einer Denksportaufgabe.

> „Worin besteht das Geheimnis Ihres langen Lebens?“ wird ein Hundertjähriger gefragt. „Ich halte mich streng an drei Diätregeln: Wenn ich kein Bier zu einer Mahlzeit trinke, dann esse ich immer Fisch. Immer wenn ich sowohl Fisch und Bier zur selben Mahlzeit esse, verzichte ich auf Eiscreme. Wenn ich Eiscreme esse oder kein Bier trinke, dann esse ich keinen Fisch.“ Können Sie diese Regeln vereinfachen?

Die Diätregeln lassen sich durch Boolesche Gleichungen formalisieren. Dazu führen wir drei Boolesche Variablen ein:

- $B = 1$ genau dann, wenn zu einer Mahlzeit Bier getrunken wird.
- $F = 1$ genau dann, wenn zu einer Mahlzeit Fisch gegessen wird.
- $E = 1$ genau dann, wenn zu einer Mahlzeit Eiscreme gegessen wird.

Jetzt können wir die Diätregeln wie folgt formalisieren:

$$\begin{aligned} (\neg B \Rightarrow F) &= 1 \\ (F \wedge B \Rightarrow \neg E) &= 1 \\ (E \vee \neg B \Rightarrow \neg F) &= 1 \end{aligned}$$

Diese Gleichungen können wir zu einer Booleschen Gleichung zusammenfassen:

$$((\neg B \Rightarrow F) \ \wedge \ (F \wedge B \Rightarrow \neg E) \ \wedge \ (E \vee \neg B \Rightarrow \neg F)) \ = \ 1$$

Mithilfe einer Wahrheitstafel überzeugt man sich leicht, dass die folgende Gleichung gilt:

$$((\neg B \Rightarrow F) \ \wedge \ (F \wedge B \Rightarrow \neg E) \ \wedge \ (E \vee \neg B \Rightarrow \neg F)) \ = \ (B \wedge \neg(F \wedge E))$$

Also kann man die Diätregeln des Hundertjährigen auch wie folgt formulieren: „Trinke zu jeder Mahlzeit Bier und esse zu keiner Mahlzeit sowohl Fisch und Eiscreme“.

9.3 Die logische Sprache AL

Abbildung 9.2 definiert relativ zu einer Menge *Var* eine logische Sprache AL. Die Formel von AL formalisieren Boolesche Ausdrücke, die sich mit Variablen sowie Negation und Konjunktion bilden lassen.

$$
\begin{array}{lcll}
X, Y & \in & \mathit{Var} & \textit{Variable} \\
A, B & \in & \mathit{For} = & \textit{Formel} \\
 & & \quad X & \quad \textit{Atom} \\
 & & \mid\ \neg A & \quad \textit{Negation} \\
 & & \mid\ A_1 \wedge A_2 & \quad \textit{Konjunktion}
\end{array}
$$

$$
\sigma \quad \in \quad \Sigma \; = \; \mathit{Var} \to \mathbb{B} \quad \textit{Belegung}
$$

$$
\mathcal{F} \in \mathit{For} \to \Sigma \to \mathbb{B}
$$

$$
\begin{aligned}
\mathcal{F}[\![X]\!]\sigma &= \sigma X \\
\mathcal{F}[\![\neg A]\!]\sigma &= 1 - \mathcal{F}[\![A]\!]\sigma \\
\mathcal{F}[\![A_1 \wedge A_2]\!]\sigma &= \min\{\mathcal{F}[\![A_1]\!]\sigma,\ \mathcal{F}[\![A_2]\!]\sigma\}
\end{aligned}
$$

Abbildung 9.2: Syntax und Semantik der Sprache AL

$$
\begin{array}{rcl}
0 & \mapsto & X_0 \wedge \neg X_0 \\
1 & \mapsto & \neg 0 \\
A_1 \vee A_2 & \mapsto & \neg(\neg A_1 \wedge \neg A_2) \\
A_1 \Rightarrow A_2 & \mapsto & \neg A_1 \vee A_2 \\
A_1 \Leftrightarrow A_2 & \mapsto & (A_1 \Rightarrow A_2) \wedge (A_2 \Rightarrow A_1)
\end{array}
$$

Abbildung 9.3: Abkürzungen für AL

Auf den ersten Blick erscheint AL etwas karg, da nur für Negation und Konjunktion Syntax vorhanden ist. Alle anderen Boolschen Funktionen sowie die Booleschen Werte lassen sich jedoch mit Negation und Konjunktion ausdrücken. Entsprechende Abkürzungen werden in Abbildung 9.3 definiert. Dabei ist X_0 eine festgewählte Variable in *Var*.

Für AL definieren wir die folgenden Sprechweisen und Notationen:

$$
\begin{array}{rcl}
A \models\!\!\mid B \overset{\text{def}}{\iff} & A \textit{ äquivalent } B & \overset{\text{def}}{\iff} \mathcal{F}[\![A]\!] = \mathcal{F}[\![B]\!] \\
\models A \overset{\text{def}}{\iff} & A \textit{ gültig} & \overset{\text{def}}{\iff} \forall\sigma \in \Sigma:\ \mathcal{F}[\![A]\!]\sigma = 1 \\
& A \textit{ unerfüllbar} & \overset{\text{def}}{\iff} \forall\sigma \in \Sigma:\ \mathcal{F}[\![A]\!]\sigma = 0 \\
& A \textit{ erfüllbar} & \overset{\text{def}}{\iff} \exists\sigma \in \Sigma:\ \mathcal{F}[\![A]\!]\sigma = 1 \\
\sigma \models A \overset{\text{def}}{\iff} & \sigma \textit{ erfüllt } A & \overset{\text{def}}{\iff} \mathcal{F}[\![A]\!]\sigma = 1
\end{array}
$$

Gültige aussagenlogische Formeln bezeichnet man auch als *Tautologien*.

Proposition 9.3.1 *Für jede Formel $A \in For$ gilt:*

$$
\begin{array}{rcccl}
A \textit{ gültig} & \iff & \neg A \textit{ unerfüllbar} & \iff & A \models\!\!\mid 1 \\
A \textit{ unerfüllbar} & \iff & \neg A \textit{ gültig} & \iff & A \models\!\!\mid 0 \\
A \models\!\!\mid B & \iff & A \Leftrightarrow B \textit{ gültig} & &
\end{array}
$$

Die drei Eigenschaften Gültigkeit, Unerfüllbarkeit und Äquivalenz können sich also wechselseitig ausdrücken.

Da jede Formel nur endlich viele Variablen enthält und für jede Variable nur zwei Werte möglich sind, kann man die gerade definierten Eigenschaften alle entscheiden, indem man die beteiligten Formeln für alle relevanten Variablenbelegungen ausrechnet und die Ergebnise in einer Wahrheitstafel vermerkt. Im Folgenden entwickeln wir Techniken, mit denen sich die obigen Eigenschaften in der Regel effizienter entscheiden lassen als mit der naiven Enumerationsmethode. Allerdings gibt es für jede dieser Techniken immer „worst case“ Beispiele, für die sie nicht effizienter als die Enumerationsmethode sind. Der folgende Satz aus der Komplexitätstheorie sagt, dass dies nicht überraschend ist.

Satz 9.3.2 (Cook 1971) *Das Problem „gegeben $A \in For$, entscheide ob A erfüllbar ist“ ist NP-vollständig.*

Seien A, B Formeln und X eine Variable. Dann bezeichnen wir mit

$$A[B/X]$$

die Formel, die wir aus A erhalten, wenn wir jedes Auftreten der Variable X durch die Formel B ersetzen. Eine rekursive Definition der entsprechenden Substitutionsfunktion sieht wie folgt aus:

$$
\begin{aligned}
Y[B/X] &= \text{if } X = Y \text{ then } B \text{ else } Y \\
(\neg A)[B/X] &= \neg(A[B/X]) \\
(A_1 \wedge A_2)[B/X] &= A_1[B/X] \wedge A_2[B/X]
\end{aligned}
$$

$$\mathcal{M} \in For \to \mathcal{P}(\Sigma)$$

$$\begin{aligned}
\mathcal{M}[\![X]\!] &= \{\sigma \in \Sigma \mid \sigma X = 1\} \\
\mathcal{M}[\![\neg A]\!] &= \Sigma - \mathcal{M}[\![A]\!] \\
\mathcal{M}[\![A_1 \wedge A_2]\!] &= \mathcal{M}[\![A_1]\!] \cap \mathcal{M}[\![A_2]\!]
\end{aligned}$$

Abbildung 9.4: Die mengenwertige Denotationsfunktion

Satz 9.3.3 (Ersetzung) *Seien $A_1, A_2, B \in For$ und $X \in Var$. Dann:*

$$A_1 \models\!\mid A_2 \;\Rightarrow\; B[A_1/X] \models\!\mid B[A_2/X]$$

Beweis Durch strukturelle Induktion über $B \in For$. □

9.4 Boolesche Operationen als Mengenoperationen

Sei M eine Menge. Dann ist die Funktion

$$CF \in \mathcal{P}(M) \to (M \to \mathbb{B})$$
$$CF(U) = \lambda\, x \in M \,.\, \text{if } x \in U \text{ then } 1 \text{ else } 0$$

bijektiv. Man bezeichnet $CF(U)$ als die *charakteristische Funktion* von U.

Die bijektive Korrespondenz zwischen $\Sigma \to \mathbb{B}$ und $\mathcal{P}(\Sigma)$ versetzt uns in die Lage, die Denotation einer Formel A statt durch eine Funktion in $\Sigma \to \mathbb{B}$ durch eine Teilmenge von Σ zu beschreiben. Die Denotation einer Formel ist dann gerade die Menge aller Belegungen, die die Formel erfüllen. Mit dieser Sichtweise entpuppen sich die Booleschen Operationen Negation, Konjunktion und Disjunktion als die Mengenoperationen Komplement, Durchschnitt und Vereinigung. Wir realisieren diese Idee mit der in Abbildung 9.4 definierten Denotationsfunktion. Offensichtlich gelten die folgenden Gleichungen:

$$\begin{aligned}
\mathcal{M}[\![A \vee B]\!] &= \mathcal{M}[\![A]\!] \cup \mathcal{M}[\![B]\!] \qquad \text{(für alle } A, B \in For\text{)} \\
\mathcal{M}[\![1]\!] &= \Sigma \\
\mathcal{M}[\![0]\!] &= \emptyset
\end{aligned}$$

Die Denotationsfunktionen $\mathcal{F}$ und $\mathcal{M}$ liefern dieselbe Äquivalenzbeziehung zwischen Formeln:

Proposition 9.4.1 *Seien $A, B \in For$ und $\sigma \in \Sigma$. Dann:*

1. $\mathcal{M}[\![A]\!] = \{\sigma \in \Sigma \mid \mathcal{F}[\![A]\!]\sigma = 1\}$.

2. $A \models\!\!\mid B \iff \mathcal{F}[\![A]\!] = \mathcal{F}[\![B]\!] \iff \mathcal{M}[\![A]\!] = \mathcal{M}[\![B]\!]$.

Beweis Die erste Aussage zeigt man mithilfe von struktureller Induktion über $A \in \mathit{For}$. Die zweite Aussage folgt aus der ersten. □

Die Lösungsmenge einer Gleichung $A = B$ ist die Menge $\mathcal{M}[\![A \Leftrightarrow B]\!]$:

Proposition 9.4.2 *Seien $A, B \in \mathit{For}$. Dann:*

$$\{\sigma \in \Sigma \mid \mathcal{F}[\![A]\!]\sigma = \mathcal{F}[\![B]\!]\sigma\} = \mathcal{M}[\![A \Leftrightarrow B]\!]$$

Damit genügen zum Lösen von Gleichungen Techniken, die zu einer Formel eine hinreichend explizite Darstellung ihrer $\mathcal{M}$-Denotation berechnen. Solche Techniken werden wir im Folgenden betrachten. Insbesondere wollen wir für eine Formel A entscheiden können, ob $\mathcal{M}[\![A]\!] = \emptyset$ (A unerfüllbar) oder $\mathcal{M}[\![A]\!] = \Sigma$ (A gültig) gilt.

9.5 Klauseln

Konjunktion und Disjunktion sind assoziative, kommutative und idempotente Operationen (siehe Abbildung 9.1). Folglich können wir die Denotation einer konjunktiven Formel

$$A_1 \wedge \cdots \wedge A_n$$

(Klammerung beliebig) auch durch die Menge

$$\{A_1, \ldots, A_n\}$$

repräsentieren. Entsprechendes gilt für disjunktive Formeln. Die Mengenrepräsentation von Konjunktion und Disjunktion ist interessant, da sie die Formulierung von Lösungstechniken erheblich vereinfacht.

Abbildung 9.5 führt eine mengenorientierte Syntax für AL ein. Eine *Klausel* ist eine endliche Menge von Formeln. Für Klauselnmengen geben wir zwei symmetrische Denotationsfunktionen an. Die *disjunktive Denotationsfunktion* $\mathcal{D}$ verknüpft die Klauseln einer Menge disjunktiv und die Formeln einer Klausel konjunktiv. Die *konjunktive Denotationsfunktion* $\mathcal{K}$ verknüpft die Klauseln einer Menge konjunktiv und die Formeln einer Klausel disjunktiv.

Proposition 9.5.1 *Zu jeder endlichen Klauselmenge S existieren Formeln A und B mit $\mathcal{D}[\![S]\!] = A$ und $\mathcal{K}[\![S]\!] = B$.*

$$\begin{array}{rcll} C, D & \in & Cla = \mathcal{P}_{fin}(For) & \quad \textit{Klausel} \\ S & \in & \mathcal{P}(Cla) & \quad \textit{Klauselmenge} \end{array}$$

$$\mathcal{D} \in \mathcal{P}(Cla) \to \mathcal{P}(\Sigma)$$
$$\mathcal{D}[\![S]\!] = \{\, \sigma \in \Sigma \mid \exists C \in S\ \forall A \in C\colon\ \sigma \in \mathcal{M}[\![A]\!]\,\}$$

$$\mathcal{K} \in \mathcal{P}(Cla) \to \mathcal{P}(\Sigma)$$
$$\mathcal{K}[\![S]\!] = \{\, \sigma \in \Sigma \mid \forall C \in S\ \exists A \in C\colon\ \sigma \in \mathcal{M}[\![A]\!]\,\}$$

Abbildung 9.5: Syntax und Semantik von Klauseln

Proposition 9.5.2 $\forall A \in For\colon\ \mathcal{M}[\![A]\!] = \mathcal{D}[\![\{\{A\}\}]\!] = \mathcal{K}[\![\{\{A\}\}]\!]$.

Proposition 9.5.3 (Leere Klauselmenge) $\mathcal{D}[\![\emptyset]\!] = \emptyset$ *und* $\mathcal{K}[\![\emptyset]\!] = \Sigma$.

Proposition 9.5.4 (Leere Klausel) *Sei* $\emptyset \in S \subseteq Cla$. *Dann* $\mathcal{D}[\![S]\!] = \Sigma$ *und* $\mathcal{K}[\![S]\!] = \emptyset$.

Eine Klausel C heißt *trivial* genau dann, wenn es eine Formel A gibt, so dass $A \in C$ und $\neg A \in C$.

Die Denotation einer Klauselmenge ändert sich nicht, wenn man triviale Klauseln löscht:

Proposition 9.5.5 (Triviale Klauseln) *Sei S eine Klauselmenge und C eine triviale Klausel. Dann* $\mathcal{D}[\![S \cup \{C\}]\!] = \mathcal{D}[\![S]\!]$ *und* $\mathcal{K}[\![S \cup \{C\}]\!] = \mathcal{K}[\![S]\!]$.

Eine Klausel C *subsumiert* eine Klausel D genau dann, wenn $C \subseteq D$.

Die Denotation einer Klauselmenge ändert sich nicht, wenn man subsumierte Klauseln löscht:

Proposition 9.5.6 (Subsumierte Klauseln) *Sei S eine Klauselmenge und C eine Klausel. Wenn S eine Klausel D enthält mit $D \subseteq C$, dann* $\mathcal{D}[\![S \cup \{C\}]\!] = \mathcal{D}[\![S]\!]$ *und* $\mathcal{K}[\![S \cup \{C\}]\!] = \mathcal{K}[\![S]\!]$.

Sei S eine Klauselmenge. Wir definieren

$$Ber(S) \stackrel{\text{def}}{=} \{\, C \in S \mid C \text{ nichttrivial und } \forall D \in S\colon\ D \subseteq C \Rightarrow D = C\,\}$$

Offensichtlich gilt $Ber(S) \subseteq S$ und $Ber(Ber(S)) = Ber(S)$.

Eine Klauselmenge S heißt *bereinigt* genau dann, wenn $Ber(S) = S$.

Proposition 9.5.7 (Bereinigung) *Für jede Klauselmenge S gilt:*

$$Ber(S) \subseteq S \quad und \quad \mathcal{D}[\![Ber(S)]\!] = \mathcal{D}[\![S]\!] \quad und \quad \mathcal{K}[\![Ber(S)]\!] = \mathcal{K}[\![S]\!]$$

Ein *Literal* ist eine Formel $A \in For$, so dass eine Variable $X \in Var$ existiert mit $A = X$ oder $A = \neg X$.

Eine Klausel heißt *literal* genau dann, wenn sie nur Literale enthält. Eine Klauselmenge heißt *literal* genau dann, wenn sie nur literale Klauseln enthält.

Eine Klausel heißt *normal* genau dann, wenn sie literal und nichttrivial ist. Eine Klauselmenge heißt *normal* genau dann, wenn sie nur normale Klauseln enthält.

Proposition 9.5.8 (Normale Klauseln) *Sei C eine normale Klausel. Dann ist*

$$\sigma_1 \stackrel{\text{def}}{=} \lambda\, X \in Var.\ \text{if } X \in C \text{ then } 1 \text{ else } 0$$

eine Belegung mit $\sigma_1 \in \mathcal{D}[\![\{C\}]\!]$. Weiter ist

$$\sigma_2 \stackrel{\text{def}}{=} \lambda\, X \in Var.\ \text{if } X \in C \text{ then } 0 \text{ else } 1$$

eine Belegung mit $\sigma_2 \notin \mathcal{K}[\![\{C\}]\!]$.

Proposition 9.5.9 *Für jede normale Klauselmenge S gilt:*

$$\mathcal{D}[\![S]\!] = \emptyset \iff \mathcal{K}[\![S]\!] = \Sigma \iff S = \emptyset$$

Eine endliche und normale Klauselmenge S heißt eine *disjunktive Normalform* für eine Formel A genau dann, wenn $\mathcal{M}[\![A]\!] = \mathcal{D}[\![S]\!]$. Eine endliche und normale Klauselmenge S heißt eine *konjunktive Normalform* für eine Formel A genau dann, wenn $\mathcal{M}[\![A]\!] = \mathcal{K}[\![S]\!]$.

Aus der obigen Proposition folgt:

- $\emptyset$ ist die einzige disjunktive Normalform für unerfüllbare Formeln.
- $\emptyset$ ist die einzige konjunktive Normalform für gültige Formeln.

Für Klauselmengen existieren Vereinfachungsregeln, mit denen man jede Klauselmenge auf eine normale Klauselmenge mit derselben Denotation überführen kann. Eine denotationserhaltene Überführung in eine normale Klauselmenge ist interessant, da damit disjunktive und konjunktive Normalformen berechnet werden können.

Wir unterscheiden zwischen *interner und externer Syntax*. Die interne Syntax ist durch die Formeln in *For* gegeben. Die externe Syntax ist durch Klauseln und Klauselmengen gegeben. Die Vereinfachungsregeln ersetzen interne Syntax schrittweise durch externe Syntax. Bei normalen Klauselmengen kommt die interne Syntax nur noch in der Form von negierten Variablen vor.

$$
\begin{array}{lrcl}
(1) & S,\ (C,\ \neg\neg A) & \stackrel{\mathrm{d}}{\to} & S,\ (C,\ A) \\
(2) & S,\ (C,\ A_1 \wedge A_2) & \stackrel{\mathrm{d}}{\to} & S,\ (C,\ A_1,\ A_2) \\
(3) & S,\ (C,\ A,\ \neg A) & \stackrel{\mathrm{d}}{\to} & S \\
(4) & S,\ (C,\ \neg(A_1 \wedge A_2)) & \stackrel{\mathrm{d}}{\to} & S,\ (C,\ \neg A_1),\ (C,\ \neg A_2)
\end{array}
$$

Abbildung 9.6: Die disjunktiven Vereinfachungsregeln

9.6 Disjunktive Vereinfachung

Abbildung 9.6 zeigt die disjunktiven Vereinfachungsregeln für Klauselmengen in schematischer Darstellung. Dabei ist S, C als Abkürzung für die Klauselmenge $S \cup \{C\}$ und C, A als Abkürzung für die Klausel $C \cup \{A\}$ zu verstehen. Wenn die Notationen S, C beziehungsweise C, A auf der linken Seite einer Vereinfachungsregel benutzt werden, soll zudem $C \notin S$ beziehungsweise $A \notin C$ gelten.

Die erste Regel eliminiert Doppelnegationen, die bei Formeln in Klauseln ganz oben auftreten. Die zweite Regel ersetzt eine konjunktive Formel in einer Klausel durch ihre zwei Konstituenten. Die dritte Regel eliminiert triviale Klauseln. Die vierte Regel unterscheidet sich von den bisherigen, da sie eine Klausel durch zwei Klauseln ersetzt. Sie basiert auf der Äquivalenz

$$A \wedge \neg(A_1 \wedge A_2) \mathrel{|\!\!\!\models\!\!\!|} (A \wedge \neg A_1) \vee (A \wedge \neg A_2)$$

Ohne die in Abbildung 9.6 verwendeten Abkürzungen können wir die disjunktiven Vereinfachungsregeln wie folgt definieren: Eine Klauselmenge S kann zu einer Klauselmenge

1. $(S - \{C\}) \cup \{(C - \{\neg\neg A\}) \cup \{A\}\}$ vereinfacht werden, wenn $C \in S$ und $\neg\neg A \in C$.
2. $(S - \{C\}) \cup \{(C - \{A_1 \wedge A_2\}) \cup \{A_1, A_2\}\}$ vereinfacht werden, wenn $C \in S$ und $A_1 \wedge A_2 \in C$.
3. $S - C$ vereinfacht werden, wenn $C \in S$ und C trivial ist.
4. $(S - \{C\}) \cup \{(C - \{\neg(A_1 \wedge A_2)\}) \cup \{\neg A_1\},\ (C - \{\neg(A_1 \wedge A_2)\}) \cup \{\neg A_2\}\}$ vereinfacht werden, wenn $C \in S$ und $\neg(A_1 \wedge A_2) \in C$.

Wir schreiben $S \stackrel{\mathrm{d}}{\to} S'$ genau dann, wenn die Klauselmenge S' aus der Klauselmenge S durch die Anwendung einer disjunktiven Vereinfachungsregel erhalten werden kann. Die wesentlichen Eigenschaften der disjunktiven Vereinfachungsregeln werden durch den folgenden Satz formuliert.

Satz 9.6.1 (Disjunktive Vereinfachung) *Sei S eine Klauselmenge. Dann:*

1. *Wenn $S \stackrel{d}{\to} S'$, dann $\mathcal{D}[\![S]\!] = \mathcal{D}[\![S']\!]$.*
2. *S ist normal genau dann, wenn es kein S' mit $S \stackrel{d}{\to} S'$ gibt.*
3. *Wenn S endlich ist, dann gibt es keine unendliche Kette*

$$S \stackrel{d}{\to} S_1 \stackrel{d}{\to} S_2 \stackrel{d}{\to} S_3 \stackrel{d}{\to} \cdots$$

Beweis Die ersten zwei Behauptungen des Satzes sind leicht zu verifizieren. Für die dritte Behauptung machen wir uns klar, dass die Vereinfachungsregeln eine Klausel entweder eliminieren oder durch ein oder zwei echt kleinere Klauseln ersetzen. Als Größe einer Klausel wählen wir

$$|C| = \sum_{A \in C} |A|$$

wobei die Größe einer Formel rekursiv wie folgt definiert ist:

$$\begin{aligned} |X| &= 1 \\ |\neg A| &= 1 + |A| \\ |A_1 \wedge A_2| &= 1 + |A_1| + |A_2| \end{aligned}$$ □

Für die Terminierung der Vereinfachungsregeln (Teil (3) des Satzes) haben wir ein Argument verwendet, dass eine explizite Formulierung verdient:

Lemma 9.6.2 (Terminierung) *Sei M eine Menge, $\gamma \in M \to \mathbb{N}$ und $\succ \subseteq \mathcal{P}_{fin}(M) \times \mathcal{P}_{fin}(M)$. Wenn für alle $U, V \in \mathcal{P}_{fin}(M)$*

$$U \succ V \;\Rightarrow\; \exists u \in U - V\ \forall v \in V - U\colon\ \gamma(u) > \gamma(v)$$

gilt, dann gibt es keine unendliche Kette $U_1 \succ U_2 \succ U_3 \succ \cdots$.

Beweis Es erfordert etwas Aufwand, das Lemma ausgehend von einfacheren Tatsachen zu beweisen. Das Lemma folgt aus einem Ergebnis für wohlfundierte Ordnungen auf Multimengen (siehe z.B. [Baader/Nipkow, Kapitel 2.5]). □

Der obige Satz liefert ein Entscheidungsverfahren für die Unerfüllbarkeit von Formeln. Gegeben eine Formel A, wenden wir auf $\{\{A\}\}$ solange disjunktive Vereinfachungsregeln an, bis wir eine normale Klauselmenge S erreichen. Dann ist A genau dann unerfüllbar, wenn S leer ist.

Wir schreiben $S \stackrel{d}{\to}^* S'$ genau dann, wenn es eine Kette wie folgt gibt ($n \geq 1$):

$$S = S_1 \;\stackrel{d}{\to}\; S_2 \;\stackrel{d}{\to}\; \ldots \;\stackrel{d}{\to}\; S_n = S'$$

Korollar 9.6.3 *Sei $A \in For$. Dann: A unerfüllbar* $\iff \{\{A\}\} \xrightarrow{\mathrm{d}}{}^* \emptyset$.

Korollar 9.6.4 *Zu jeder Formel existiert mindestens eine disjunktive Normalform.*

Die Vereinfachungsregeln lassen sicht leicht anpassen, wenn man die interne Syntax ändert oder erweitert. Allerdings muss die interne Syntax stets Negation enthalten. Für Disjunktion bekommen wir beispielsweise die folgenden Vereinfachungsregeln:

$$\begin{aligned} S,\ (C,\ A_1 \vee A_2) &\ \stackrel{\mathrm{d}}{\to}\ S,\ (C,\ A_1),\ (C,\ A_2) \\ S,\ (C,\ \neg(A_1 \vee A_2)) &\ \stackrel{\mathrm{d}}{\to}\ S,\ (C,\ \neg A_1,\ \neg A_2) \end{aligned}$$

9.7 Tableau-Notation

Wir wollen jetzt genauer verstehen, wie die disjunktiven Vereinfachungsregeln arbeiten. Zunächst unterscheiden wir zwischen den *einfachen Regeln* (die ersten drei) und der *verzweigenden Regel* (die vierte). Die verzweigende Regel ist die einzige Regel, die die Anzahl der Klauseln erhöhen kann. Für die meisten Klauselmengen führt die verzweigende Regel zu einem explosiven Anwachsen der Klauseln. Wenn man disjunktive Vereinfachung effizient realisieren will, wird man die verzweigende Regel nur dann anwenden, wenn keine einfache Regel mehr anwendbar ist.

Ohne notationale Tricks ist es mühsam, eine konkrete disjunktive Vereinfachungskette aufzuschreiben. Die sogenannte Tableau-Notation erweist sich auch für größere Beispiele als handhabbar.[1]

Als Beispiel betrachten wir eine Formel A wie folgt

$$(X \vee Z) \wedge ((\neg Z \vee \neg X) \vee \neg Y) \wedge ((\neg Y \wedge X) \vee \neg Z) \wedge (\neg X \vee (Z \wedge Y))$$

und nehmen an, dass die interne Syntax neben Negation und Konjunktion auch Disjunktion zur Verfügung stellt. Das Tableau in Abbildung 9.7 zeigt die Essenz einer disjunktiven Vereinfachungskette $\{\{A\}\} \xrightarrow{\mathrm{d}}{}^* \emptyset$. Das Tableau ist als Baum organisiert. Jede Verzweigung des Baumes repräsentiert eine Anwendung der verzweigenden Regel. Die Anwendung der einfachen Regeln erfolgt stillschweigend und ist im Tableau nicht vermerkt.

[1] Die Tableau-Notation geht auf Hintikka (1955) zurück und wird auch von Schütte (1956), Beth (1959) und Smullyan (1968) benutzt.

(1) $X \vee Z$
(2) $(\neg Z \vee \neg X) \vee \neg Y$
(3) $(\neg Y \wedge X) \vee \neg Z$
(4) $\neg X \vee (Z \wedge Y)$

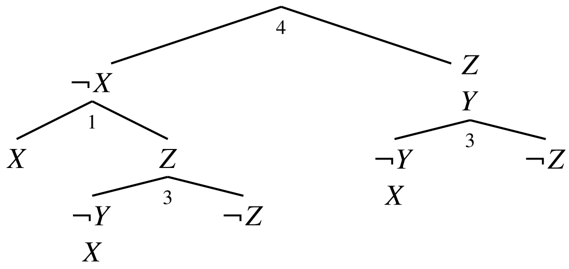


Abbildung 9.7: Ein geschlossenes Tableau

Die Wurzel des Baumes ist mit den Formeln markiert, die man aus $\{\{A\}\}$ durch Anwendung der einfachen Regeln erhält. Die erste Anwendung der verzweigenden Regel eliminiert die disjunktive Formel (4). Durch die Baumrepräsentation ist es möglich, den gemeinsamen Teil der zwei neuen Klauseln nur einmal zu repräsentieren (das ist die entscheidende Idee der Tableau-Notation). Auf der rechten Seite wurde die Konjunktion sofort durch die Anwendung der entsprechenden einfachen Regel eliminiert.

Die maximalen Pfade des durch das Tableau dargestellen Baumes (das sind die Pfade von der Wurzel zu den Blättern) repräsentieren Klauseln. Offensichtlich können wir die entsprechende Klauselmenge von $\{\{A\}\}$ ausgehend mithilfe der disjunktiven Vereinfachungsregeln erreichen. Da auf jedem Pfad ein komplementäres Paar von Formeln liegt, enthält diese Klauselmenge nur triviale Klauseln. Also können wir $\{\{A\}\}$ sogar zu der leeren Klauselmenge vereinfachen. Folglich ist A unerfüllbar.

Ein Tableau heißt *geschlossen* genau dann, wenn auf jedem maximalen Pfad ein komplementäres Paar von Formeln liegt (so wie im Beispiel). Ein geschlossenes Tableau für eine Ausgangsformel A repräsentiert eine disjunktive Vereinfachungskette $\{\{A\}\} \overset{\mathrm{d}}{\rightarrow}{}^{*} \emptyset$ und folglich einen Beweis dafür, dass A unerfüllbar ist.

$$
\begin{array}{lrcl}
(1) & S,\ (C,\ \neg\neg A) & \stackrel{\text{k}}{\rightarrow} & S,\ (C,\ A) \\
(2) & S,\ (C,\ \neg(A_1 \wedge A_2)) & \stackrel{\text{k}}{\rightarrow} & S,\ (C,\ \neg A_1,\ \neg A_2) \\
(3) & S,\ (C,\ A,\ \neg A) & \stackrel{\text{k}}{\rightarrow} & S \\
(4) & S,\ (C,\ A_1 \wedge A_2) & \stackrel{\text{k}}{\rightarrow} & S,\ (C,\ A_1),\ (C,\ A_2)
\end{array}
$$

Abbildung 9.8: Die konjunktiven Vereinfachungsregeln

9.8 Konjunktive Vereinfachung

Die konjunktiven Vereinfachungsregeln sind symmetrisch zu den disjunktiven Vereinfachungsregeln. Sie sind in Abbildung 9.8 angegeben.

Wir schreiben $S \stackrel{\text{k}}{\rightarrow} S'$ genau dann, wenn die Klauselmenge S' aus der Klauselmenge S durch die Anwendung einer konjunktiven Vereinfachungsregel erhalten werden kann. Die wesentlichen Eigenschaften der konjunktiven Vereinfachungsregeln werden durch den folgenden Satz formuliert.

Satz 9.8.1 (Konjunktive Vereinfachung) *Sei S eine Klauselmenge. Dann:*

1. *Wenn $S \stackrel{\text{k}}{\rightarrow} S'$, dann $\mathcal{K}[\![S]\!] = \mathcal{K}[\![S']\!]$.*
2. *S ist normal genau dann, wenn es kein S' mit $S \stackrel{\text{k}}{\rightarrow} S'$ gibt.*
3. *Wenn S endlich ist, dann gibt es keine unendliche Kette*

$$S \stackrel{\text{k}}{\rightarrow} S_1 \stackrel{\text{k}}{\rightarrow} S_2 \stackrel{\text{k}}{\rightarrow} S_3 \stackrel{\text{k}}{\rightarrow} \cdots$$

Beweis Analog zu dem Satz über disjunktive Vereinfachung. □

Der obige Satz liefert ein Entscheidungsverfahren für die Gültigkei von Formeln. Gegeben eine Formel A, wenden wir auf $\{\{A\}\}$ solange konjunktive Vereinfachungsregeln an, bis wir eine normale Klauselmenge S erreichen. Dann ist A genau dann gültig, wenn S leer ist.

Wir schreiben $S \stackrel{\text{k}}{\rightarrow}^* S'$ genau dann, wenn es eine Kette wie folgt gibt ($n \geq 1$):

$$S = S_1 \stackrel{\text{d}}{\rightarrow} S_2 \stackrel{\text{d}}{\rightarrow} \ldots \stackrel{\text{d}}{\rightarrow} S_n = S'$$

Korollar 9.8.2 *Sei $A \in For$. Dann: A gültig $\iff$ $\{\{A\}\} \stackrel{\text{k}}{\rightarrow}^* \emptyset$.*

Korollar 9.8.3 *Zu jeder Formel existiert mindestens eine konjunktive Normalform.*

Die Tableau-Notation kann natürlich auch für konjunktive Vereinfachungsketten verwendet werden.

9.9 Sequenten

Mit den Vereinfachungsregeln für Klauseln kann man bis auf Negation alle interne Syntax eliminieren. Wenn wir jede Klausel in einen postiven und negativen Teil aufteilen und die Formel im negativen Teil implizit mit einer Negation versehen, können wir auch interne Negationen elimieren. Diese Idee geht auf Gentzen zurück und liefert sehr elegante Vereinfachungsregeln.

Zweigeteilte Klauseln nennt man *Sequenten*. Sequenten können wir als ein Paar $\langle C, D\rangle$ aus zwei Klauseln formalisieren.

Für Sequenten betrachten wir hier nur die konjunktiven Vereinfachungsregeln.[2] Das bedeutet, dass die Formeln eines Sequenten disjunktiv verknüpft sind. Als den negativen Teil eines Sequenten wählen wir die linke Klausel. Die Denotation eines Sequenten

$$\langle \{A_1, \ldots, A_m\}, \{B_1, \ldots, B_n\}\rangle$$

ist dann gleich der Denotation der Formel

$$\neg A_1 \vee \cdots \vee \neg A_m \vee B_1 \vee \cdots \vee B_n$$

Wir können die Denotation des Sequenten also auch durch die Formel

$$A_1 \wedge \cdots \wedge A_m \;\Rightarrow\; B_1 \vee \cdots \vee B_n$$

beschreiben. Dieser Darstellung entsprechend schreibt man für einen Sequenten $\langle C, D\rangle$ meistens $(C \Rightarrow D)$. Diese Notation macht explizit, dass der negative Teil des Sequenten links steht.

Abbildung 9.9 definiert die Syntax und Semantik von Sequenten entsprechend den obigen Ausführungen.

Proposition 9.9.1 $\forall A \in For\colon\; \mathcal{M}[\![A]\!] = \mathcal{S}[\![\{(\emptyset \Rightarrow \{A\})\}]\!]$.

Proposition 9.9.2 (Leere Sequentenmenge) $\mathcal{S}[\![\emptyset]\!] = \Sigma$.

[2] Natürlich kann man genausogut die disjunktive Vereinfachungsregeln betrachten.

$$
\begin{array}{rcll}
C, D & \in & Cla = \mathcal{P}_{fin}(For) & \textit{Klausel} \\
(C \Rightarrow D) & \in & Cla \times Cla & \textit{Sequent} \\
G & \in & \mathcal{P}(Cla \times Cla) & \textit{Sequentenmenge}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S} \in \mathcal{P}(Cla \times Cla) \to \mathcal{P}(\Sigma) \\
\mathcal{S}[\![G]\!] = \{\sigma \in \Sigma \mid \forall (C \Rightarrow D) \in G: \\
\qquad\qquad (\forall A \in C\colon\ \sigma \in \mathcal{M}[\![A]\!]) \;\Rightarrow\; (\exists A \in D\colon\ \sigma \in \mathcal{M}[\![A]\!])\,\}
\end{array}
$$

Abbildung 9.9: Syntax und Semantik von Sequenten

Proposition 9.9.3 (Leerer Sequent) *Sei* $(\emptyset \Rightarrow \emptyset) \in G \subseteq Cla \times Cla$. *Dann* $\mathcal{S}[\![G]\!] = \emptyset$.

Ein Sequent $(C \Rightarrow D)$ heißt *trivial* genau dann, wenn $C \cap D \neq \emptyset$.

Proposition 9.9.4 (Triviale Sequenten) *Sei* G *eine Sequentenmenge und* $(C \Rightarrow D)$ *ein trivialer Sequent. Dann* $\mathcal{S}[\![G - \{(C \Rightarrow D)\}]\!] = \mathcal{S}[\![G]\!]$.

Ein nichttrivialer Sequent $(C \Rightarrow D)$ heißt *normal* genau dann, wenn jede Formel in $C \cup D$ eine Variable ist. Eine Sequentenmenge G heißt *normal* genau dann, wenn jeder Sequent in G normal ist.

Proposition 9.9.5 *Für jede normale Sequentenmenge* G *gilt:*

$$\mathcal{S}[\![G]\!] = \Sigma \iff G = \emptyset$$

Abbildung 9.10 zeigt die Vereinfachungsregeln für Sequenten. Beachten Sie, dass pro Boolescher Verknüpfung immer genau zwei Regeln benötigt werden, die die Verknüpfung links beziehungsweise rechts eliminieren.

Wir schreiben $G \xrightarrow{s} G'$ genau dann, wenn die Sequentenmenge G' aus der Sequentenmenge G durch die Anwendung einer Vereinfachungsregel erhalten werden kann. Die wesentlichen Eigenschaften der Vereinfachungsregeln für Sequenten werden durch den folgenden Satz formuliert.

Satz 9.9.6 (Vereinfachung von Sequenten) *Sei* G *eine Sequentenmenge. Dann:*

1. *Wenn* $G \xrightarrow{s} G'$, *dann* $\mathcal{S}[\![G]\!] = \mathcal{S}[\![G']\!]$.
2. G *ist normal genau dann, wenn es kein* G' *mit* $G \xrightarrow{s} G'$ *gibt.*

$$
\begin{array}{lrcl}
(T) & G,\ (C, A \Rightarrow D, A) & \xrightarrow{s} & G \\
(L\neg) & G,\ (C, \neg A \Rightarrow D) & \xrightarrow{s} & G,\ (C \Rightarrow D, A) \\
(R\neg) & G,\ (C \Rightarrow D, \neg A) & \xrightarrow{s} & G,\ (C, A \Rightarrow D) \\
(L\wedge) & G,\ (C, A_1 \wedge A_2 \Rightarrow D) & \xrightarrow{s} & G,\ (C, A_1, A_2 \Rightarrow D) \\
(R\wedge) & G,\ (C \Rightarrow D, A_1 \wedge A_2) & \xrightarrow{s} & G,\ (C \Rightarrow D, A_1),\ (C \Rightarrow D, A_2)
\end{array}
$$

Abbildung 9.10: Vereinfachungsregeln für Sequenten

3. *Wenn G endlich ist, dann gibt es keine unendliche Kette*

$$G \xrightarrow{s} G_1 \xrightarrow{s} G_2 \xrightarrow{s} G_3 \xrightarrow{s} \cdots$$

Beweis Analog zu dem Satz über disjunktive Vereinfachung von Klauseln. □

Der obige Satz liefert ein Entscheidungsverfahren für die Gültigkeit von Formeln. Gegeben eine Formel A, wenden wir auf $\{(\emptyset \Rightarrow \{A\})\}$ solange Vereinfachungsregeln an, bis wir eine normale Sequentenmenge G erreichen. Dann ist A gültig genau dann wenn G leer ist.

Wir schreiben $S \xrightarrow{s}{}^* S'$ genau dann, wenn es eine Kette wie folgt gibt ($n \geq 1$):

$$S = S_1 \xrightarrow{d} S_2 \xrightarrow{d} \dots \xrightarrow{d} S_n = S'$$

Korollar 9.9.7 *Sei $A \in For$. Dann: A gültig $\iff \{(\emptyset \Rightarrow \{A\})\} \xrightarrow{s}{}^* \emptyset$.*

Die Inferenzregeln in Abbildung 9.11 heißen Beweisregeln für Sequenten und definieren eine Menge von Sequenten. Die Beweisregeln entsprechen genau den Vereinfachungsregeln für Sequenten, wobei die linke Seite einer Vereinfachungsregel der Konklusion und die rechte Seite den Prämissen der Beweisregel entspricht. Die Beweisregeln sind also umgedrehte Vereinfachungsregeln, die alle gültigen Sequenten generieren.

Proposition 9.9.8 (Korrektheit) *Sei $\vdash (C \Rightarrow D)$. Dann $\mathcal{S}[\![\{(C \Rightarrow D)\}]\!] = \Sigma$.*

Beweis Durch Regelinduktion. □

Lemma 9.9.9 *Sei $G_1 \xrightarrow{s} \dots \xrightarrow{s} G_n$ und $G_n = \emptyset$. Dann gilt $\vdash (C \Rightarrow D)$ für alle $(C \Rightarrow D) \in G_1$.*

Beweis Durch Induktion über n. □

Satz 9.9.10 (Vollständigkeit) *Sei $\mathcal{S}[\![\{(C \Rightarrow D)\}]\!] = \Sigma$. Dann $\vdash (C \Rightarrow D)$.*

$$\frac{C \cap D \neq \emptyset}{\vdash (C \Rightarrow D)}$$

$$\frac{\vdash (C \Rightarrow D, A)}{\vdash (C, \neg A \Rightarrow D)} \qquad \frac{\vdash (C, A \Rightarrow D)}{\vdash (C \Rightarrow D, \neg A)}$$

$$\frac{\vdash (C, A_1, A_2 \Rightarrow D)}{\vdash (C, A_1 \wedge A_2 \Rightarrow D)} \qquad \frac{\vdash (C \Rightarrow D, A_1) \qquad \vdash (C \Rightarrow D, A_2)}{\vdash (C \Rightarrow D, A_1 \wedge A_2)}$$

Abbildung 9.11: Beweisregeln für Sequenten

$$\frac{C \in S}{C \in Res(S)} \qquad \frac{A \in C_1 \in Res(S) \qquad \neg A \in C_2 \in Res(S)}{(C_1 - \{A\}) \cup (C_2 - \{\neg A\}) \in Res(S)}$$

Abbildung 9.12: Resolutionsabschluss $Res(S)$ einer Klauselmenge S

Beweis Korollar 9.9.7 liefert eine Kette

$$\{(C \Rightarrow D)\} = G_1 \xrightarrow{s} \ldots \xrightarrow{s} G_n = \emptyset$$

Damit folgt $\vdash (C \Rightarrow D)$ mit dem obigen Lemma. □

9.10 Resolution

Resolution ist eine Methode, mit der man für endliche und normale Klauselmengen S entscheiden kann, ob $\mathcal{D}[\![S]\!] = \Sigma$ oder $\mathcal{K}[\![S]\!] = \emptyset$ gilt. Resolution erweitert eine Klauselmenge solange um neue Klauseln (sogenannte Resolventen), bis entweder die leere Klausel generiert wird oder keine neue Klausel mehr hinzugefügt werden kann.

Abbildung 9.12 definiert mithilfe zweier Inferenzregeln den *Resolutionsabschluss* $Res(S)$ einer Klauselmenge S. Die durch die zweite Inferenzregel eingeführten Klauseln heißen *Resolventen*.

Proposition 9.10.1 *Sei S eine Klauselmenge. Dann $\mathcal{D}[\![S]\!] = \mathcal{D}[\![Res(S)]\!]$ und $\mathcal{K}[\![S]\!] = \mathcal{K}[\![Res(S)]\!]$.*

Beweis Die Behauptung folgt aus der Gültigkeit der Formeln:

$$\begin{array}{ll} (1) & (A \wedge B_1) \vee (\neg A \wedge B_2) \Leftarrow B_1 \wedge B_2 \\ (2) & (A \vee B_1) \wedge (\neg A \vee B_2) \Rightarrow B_1 \vee B_2 \end{array}$$

□

Lemma 9.10.2 *Sei S eine Klauselmenge, $A \in For$ und*

$$D \in Res(\{\, C - \{A\} \mid C \in S \,\})$$

Dann $D \in Res(S)$ oder $D \cup \{A\} \in Res(S)$.

Beweis Durch Regelinduktion über $Res(\{\, C - \{A\} \mid C \in S \,\})$. □

Lemma 9.10.3 *Sei S eine endliche und literale Klauselmenge mit $\mathcal{K}[\![S]\!] = \emptyset$. Dann $\emptyset \in Res(S)$.*

Beweis Durch Induktion über die Anzahl n der in S vorkommenden Variablen.

Sei $n = 0$. Die leere Klausel ist die einzige Klausel, die keine Variablen enthält. Da $\mathcal{K}[\![\emptyset]\!] = \Sigma$, folgt $S \neq \emptyset$. Also $\emptyset \in S$. Also $\emptyset \in Res(S)$.

Sei $n > 0$. Sei X eine Variable, die in S vorkommt. Wir definieren S_1 und S_2 wie folgt:

$$S_1 = \{\, C - \{X\} \mid C \in S,\ \neg X \notin C \,\}$$
$$S_2 = \{\, C - \{\neg X\} \mid C \in S,\ X \notin C \,\}$$

Dann $\mathcal{K}[\![S_1]\!] = \mathcal{K}[\![S_2]\!] = \emptyset$. Also gilt nach Induktionsannahme $\emptyset \in Res(S_1)$ und $\emptyset \in Res(S_2)$. Mit Lemma 9.10.2 folgt

$$\emptyset \in Res(S) \vee (\{X\} \in Res(S) \wedge \{\neg X\} \in Res(S))$$

Also $\emptyset \in Res(S)$. □

Lemma 9.10.4 *Sei S eine endliche und literale Klauselmenge mit $\mathcal{D}[\![S]\!] = \Sigma$. Dann $\emptyset \in Res(S)$.*

Beweis Analog zu Lemma 9.10.3. □

Satz 9.10.5 (Resolution) *Sei S eine endliche und literale Klauselmenge. Dann*

$$\mathcal{D}[\![S]\!] = \Sigma \iff \mathcal{K}[\![S]\!] = \emptyset \iff \emptyset \in Res(S)$$

Beweis Proposition 9.10.1 und Lemmata 9.10.3 und 9.10.4. □

Abbildung 9.13 zeigt eine Graphdarstellung des Ableitungsbaums (siehe Kapitel 2) für $\emptyset \in Res(S)$ mit $S = \{\{\neg X, Y\}, \{X, Z\}, \{X, \neg Z\}, \{\neg X, Z\}, \{\neg Y, \neg Z\}\}$.

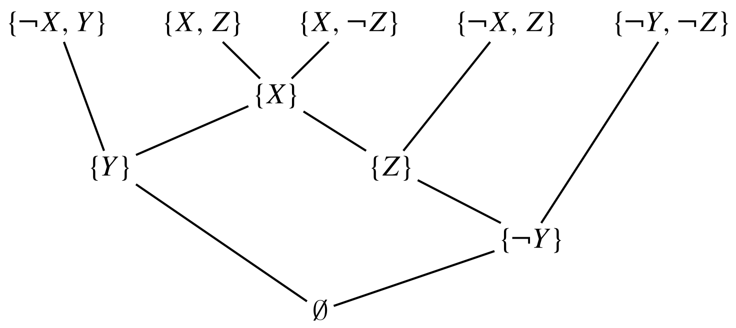


Abbildung 9.13: Ein Resolutionsgraph

Man spricht von einem *Resolutionsgraphen* für S. Dieser Resolutionsgraph ist ein Beweis für $\mathcal{D}[\![S]\!] = \Sigma$ und $\mathcal{K}[\![S]\!] = \emptyset$.

Offensichtlich können die Klauseln in $Res(S)$ nur Formeln enthalten, die in mindestens einer Klausel von S enthalten sind. Also haben wir:

Proposition 9.10.6 *Wenn S eine endliche Klauselmenge ist, dann ist $Res(S)$ ebenfalls endlich.*

Also können wir den Resolutionsabschluss einer endlichen Menge berechnen. Damit liefert uns Resolution für literale und endliche Klauselmengen S ein Entscheidungsverfahren für $\mathcal{D}[\![S]\!] = \Sigma$ und $\mathcal{K}[\![S]\!] = \emptyset$.

9.11 Primdarstellungen

Eine endliche und normale Klauselmenge S heißt *Primdarstellung* genau dann, wenn $S = Ber(Res(S))$.

Wir werden zeigen, dass zu jeder Formel A genau eine disjunktive beziehungsweise konjunktive Normalform existiert, die eine Primdarstellung ist. Wir sprechen von der disjunktiven beziehungsweise konjunktiven Primdarstellung von A. Wir werden zeigen, dass zwei Formeln genau dann äquivalent sind, wenn sie dieselbe disjunktive beziehungsweise konjunktive Primdarstellung haben. Die zwei Primdarstellungen einer Formel kann man mit Resolution bestimmen. Primdarstellungen haben wichtige Anwendungen beim Schaltkreisentwurf und in der Künstlichen Intelligenz.[3]

[3] Es scheint noch kein Logik-Lehrbuch zu geben, in dem Primdarstellungen behandelt werden. Eine interessante Arbeit über die Berechnung von Primdarstellungen ist: Alex Kean und Ge-

Um eine von Resolution unabhängige Charakterisierung von Primdarstellungen angeben zu können, benötigen wir eine stärkere Version des Resolutionssatzes 9.10.5.

Sei L ein Literal. Mit $\overline{L}$ bezeichnen wir das zu L komplementäre Literal. Für jedes Literal L gilt $\overline{L} = \neg L$ oder $\neg\overline{L} = L$.

Lemma 9.11.1 *Sei S eine Klauselmenge und C eine normale Klausel. Weiter sei*

$$D \in Res(S \cup \{\,\{\overline{L}\} \mid L \in C\,\}) - \{\,\{\overline{L}\} \mid L \in C\,\}$$

Dann existiert $C' \subseteq C$ mit $D \cup C' \in Res(S)$.

Beweis Durch Regelinduktion über $Res(S \cup \{\,\{\overline{L}\} \mid L \in C\,\})$. □

Satz 9.11.2 (Resolution) *Sei S eine endliche und literale Klauselmenge und C eine normale Klausel. Dann:*

$$\mathcal{D}[\![\{C\}]\!] \subseteq \mathcal{D}[\![S]\!] \iff \mathcal{K}[\![S]\!] \subseteq \mathcal{K}[\![\{C\}]\!] \iff \exists C' \in Res(S)\colon\; C' \subseteq C$$

Beweis Da $\mathcal{D}[\![S]\!] = \mathcal{D}[\![Res(S)]\!]$ und $\mathcal{K}[\![S]\!] = \mathcal{K}[\![Res(S)]\!]$, folgt aus der rechten Aussage die linke und die mittlere Aussage. Wir zeigen, dass aus der mittleren Aussage die rechte Aussage folgt. Analog kann man zeigen, dass aus der linken Aussage die rechte Aussage folgt.

Sei $\mathcal{K}[\![S]\!] \subseteq \mathcal{K}[\![\{C\}]\!]$. Dann $\mathcal{K}[\![S \cup \{\,\{\overline{L}\} \mid L \in C\,\}]\!] = \emptyset$. Also folgt $\emptyset \in Res(S \cup \{\,\{\overline{L}\} \mid L \in C\,\})$ mit dem ersten Resolutionssatz. Also existiert mit Lemma 9.11.1 ein $C' \subseteq C$ mit $C' \in Res(S)$. □

Korollar 9.11.3 *Sei S eine endliche und literale Klauselmenge. Dann*

$$\begin{aligned} Ber(Res(S)) &= Ber(\{\, C \in Cla \mid C \text{ normal und } \mathcal{D}[\![\{C\}]\!]\,\} \subseteq \mathcal{D}[\![S]\!]) \\ &= Ber(\{\, C \in Cla \mid C \text{ normal und } \mathcal{K}[\![S]\!] \subseteq \mathcal{K}[\![\{C\}]\!]\,\}) \end{aligned}$$

Korollar 9.11.4 *Seien S und S' endliche und literale Klauselmengen. Dann:*

$$\mathcal{D}[\![S]\!] = \mathcal{D}[\![S']\!] \iff \mathcal{K}[\![S]\!] = \mathcal{K}[\![S']\!] \iff Ber(Res(S)) = Ber(Res(S'))$$

Beweis Folgt aus Korollar 9.11.3 und der Tatsache, dass Bereinigung und Resolution denotationserhaltend sind. □

Korollar 9.11.5 *Sei S eine endliche und literale Klauselmenge. Dann ist $Ber(Res(S))$ eine Primdarstellung.*

orge Tsiknis, An Incremental Method for Generating Prime Implicants/Implicates. Journal of Symbolic Computation (1990) 9, 185–206.

Beweis Folgt aus Korollar 9.11.4 und der Tatsache, dass Bereinigung und Resolution denotationserhaltend sind. □

Die *disjunktive und die konjunktive Primdarstellung* einer Formel A definieren wir wie folgt:

$$DPD(A) \stackrel{\text{def}}{=} Ber(\{\, C \in Cla \mid C \text{ normal und } \mathcal{D}[\![\{C\}]\!] \subseteq \mathcal{M}[\![A]\!] \,\})$$

$$KPD(A) \stackrel{\text{def}}{=} Ber(\{\, C \in Cla \mid C \text{ normal und } \mathcal{M}[\![A]\!] \subseteq \mathcal{K}[\![\{C\}]\!] \,\})$$

Die Klauseln in $DPD(A)$ nennt man die *Primimplikanten* von A, und die Klauseln in $KPD(A)$ die *Primimplikate* von A.

Satz 9.11.6 (Primdarstellung) *Seien A und B Formeln. Dann:*

1. *Für jede disjunktive Normalform S von A gilt:* $DPD(A) = Ber(Res(S))$.
2. *Für jede konjunktive Normalform S von A gilt:* $KPD(A) = Ber(Res(S))$.
3. $\mathcal{M}[\![A]\!] = \mathcal{D}[\![DPD(A)]\!] = \mathcal{K}[\![KPD(A)]\!]$.
4. $DPD(A)$ *ist eine disjunktive Normalform von A und* $KPD(A)$ *ist eine konjunktive Normalform von A.*
5. $A \models\!\mid B \iff DPD(A) = DPD(B) \iff KPD(A) = KPD(B)$.
6. *A gültig* $\iff DPD(A) = \{\emptyset\} \iff KPD(A) = \emptyset$.
7. *A unerfüllbar* $\iff DPD(A) = \emptyset \iff KPD(A) = \{\emptyset\}$.
8. *Für jede disjunktive Normalform S von A gilt:* $Ber(DPD(A) \cup S) = DPD(A)$.
9. *Für jede konjunktive Normalform S von A gilt:* $Ber(KPD(A) \cup S) = KPD(A)$.

Beweis Folgt aus Korollar 9.11.3 und der Tatsache, dass Bereinigung und Resolution denotationserhaltend sind. □

Wir sind jetzt in der Lage, die zwei Primdarstellungen einer Formel zu berechnen (Primdarstellungssatz (1) und (2)). Als Beispiel bestimmen wir die konjunktive Primdarstellung einer Formel A wie folgt:

$$(\neg X \Rightarrow Z) \;\wedge\; (Z \wedge X \Rightarrow \neg Y) \;\wedge\; (Y \vee \neg X \Rightarrow \neg Z)$$

Diese Formel entspricht der Formel in Abschnitt 9.2, wobei wir jetzt allerdings die Buchstaben X, Y und Z anstelle von B, E und F für die Variablen benutzen. Die obige Formel ist äquivalent zu

$$(X \vee Z) \;\wedge\; (\neg Z \vee \neg X \vee \neg Y) \;\wedge\; ((\neg Y \wedge X) \vee \neg Z)$$

Das sieht man schnellsten durch äquivalente Umformung von Teilformeln (Ersetzungssatz). Durch Ausnützung der Distributivität bekommen wir die äquivalente Formel

$$(X \vee Z) \;\wedge\; (\neg Z \vee \neg X \vee \neg Y) \;\wedge\; (\neg Y \vee \neg Z) \;\wedge\; (X \vee \neg Z)$$

Also ist Klauselmenge

$$\{\{X, Z\},\ \{\neg X, \neg Y, \neg Z\},\ \{\neg Y, \neg Z\},\ \{X, \neg Z\}\}$$

eine konjunktive Normalform für A. Das gilt auch für die Klauselmenge

$$\{\{X, Z\},\ \{\neg Y, \neg Z\},\ \{X, \neg Z\}\}$$

die wir durch Streichen der subsumierten Klausel $\{\neg X, \neg Y, \neg Z\}$ erhalten. Wenn wir die Resolvente $\{X\}$ hinzufügen, bekommen wir eine weiter konjunktive Normalform für A:

$$\{\{X\},\ \{X, Z\},\ \{\neg Y, \neg Z\},\ \{X, \neg Z\}\}$$

Jetzt löschen wir alle von $\{X\}$ subsumierte Klauseln und bekommen eine konjunktive Normalform S für A:

$$\{\{X\},\ \{\neg Y, \neg Z\}\}$$

Da $S = Ber(Res(S))$, ist S die konjunktive Primdarstellung von A.

Sei A eine Formel. Die konjunktive Primdarstellung von A ist nicht notwendigerweise eine größenmäßig minimale konjunktive Normalform für A. Aus dem Primdarstellungssatz (9) folgt aber immerhin, dass jede größenmäßig minimale konjunktive Normalform für A eine Teilmenge der konjunktiven Primdarstellung von A ist. Im Allgemeinen gibt es jedoch mehr als eine größenmäßig minimale konjunktive Normalform für A. Betrachten Sie dazu die Klauselmenge

$$S = \{\{\neg X, Z\},\ \{\neg Y, Z\},\ \{\neg X, Y\},\ \{Y, \neg Z\},\ \{X, \neg Z\},\ \{X, \neg Y\}\}$$

Da $S = Ber(Res(S))$ gilt, ist S eine konjunktive Primdarstellung einer geeignet gewählten Formel A. Betrachten Sie nun die beiden Teilmengen

$$S_1 = \{\{\neg Y, Z\},\ \{\neg X, Y\},\ \{X, \neg Z\}\}$$
$$S_2 = \{\{\neg X, Z\},\ \{Y, \neg Z\},\ \{X, \neg Y\}\}$$

Da $Res(S) = Res(S_1) = Res(S_2)$ gilt, sind S_1 und S_2 konjunktive Normalformen von A, die kleiner sind als S. Man überlegt sich leicht, dass S_1 und S_2 beides größenmäßig minimale konjunktive Normalformen von A sind.

9.12 Kompaktheit

Sei $\sigma \in \Sigma$ und $M \subseteq For$. Wir definieren:

$$\sigma \text{ erfüllt } M \overset{\text{def}}{\iff} \sigma \models M \overset{\text{def}}{\iff} \forall A \in M\colon\ \sigma \models A$$

Eine Menge $M \subseteq For$ heißt *erfüllbar* genau dann, wenn es ein $\sigma \in \Sigma$ gibt, das M erfüllt. Eine Menge $M \subseteq For$ heißt *unerfüllbar* genau dann, wenn sie nicht erfüllbar ist.

Wir werden jetzt zeigen, dass es zu einer unerfüllbaren Menge $M \subseteq For$ stets eine endliche Teilmenge gibt, die unerfüllbar ist. Dieses Ergebnis ist als Kompaktheitseigenschaft der Aussagenlogik bekannt und wird für die Beweise einiger wichtiger prädikatenlogischer Sätze benötigt.

Da es bei der Kompaktheitseigenschaft offensichtlich um die Unerfüllbarkeit unendlicher Formelmengen geht, benötigen wir neue Beweistechniken. Unser Beweis der Kompaktheitseigenschaft folgt [Fitting, Kapitel 3.6]. In [Schöning, Kapitel 1.4] findet man einen konstruktiveren Beweis.

Eine *Hintikka-Menge* ist eine Menge $M \subseteq For$, für die die folgenden Eigenschaften gelten:

1. Wenn $\neg A \in M$, dann $A \notin M$.
2. Wenn $\neg\neg A \in M$, dann $A \in M$.
3. Wenn $A_1 \wedge A_1 \in M$, dann $A_1 \in M$ und $A_2 \in M$.
4. Wenn $\neg(A_1 \wedge A_1) \in M$, dann $\neg A_1 \in M$ oder $\neg A_2 \in M$.

Proposition 9.12.1 *Jede Hintikka-Menge ist erfüllbar.*

Beweis Sei M eine Hintikka-Menge. Wir definieren eine Belegung

$$\sigma \overset{\text{def}}{=} \lambda X \in Var\,\textbf{.}\ \text{if } X \in M \text{ then } 1 \text{ else } 0$$

und zeigen durch Induktion über A: $\forall A \in For\colon\ A \in M \Rightarrow \mathcal{F}[\![A]\!]\sigma = 1$. $\square$

Eine *Konsistenzeigenschaft* ist eine Menge $\mathcal{C} \subseteq \mathcal{P}(For)$, so dass für jede Menge $M \in \mathcal{C}$ die folgenden Eigenschaften gelten:

1. Wenn $\neg A \in M$, dann $A \notin M$.
2. Wenn $\neg\neg A \in M$, dann $(M \cup \{A\}) \in \mathcal{C}$.
3. Wenn $A_1 \wedge A_1 \in M$, dann $(M \cup \{A_1, A_2\}) \in \mathcal{C}$.
4. Wenn $\neg(A_1 \wedge A_1) \in M$, dann $(M \cup \{\neg A_1)\} \in M$ oder $(M \cup \{\neg A_2)\} \in M$.

Proposition 9.12.2 *Jedes maximale Element einer Konsistenzeigenschaft ist eine Hintikka-Menge und folglich erfüllbar.*

Eine Konsistenzeigenschaft heißt *finitär* genau dann, wenn für alle $M \subseteq For$ gilt:

$$M \in \mathcal{C} \iff \mathcal{P}_{fin}(M) \subseteq \mathcal{C}$$

Die folgende Proposition sagt, dass jede finitäre Konsistenzeigenschaft unter Teilmengen und Vereinigung von aufsteigenden Ketten abgeschlossen ist.

Proposition 9.12.3 *Sei $\mathcal{C}$ eine finitäre Konsistenzeigenschaft. Dann gilt:*

1. $\forall M \in \mathcal{C}\ \forall M' \subseteq M:\ M' \in \mathcal{C}$.
2. *Für jede Kette $M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots$ in $\mathcal{C}$ ist $\bigcup_{i \in \mathbb{N}} M_i \in \mathcal{C}$.*

Jede Konsistenzeigenschaft kann zu einer finitären Konsistenzeigenschaft erweitert werden.

Proposition 9.12.4 *Zu jeder Konsistenzeigenschaft $\mathcal{C}$ existiert eine finitäre Konsistenzeigenschaft $\mathcal{C}'$ mit $\mathcal{C} \subseteq \mathcal{C}'$.*

Beweis Sei $\mathcal{C}$ eine Konsistenzeigenschaft. Wir definieren

$$\mathcal{C}' = \{\, M' \mid \exists M \in \mathcal{C}:\ M' \subseteq M \,\}$$

Dann ist $\mathcal{C}'$ eine unter beliebigen Teilmengen abgeschlossene Konsistenzeigenschaft mit $\mathcal{C} \subseteq \mathcal{C}'$. Als Nächstes definieren wir

$$\mathcal{C}'' = \{\, M \subseteq For \mid \mathcal{P}_{fin}(M) \subseteq \mathcal{C}' \,\}$$

$\mathcal{C}''$ erweitert $\mathcal{C}'$ genau um die unendlichen Mengen, deren endliche Teilmengen alle in $\mathcal{C}'$ sind. Man überzeugt sich leicht davon, dass $\mathcal{C}''$ eine finitäre Konsistenzeigenschaft ist. □

Annahme: Für den Rest des Kapitels nehmen wir an, dass *Var* abzählbar ist.

Proposition 9.12.5 *Sei $\mathcal{C}$ eine finitäre Konsistenzeigenschaft. Dann existiert zu jedem Element $M \in \mathcal{C}$ ein maximales Element $M' \in \mathcal{C}$ mit $M \subseteq M'$.*

Beweis Da *Var* abzählbar ist, ist auch *For* abzählbar (Gödelisierung). Sei $A_1, A_2, A_3, \ldots$ eine Aufzählung von *For*. Sei $M \in \mathcal{C}$. Wir definieren eine aufsteigende Kette

$$M = M_0 \subseteq M_1 \subseteq M_2 \subseteq M_3 \subseteq \cdots$$

in $\mathcal{C}$ wie folgt:

$$\begin{aligned} M_0 &= M \\ M_i &= \text{if } M_{i-1} \cup \{A_i\} \in \mathcal{C} \text{ then } M_{i-1} \cup \{A_i\} \text{ else } M_{i-1} \end{aligned}$$

Wegen Proposition 9.12.3 gilt $\bigcup_{i\in\mathbb{N}} M_i \in \mathcal{C}$. Man überlegt sich leicht, dass $\bigcup_{i\in\mathbb{N}} M_i$ sogar ein maximales Element von $\mathcal{C}$ ist. □

Satz 9.12.6 (Modellexistenz) *Jede Konsistenzeigenschaft enthält nur erfüllbare Mengen.*

Beweis Sei $\mathcal{C}$ eine Konsistenzeigenschaft und $M \in \mathcal{C}$. Mit Proposition 9.12.4 bekommen wir eine finitäre Konsistenzeigenschaft $\mathcal{C}'$ mit $M \in \mathcal{C} \subseteq \mathcal{C}'$. Mit Proposition 9.12.5 bekommen wir ein maximales Element $M' \in \mathcal{C}'$ mit $M \subseteq M'$. Mit Proposition 9.12.2 folgt jetzt, dass M' erfüllbar ist. Also ist M erfüllbar. □

Proposition 9.12.7 *Die Menge* $\{\, M \subseteq \mathit{For} \mid \forall F \in \mathcal{P}_{\mathit{fin}}(M){:}\;\; F \text{ erfüllbar} \,\}$ *ist eine Konsistenzeigenschaft.*

Satz 9.12.8 (Kompaktheit) *Eine Formelmenge ist erfüllbar genau dann, wenn jede ihrer endlichen Teilmengen erfüllbar ist.*

Beweis Folgt aus dem Modellexistenzsatz und Proposition 9.12.7. □

Korollar 9.12.9 *Eine Formelmenge ist unerfüllbar genau dann, wenn sie eine unerfüllbare endlichen Teilmenge hat.*

Satz 9.12.10 (Testbarkeit) *Sei Var eine entscheidbare Menge. Dann ist die Unerfüllbarkeit von testbaren Formelmengen testbar.*

Beweis Sei ein Tester für eine Formelmenge M gegeben. Aus dem Tester können wir einen Aufzähler für M konstruieren. Damit können wir eine Kette $M_0 \subseteq M_1 \subseteq \dots$ von endlichen Teilmengen von M aufzählen, so dass zu jeder endlichen Teilmenge M' von M ein $n \in \mathbb{N}$ existiert mit $S' \subseteq S_n$. Da die Unerfüllbarkeit der S_i entscheidbar ist, haben wir damit wegen Korollar 9.12.9 eine Prozedur, die genau dann terminiert, wenn M unerfüllbar ist. □

Korollar 9.12.11 *Sei S eine Klauselmenge. Dann $\mathcal{K}[\![S]\!] = \emptyset$ genau dann, wenn es eine endliche Teilmenge S' von S gibt, so dass $\mathcal{K}[\![S']\!] = \emptyset$.*

Satz 9.12.12 (Resolution) *Für jede literale Klauselmenge S gilt:*

$$\mathcal{K}[\![S]\!] = \emptyset \iff \emptyset \in \mathit{Res}(S)$$

Sei $M \subseteq \mathit{For}$ und $A \in \mathit{For}$. Wir definieren:

$$M \models A \overset{\text{def}}{\iff} (\forall \sigma \in \Sigma{:}\;\; \sigma \models M \Rightarrow \sigma \models A)$$

Korollar 9.12.13 *Sei $M \subseteq \mathit{For}$ und $A \in \mathit{For}$. Dann:*

$$M \models A \iff \exists M' \subseteq \mathcal{P}_{\mathit{fin}}(M){:}\;\; M' \models A$$

Kapitel 10

Prädikatenlogik

In diesem Kapitel behandeln wir die Familie der prädikatenlogischen Sprachen erster Ordnung. Mit ASSN haben wir bereits eine solche Sprache kennengelernt. Eine prädikatenlogische Sprache erster Ordnung ist durch eine Menge U (das sogenannte Universum, bei ASSN $\mathbb{Z}$) und durch eine Menge von Funktionen des Typs $U^n \to U$ oder $U^n \to \mathbb{B}$ gegeben. Die Funktionen des zweiten Typs nennt man Prädikate (ASSN hat das Prädikat $\leq$).

Lesematerial

[Schöning, Kapitel 2]

10.1 Syntax

Abbildung 10.1 definiert die Syntax von prädikatenlogischen Sprachen. Ausgangspunkt sind die drei Mengen *Var*, *Fun* und *Pre*, die die benötigten Symbole zur Verfügung stellen. Jedem Funktions- und Prädikatensymbol ist durch die Stelligkeitsfunktion eine feste Anzahl von Argumenten zuordnet. Nullstellige Funktionssymbole bezeichnet man auch als *Konstanten*. Aufbauend auf den Symbolen werden Terme und Formeln definiert. Die Funktion dieser syntaktischen Objekte ist aus ASSN bekannt.

Die Konstruktion der Symbole sorgt dafür, dass es unendlich viele Variablen und für jede Stelligkeit unendlich viele Funktions- und Prädikatensymbole gibt. Ausserdem sind die Mengen *Ter* und *For* gödelisierbar und entscheidbar.

Eine *atomare Formel* ist eine Formel der Form $p(t_1, \ldots, t_n)$.

Eine *Signatur* Σ ist eine nichtleere und entscheidbare Teilmenge von $Fun \cup Pre$.

x, y	$\in$	$Var = \mathbb{N}$		*Variable*
f, g	$\in$	$Fun = \{0\} \times \mathbb{N} \times \mathbb{N}$		*Funktionssymbol*
p, q	$\in$	$Pre = \{1\} \times \mathbb{N} \times \mathbb{N}$		*Prädikatensymbol*
		$\lvert \cdot \rvert \in Fun \cup Pre \to \mathbb{N}$		*Stelligkeit*
		$\lvert(k, m, n)\rvert = n$		
s, t	$\in$	$Ter =$		*Term*
		x		
		$\mid f(t_1, \ldots, t_{\lvert f \rvert})$		
A, B	$\in$	$For =$		*Formel*
		$p(t_1, \ldots, t_{\lvert p \rvert})$		*Atom*
		$\mid \neg A$		*Negation*
		$\mid A_1 \wedge A_2$		*Konjunktion*
		$\mid \exists x\, A$		*existenzielle Quantifizierung*

Abbildung 10.1: Prädikatenlogische Syntax

Ein Σ-*Term* ist ein Term, in dem neben Variablen nur Funktions- und Prädikatensymbole aus Σ vorkommen. Eine Σ-*Formel* ist eine Formel, in der neben Variablen nur Funktions- und Prädikatensymbole aus Σ vorkommen. Die Mengen aller Σ-Terme und Σ-Formeln bezeichnen wir mit Ter_Σ und For_Σ. Wenn M eine Menge syntaktischer Objekte ist, bezeichen wir mit M_Σ die größte Teilmenge von M, deren Objekte neben Variablen nur Funktions- und Prädikatensymbole aus Σ enthalten.

Die Syntax einer konkreten prädikatenlogischen Sprache wird durch eine Signatur festgelegt. Beispielsweise enthält eine Signatur für ASSN die folgenden Symbole:

1. Für jede ganze Zahl eine Konstante (also ein nullstelliges Funktionssymbol).
2. Für $+$ und $*$ je ein zweistelliges Funktionssymbol.
3. Für $\leq$ ein zweistelliges Prädikatensymbol.

Welche Symbole dabei genau gewählt werden spielt keine Rolle.

Beachten Sie, dass Abbildung 10.1 die abstrakte Syntax von prädikatenlogischen

$$\begin{array}{rcl} A_1 \vee A_2 & \mapsto & \neg(\neg A_1 \wedge \neg A_2) \\ A_1 \Rightarrow A_2 & \mapsto & \neg A_1 \vee A_2 \\ A_1 \Leftrightarrow A_2 & \mapsto & (A_1 \Rightarrow A_2) \wedge (A_2 \Rightarrow A_1) \\ \forall x A_1 & \mapsto & \neg\exists x \neg A \end{array}$$

Abbildung 10.2: Abkürzungen für prädikatenlogische Sprachen

Sprachen definiert. Für die konkrete Syntax verwendet man je nach Anwendung unterschiedliche Notationen. Eine Möglichkeit haben wir am Beispiel von ASSN gesehen.

Wenn s ein nullstelliges Funktions- oder Prädikatensymbol ist, werden wir im Folgenden stets s statt $s()$ schreiben. Für nullstellige Funktionssymbole verwenden wir in der Regel die Metavariablen a und b. Ausserdem verwenden wir die Abkürzungen in Abbildung 10.2. Damit wir nicht so viele Klammern schreiben müssen, priorisieren wir die logischen Operatoren wie folgt: $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. Damit können wir beispielsweise

$$(((\neg A \wedge B) \vee A) \Rightarrow (A \wedge B))) \Leftrightarrow (A \vee B)$$

ohne Klammern schreiben:

$$\neg A \wedge B \vee A \Rightarrow A \wedge B \Leftrightarrow A \vee B$$

Die *freien Variablen* $FV(A)$ einer Formel A sind analog zu ASSN definiert. Eine Formel heißt *geschlossen* genau dann, wenn sie keine freien Variablen hat.

Sei $A \in For$ eine Formel mit $FV(A) = \{x_1, \ldots, x_n\}$ und $x_1 < \cdots < x_n$.[1] Wir definieren:

$$\forall A \stackrel{\text{def}}{=} \forall x_1 \ldots \forall x_n A \qquad \textit{Allabschluss von } A$$

$$\exists A \stackrel{\text{def}}{=} \exists x_1 \ldots \exists x_n A \qquad \textit{Existenzabschluss von } A$$

10.2 Semantik

Die Semantik einer konkreten prädikatenlogische Sprache mit der Signatur Σ gibt man durch eine nichtleere Menge U (das sogenannte *Universum*) und durch Interpretationen für die Funktions- und Prädikatensymbole in Σ vor. Dabei muss ein

[1] Die Variablen $x_1, \ldots, x_n$ sind natürliche Zahlen.

n-stelliges Funktionssymbol durch eine Funktion in $U^n \to U$ und ein n-stelliges Prädikatensymbol durch eine Funktion in $U^n \to \mathbb{B}$ interpretiert werden. Die Menge U zusammen mit den Interpretationen für die Symbole in Σ bezeichnet man als eine *Struktur*. Der Ordnung halber wollen wir Strukturen noch explizit definieren.

Eine *Struktur* ist ist ein Paar $\langle U, I \rangle$ wie folgt:

- U ist eine nichtleere Menge.
- I ist eine Funktion.
- $Dom\ I$ ist eine Signatur.
- Für alle $f \in Dom\ I \cap Fun$ ist $I(f) \in U^{|f|} \to U$.
- Für alle $p \in Dom\ I \cap Pre$ ist $I(p) \in U^{|p|} \to \mathbb{B}$.

Gegeben eine Struktur $\mathcal{A}$, verwendet wir die folgenden Notationen:

- $U_{\mathcal{A}}$ bezeichnet das Universum von $\mathcal{A}$.
- $\Sigma_{\mathcal{A}}$ bezeichnet die Signatur von $\mathcal{A}$ (also die Menge aller Funktions- und Prädikatensymbole, für die $\mathcal{A}$ Interpretationen vorgibt).
- $s_{\mathcal{A}}$ bezeichnet die Interpretation, die $\mathcal{A}$ dem Funktions- oder Prädikatensymbol $s \in \Sigma_{\mathcal{A}}$ zuordnet.
- $Val_{\mathcal{A}}$ bezeichnet die Menge $Var \to U_{\mathcal{A}}$. Die Elemente dieser Menge bezeichnet man als *Valuationen* oder *Belegungen*.

Eine Σ-*Struktur* ist ein Struktur $\mathcal{A}$ mit $\Sigma_{\mathcal{A}} = \Sigma$.

Jede Struktur legt also genau eine prädikatenlogische Sprache fest.

Abbildung 10.3 definiert zu einer Struktur die Denotationsfunktionen für Terme und Formeln. Statt $\mathcal{A}^{\mathrm{T}}[\![t]\!]$ und $\mathcal{A}^{\mathrm{F}}[\![A]\!]$ schreiben wir kürzer $\mathcal{A}[\![t]\!]$ und $\mathcal{A}[\![A]\!]$.

Eine Struktur *passt* zu einem Term oder einer Formel genau dann, wenn sie alle Funktions- und Prädikatensymbole des Terms oder der Formel interpretiert.

Zwei Strukturen $\mathcal{A}$ und $\mathcal{B}$ *unterscheiden sich nur auf* Σ genau dann, wenn $U_{\mathcal{A}} = U_{\mathcal{B}}$, $\Sigma_{\mathcal{A}} - \Sigma = \Sigma_{\mathcal{B}} - \Sigma$, und wenn $\mathcal{A}$ und $\mathcal{B}$ alle Symbole in $\Sigma_{\mathcal{A}} - \Sigma$ gleich interpretieren.

Proposition 10.2.1 *Seien Σ und Σ' Signaturen. Dann existiert zu jeder Struktur $\mathcal{A}$ mit $\Sigma \subseteq \Sigma_{\mathcal{A}} \cap \Sigma'$ eine Σ'-Struktur, die sich von $\mathcal{A}$ nur auf Symbolen unterscheidet, die in Σ nicht vorkommen.*

$$\mathcal{A}^{\mathrm{T}} \in Ter_{\Sigma_{\mathcal{A}}} \to Val_{\mathcal{A}} \to U_{\mathcal{A}}$$

$$\begin{aligned}
\mathcal{A}^{\mathrm{T}}[\![x]\!]\sigma &= \sigma x \\
\mathcal{A}^{\mathrm{T}}[\![f(t_1, \ldots, t_n)]\!]\sigma &= f_{\mathcal{A}}(\mathcal{A}^{\mathrm{T}}[\![t_1]\!]\sigma, \ldots, \mathcal{A}^{\mathrm{T}}[\![t_n]\!]\sigma)
\end{aligned}$$

$$\mathcal{A}^{\mathrm{F}} \in For_{\Sigma_{\mathcal{A}}} \to Val_{\mathcal{A}} \to \mathbb{B}$$

$$\begin{aligned}
\mathcal{A}^{\mathrm{T}}[\![p(t_1, \ldots, t_n)]\!]\sigma &= p_{\mathcal{A}}(\mathcal{A}^{\mathrm{T}}[\![t_1]\!]\sigma, \ldots, \mathcal{A}^{\mathrm{T}}[\![t_n]\!]\sigma) \\
\mathcal{A}^{\mathrm{F}}[\![\neg A]\!]\sigma &= 1 - \mathcal{A}^{\mathrm{F}}[\![A]\!]\sigma \\
\mathcal{A}^{\mathrm{F}}[\![A_1 \wedge A_2]\!]\sigma &= \min\{\mathcal{A}^{\mathrm{F}}[\![A_1]\!]\sigma,\ \mathcal{A}^{\mathrm{F}}[\![A_2]\!]\sigma\} \\
\mathcal{A}^{\mathrm{F}}[\![\exists x A]\!]\sigma &= \max\{\, \mathcal{A}^{\mathrm{F}}[\![A]\!](\sigma[u/x]) \mid u \in U_{\mathcal{A}} \,\}
\end{aligned}$$

Abbildung 10.3: Die Denotationsfunktionen einer Struktur $\mathcal{A}$

Proposition 10.2.2 (Signaturwechsel) *Seien $\mathcal{A}$ und $\mathcal{B}$ Strukturen mit $\Sigma \subseteq \Sigma_{\mathcal{A}} \cap \Sigma_{\mathcal{B}}$, die sich nur auf Symbolen unterscheiden, die in Σ nicht vorkommen. Dann:*

1. $Val_{\mathcal{A}} = Val_{\mathcal{B}}$.
2. $\forall t \in Ter_{\Sigma}\ \forall \sigma \in Val_{\mathcal{A}}\colon\ \mathcal{A}[\![t]\!]\sigma = \mathcal{B}[\![t]\!]\sigma$.
3. $\forall A \in For_{\Sigma}\ \forall \sigma \in Val_{\mathcal{A}}\colon\ \mathcal{A}[\![A]\!]\sigma = \mathcal{B}[\![A]\!]\sigma$.

Beweis Durch strukturelle Induktion über $t \in Ter_{\Sigma}$ und $A \in For_{\Sigma}$. □

Substitution

Am Beispiel von ASSN haben wir gezeigt, wie man eine Substitutionsfunktion für prädikatenlogische Formeln definiert (Abschnitt 6.5). Das war nicht ganz einfach. Glücklicherweise lässt sich die Konstruktion leicht auf beliebige prädikatenlogische Sprachen übertragen. Also gehen wir im Folgenden davon aus, dass wir eine Substitutionsfunktion

$$For \times Ter \times Var \to For$$

mit den folgen Eigenschaften haben:

1. $\forall A \in For_{\Sigma}\ \forall t \in Ter_{\Sigma}\ \forall x \in Var\colon\ A[t/x] \in For_{\Sigma}$.
2. Für jede Σ-Struktur $\mathcal{A}$:

$$\begin{aligned}
&\forall A \in For_{\Sigma}\ \forall t \in Ter_{\Sigma}\ \forall x \in Var\ \forall \sigma \in Val_{\mathcal{A}}\colon \\
&\qquad \mathcal{A}[\![A[t/s]]\!]\sigma = \mathcal{A}[\![A]\!](\sigma[\mathcal{A}[\![t]\!]\sigma/x])
\end{aligned}$$

Äquivalenz

Zwei Formel A und B heißen *äquivalent* (wir schreiben $A \models\!\!\!\dashv B$) genau dann, wenn $\mathcal{A}[\![A]\!] = \mathcal{A}[\![B]\!]$ für jede Struktur $\mathcal{A}$ gilt, die zu A und B passt. Selbstverständlich ist Äquivalenz von Formeln eine Äquivalenzrelation, die die folgende Ersetzungseigenschaft hat:

Proposition 10.2.3 (Ersetzung) *Sei $A \models\!\!\!\dashv A'$ und sei B' aus B durch das Ersetzen eines Auftretens der Teilformel A durch A' erhaltbar. Dann $B' \models\!\!\!\dashv B$.*

Abbildung 10.4 zeigt einige hilfreiche prädikatenlogischen Äquivalenzen.

Pränex- und Skolem-Form

Eine Formel ist in *Pränexform* genau dann, wenn sie die Form

$$Q_1 x_1 \ldots Q_n x_n \; A$$

hat, wobei $n \geq 0$, jedes Q_i für $\exists$ oder $\forall$ steht, und A eine quantorenfreie Formel ist. Eine Formel ist in *Skolem-Form* genau dann, wenn sie die Form

$$\forall x_1 \ldots \forall x_n \; A$$

hat, wobei $n \geq 0$ und A eine quantorenfreie Formel ist.

Proposition 10.2.4 *Zu jeder Formel in For_Σ existiert eine äquivalente Formel in For_Σ in Pränexform.*

Beweis Folgt mit den Äquivalenzen in Abbildung 10.4. □

Modelle, Allgemeingültigkeit und Erfüllbarkeit

Eine Struktur $\mathcal{A}$ heißt *Modell* einer Formel A genau dann, wenn $\mathcal{A}$ zu A passt und $\mathcal{A}[\![A]\!]\sigma = 1$ für jede Belegung $\sigma \in Val_{\mathcal{A}}$ gilt. Eine Formel A ist in einer Struktur $\mathcal{A}$ *gültig* genau dann, wenn $\mathcal{A}$ ein Modell von A ist.

Eine Formel A heißt *allgemeingültig* genau dann, wenn jede zu A passende Struktur ein Modell von A ist.

Proposition 10.2.5 $\forall A, B \in For\colon\; A \models\!\!\!\dashv B \iff (A \Leftrightarrow B)$ *allgemeingültig.*

Eine Formel A heißt *erfüllbar* genau dann, wenn es eine zu A passende Struktur $\mathcal{A}$ und eine Belegung $\sigma \in Val_{\mathcal{A}}$ gibt, so dass $\mathcal{A}[\![A]\!]\sigma = 1$. Eine Formel heißt *unerfüllbar* genau dann, wenn sie nicht erfüllbar ist.

Eine Struktur $\mathcal{A}$ heißt *Modell einer Formelmenge M* genau dann, wenn $\mathcal{A}$ ein Modell jeder Formel in M ist. Eine Formelmenge heißt *erfüllbar* genau dann, wenn sie ein Modell hat. Eine Formelmenge heißt *unerfüllbar* genau dann, wenn sie nicht erfüllbar ist.

$$
\begin{array}{rcll}
\neg\exists x A & \models\!\!\mid & \forall x \neg A & \\
\neg\forall x A & \models\!\!\mid & \exists x \neg A & \\[1ex]
\neg\exists A & \models\!\!\mid & \forall \neg A & \\
\neg\forall A & \models\!\!\mid & \exists \neg A & \\[1ex]
\exists x A & \models\!\!\mid & \neg\forall x \neg A & \\
\forall x A & \models\!\!\mid & \neg\exists x \neg A & \\[1ex]
\exists x A & \models\!\!\mid & A & \text{falls } x \notin FV(A) \\
\forall x A & \models\!\!\mid & A & \text{falls } x \notin FV(A) \\[1ex]
\exists x A & \models\!\!\mid & \exists y (A[y/x]) & \text{falls } y \notin FV(A) \\
\forall x A & \models\!\!\mid & \forall y (A[y/x]) & \text{falls } y \notin FV(A) \\[1ex]
(\exists x A) \wedge B & \models\!\!\mid & \exists x (A \wedge B) & \text{falls } x \notin FV(B) \\
(\forall x A) \vee B & \models\!\!\mid & \forall x (A \vee B) & \text{falls } x \notin FV(B) \\[1ex]
(\exists x A) \vee B & \models\!\!\mid & \exists x (A \vee B) & \text{falls } x \notin FV(B) \\
(\forall x A) \wedge B & \models\!\!\mid & \forall x (A \wedge B) & \text{falls } x \notin FV(B) \\[1ex]
(\exists x A) \vee (\exists x B) & \models\!\!\mid & \exists x (A \vee B) & \\
(\forall x A) \wedge (\forall x B) & \models\!\!\mid & \forall x (A \wedge B) & \\[1ex]
\forall x \forall y A & \models\!\!\mid & \forall y \forall x A & \\
\exists x \exists y A & \models\!\!\mid & \exists y \exists x A &
\end{array}
$$

Abbildung 10.4: Einige prädikatenlogische Äquivalenzen

Proposition 10.2.6 *Für jede Formel A sind die folgenden Aussagen äquivalent:*

1. *A allgemeingültig.*
2. *$\forall A$ allgemeingültig.*
3. *$\neg A$ unerfüllbar.*
4. *$\exists\neg A$ unerfüllbar.*
5. *$\{\exists\neg A\}$ unerfüllbar.*

Wir schreiben $M \models_\Sigma A$ genau dann, wenn $M \cup \{A\} \subseteq For_\Sigma$ und jedes Σ-Modell von M ein Modell von A ist. Wenn $M \models_\Sigma A$ gilt, bezeichnet man A als eine *logische Konsequenz* von M.

Proposition 10.2.7 *Sei $M \cup \{A\} \subseteq For_\Sigma$. Dann:*

$$M \models_\Sigma A \iff M \cup \{\exists\neg A\} \text{ unerfüllbar}$$

In diesem Kapitel werden wir zeigen, dass die Menge der allgemeingültigen Formeln testbar und unentscheidbar ist. Die Testbarkeit von Allgemeingültigkeit ist keineswegs offensichtlich, da Allgemeingültigkeit mit einer Quantifizierung über alle Strukturen definiert ist.

10.3 Variablenfreie Formeln

Eine Term oder eine Formel heißen *variablenfrei* genau dann, wenn sie keine Variablen enthalten. Variablenfreie Terme und Formeln nennt man auch *Grundterme* und *Grundformeln*. Wir definieren:

$$\begin{aligned} GTer &\overset{\text{def}}{=} \{\, t \in Ter \mid t \text{ variablenfrei} \,\} \\ GFor &\overset{\text{def}}{=} \{\, A \in For \mid A \text{ variablenfrei} \,\} \\ AGFor &\overset{\text{def}}{=} \{\, A \in GFor \mid A \text{ atomar} \,\} \end{aligned}$$

Wir werden zeigen, dass variablenfreie prädikatenlogische Formeln auch aussagenlogische Formeln sind, und dass für solche Formeln prädikatenlogische Erfüllbarkeit mit aussagenlogischer Erfüllbarkeit identisch ist.

Wir haben die aussagenlogische Sprache AL so definiert, dass wir die Menge der Variablen frei wählen können. Für den Kompaktheitssatz haben wir einschränkend verlangt, dass die Menge der Variablen abzählbar ist. Damit haben wir die Möglichkeit, die Menge *AGFor* der variablenfreien atomaren prädikatenlogischen Formeln als die Menge der aussagenlogische Variablen zu wählen. Das werden wir ab sofort tun. Wenn wir jetzt die Definition der prädikatenlogischen Syntax (Abbildung 10.1) mit der Definition der aussagenlogischen Syntax (Abbildung 9.2) vergleichen, stellen wir fest, dass die Menge der aussagenlogischen Formeln genau die Menge *GFor* der variablenfreien prädikatenlogischen Formeln ist. Wir werden zeigen, dass eine aussagenlogische Formel genau dann gültig [erfüllbar] ist, wenn sie als prädikatenlogische Formel gültig [erfüllbar] ist.

Sei $\mathcal{A}$ eine Σ-Struktur, $\sigma \in Val_{\mathcal{A}}$, $t \in GTer_{\Sigma}$ und $A \in GFor_{\Sigma}$. Da $\mathcal{A}[\![t]\!]\sigma$ und $\mathcal{A}[\![A]\!]\sigma$ nicht von σ abhängt, definieren wir:

$$\mathcal{A}^{\mathrm{G}}[\![t]\!] \stackrel{\text{def}}{=} \mathcal{A}[\![t]\!]\sigma$$

$$\mathcal{A}^{\mathrm{G}}[\![A]\!] \stackrel{\text{def}}{=} \mathcal{A}[\![A]\!]\sigma$$

Sei $\mathcal{A}$ eine Σ-Struktur. Dann ist

$$\sigma_{\mathcal{A}} \stackrel{\text{def}}{=} \lambda\, A \in AGFor\,\textbf{.}\ \text{if } A \in AGFor_{\Sigma} \text{ then } \mathcal{A}^{\mathrm{G}}[\![A]\!] \text{ else } 0$$

eine aussagenlogische Belegung.

Lemma 10.3.1 *Für jede Σ-Struktur $\mathcal{A}$ gilt: $\forall A \in GFor_{\Sigma}\colon\ \mathcal{F}[\![A]\!]\sigma_{\mathcal{A}} = \mathcal{A}^{G}[\![A]\!]$.*

Beweis Beweis durch strukturelle Induktion über $A \in GFor_{\Sigma}$. □

Eine *Termstruktur* ist eine Struktur $\mathcal{A}$ wie folgt:

- $U_{\mathcal{A}} = GTer_{\Sigma_{\mathcal{A}}}$.
- $\forall t \in GTer_{\Sigma_{\mathcal{A}}}\colon\ \mathcal{A}^{\mathrm{G}}[\![t]\!] = t$.

Termstrukturen nennt man auch *Herbrand-Strukturen*. Ein *Termmodell* für eine Formel A ist eine Termstruktur, die ein Modell von A ist.

Lemma 10.3.2 *Sei Σ eine Signatur mit mindestens einem Konstantensymbol. Dann existiert für jede aussagenlogische Belegung $\sigma \in AGFor \to \mathbb{B}$ eine Termstruktur $\mathcal{A}$ mit $\Sigma_{\mathcal{A}} = \Sigma$, so dass σ und $\sigma_{\mathcal{A}}$ auf $AGFor_{\Sigma}$ übereinstimmen.*

Beweis Sei $\sigma \in AGFor \to \mathbb{B}$. Sei $\mathcal{A}$ die Termstruktur mit $\Sigma_{\mathcal{A}} = \Sigma$ und

$$\forall p \in \Sigma \cap Pre\colon\ p_{\mathcal{A}} = \lambda\,(t_1, \ldots, t_{|p|}) \in GTer_{\Sigma}^{|p|}\,\textbf{.}\ \sigma(p(t_1, \ldots, t_{|p|}))$$

Dann stimmen σ und $\sigma_{\mathcal{A}}$ auf $AGFor_{\Sigma_{\mathcal{A}}}$ überein. □

Satz 10.3.3 (Transfer) *Sei $A \in GFor$. Dann:*

1. *A ist erfüllbar genau dann, wenn A aussagenlogisch erfüllbar ist.*
2. *A ist allgemeingültig genau dann, wenn A aussagenlogisch gültig ist.*

Beweis Sei A erfüllbar. Dann existiert eine zu A passende Struktur $\mathcal{A}$ mit $\mathcal{A}^{\mathrm{G}}[\![A]\!] = 1$. Also $\mathcal{F}[\![A]\!]\sigma_{\mathcal{A}} = 1$ mit Lemma 10.3.1. Also ist A aussagenlogisch erfüllbar.

Sei A aussagenlogisch erfüllbar. Dann existiert ein $\sigma \in AGFor \to \mathbb{B}$ mit $\mathcal{F}[\![A]\!]\sigma = 1$. Wir wählen eine Signatur Σ mit mindestens einem Konstantensymbol, so dass $A \in GFor_{\Sigma}$. Nach Lemma 10.3.2 existiert eine Termstruktur $\mathcal{A}$ mit $\Sigma_{\mathcal{A}} = \Sigma$, so dass σ und $\sigma_{\mathcal{A}}$ auf $AGFor_{\Sigma}$ übereinstimmen. Also $\mathcal{F}[\![A]\!]\sigma_{\mathcal{A}} = 1$. Also $\mathcal{A}^{\mathrm{G}}[\![A]\!] = 1$ mit Lemma 10.3.1. Also ist A erfüllbar.

Die zweite Behauptung folgt aus der ersten:

$$\begin{array}{rcl} A \text{ allgemeingültig} & \iff & \neg A \text{ unerfüllbar} \\ & \iff & \neg A \text{ aussagenlogisch unerfüllbar} \\ & \iff & A \text{ aussagenlogisch gültig} \end{array}$$

□

Korollar 10.3.4 *Die Allgemeingültigkeit und Erfüllbarkeit von variablenfreien prädikatenlogischen Formeln ist entscheidbar.*

10.4 Aussagenlogisch gültige Formeln

Auch auf Formeln mit Variablen lassen sich aussagenlogische Ergebnisse übertragen. Sei A eine allgemeingültige variablenfreie Formel. Sei A' aus A dadurch erhalten, dass man atomare variablenfreie Teilformeln konsistent durch irgendwelche Formeln ersetzt. Dann ist A' allgemeingültig. Aus dieser Behauptung folgt beispielsweise, dass die Formel

$$(A_1 \wedge A_2) \vee B \Leftrightarrow (A_1 \vee B) \wedge (A_2 \vee B)$$

für beliebige Formeln A_1, A_2 und B allgemeingültig ist.

Um diese Behauptung beweisen zu können, müssen wir präzise sagen, was wir unter konsistenter Ersetzung meinen. Jede Funktion

$$\gamma \in AGFor \to For$$

lässt sich wie folgt auf $GFor$ erweitern:

$$\gamma^{\mathrm{F}} \in GFor \to For$$

$$\begin{array}{rcl} \gamma^{\mathrm{F}}(p(t_1, \ldots, t_n)) & = & \gamma(p(t_1, \ldots, t_n)) \\ \gamma^{\mathrm{F}}(\neg A) & = & \neg(\gamma^{\mathrm{F}}(A)) \\ \gamma^{\mathrm{F}}(A_1 \wedge A_2) & = & \gamma^{\mathrm{F}}(A_1) \wedge \gamma^{\mathrm{F}}(A_2) \end{array}$$

Lemma 10.4.1 (Substitution) *Sei $\gamma \in AGFor \to For$ und $\mathcal{A}$ eine Struktur, die zu jeder Formel passt. Dann:*

$$\forall A \in GFor\ \forall \sigma \in Val_{\mathcal{A}}:$$
$$\mathcal{A}[\![\gamma^F(A)]\!]\sigma = \mathcal{F}[\![A]\!](\lambda\, B \in AGFor.\ \mathcal{A}[\![\gamma(B)]\!]\sigma)$$

Beweis Durch strukturelle Induktion über $A \in GFor$. □

Eine Struktur $\mathcal{A}'$ heißt *konservative Erweiterung* einer Struktur $\mathcal{A}$ genau dann, wenn $U_{\mathcal{A}} = U_{\mathcal{A}'}$, $\Sigma_{\mathcal{A}} \subseteq \Sigma_{\mathcal{A}'}$ und $\mathcal{A}$ und $\mathcal{A}'$ alle Symbole in $\Sigma_{\mathcal{A}}$ gleich interpretieren.

Satz 10.4.2 (Transfer) *Sei $A \in GFor$ allgemeingültig und $\gamma \in AGFor \to For$. Dann ist $\gamma^F(A)$ allgemeingültig.*

Beweis Sei $\mathcal{A}$ eine Struktur, die zu $\gamma^{\mathrm{F}}(A)$ passt. Sei $\mathcal{A}'$ eine konservative Erweiterung von $\mathcal{A}$, die zu allen Formeln passt. Dann folgt mit Lemma 10.4.1, dem ersten Transfersatz 10.3.3 und der Allgemeingültigkeit von A, dass $\mathcal{A}'$ ein Modell von $\gamma^{\mathrm{F}}(A)$ ist. Also ist $\mathcal{A}$ ein Modell von $\gamma^{\mathrm{F}}(A)$. □

Damit gelten wegen Proposition 10.2.5 alle aussagenlogischen Äquivalenzen auch für prädikatenlogische Sprachen. Insbesondere gilt dies für die Äquivalenzen in Abbildung 9.1.

10.5 Berechnung von Skolemformen

Wir zeigen jetzt, dass man zu jeder Formel A eine Formel B in Skolemform berechnen kann, so dass A genau dann erfüllbar ist, wenn B erfüllbar ist. Dafür benötigen wir eine Technik zur Elimination von Existenzquantoren. Das in diesem Abschnitt entwickelte Verfahren zur Elimination von Existenzquantoren ist der erste Teil eines Verfahrens zum Testen von Allgemeingültigkeit.

Lemma 10.5.1 (Skolem) *Sei A eine Formel mit $FV(\exists x A) = \{x_1, \dots, x_n\}$ und sei f ein n-stelliges Funktionssymbol, das nicht in A vorkommt. Dann:*

1. *Für jede Struktur, die zu $A[f(x_1, \dots, x_n)/x]$ passt, gilt:*

   $$\forall \sigma \in Val_{\mathcal{A}}:\ \mathcal{A}[\![A[f(x_1, \dots, x_n)/x]]\!]\sigma = 1 \ \Rightarrow\ \mathcal{A}[\![\exists x A]\!]\sigma = 1$$

2. *Für jede Struktur $\mathcal{A}$, die zu $\exists x A$ passt, existiert eine Struktur $\mathcal{B}$, die zu $A[f(x_1, \dots, x_n)/x]$ passt, wie folgt:*

   a) *$\mathcal{A}$ und $\mathcal{B}$ unterscheiden sich nur auf $\{f\}$.*

   b) $\forall \sigma \in Val_{\mathcal{A}}:\ \mathcal{A}[\![\exists x A]\!]\sigma = 1 \ \Rightarrow\ \mathcal{B}[\![A[f(x_1, \dots, x_n)/x]]\!]\sigma = 1.$

Beweis (1) Sei $\mathcal{A}$ eine Struktur, die zu $A[f(x_1, \ldots, x_n)/x]$ passt. Sei $\sigma \in Val_{\mathcal{A}}$ mit $\mathcal{A}[\![A[f(x_1, \ldots, x_n)/x]]\!]\sigma = 1$. Dann:

$$\begin{aligned}\mathcal{A}[\![\exists x A]\!]\sigma &= \max\{\, \mathcal{A}[\![A]\!](\sigma[u/x]) \mid u \in U_{\mathcal{A}} \,\} \\ &\geq \mathcal{A}[\![A]\!](\sigma[\mathcal{A}[\![f(x_1, \ldots, x_n)]\!]\sigma/x]) \\ &= \mathcal{A}[\![A[f(x_1, \ldots, x_n)/x]]\!]\sigma \\ &= 1\end{aligned}$$

(2) Sei $\mathcal{A}$ eine Struktur, die zu $\exists x A$ passt. Dann existiert für jedes $\sigma \in Val_{\mathcal{A}}$ mit $\mathcal{A}[\![\exists x A]\!]\sigma = 1$ ein $u \in U_{\mathcal{A}}$ mit $\mathcal{A}[\![A]\!](\sigma[u/x]) = 1$. Dabei hängt u nur von $\sigma(x_1), \ldots, \sigma(x_n)$ ab. Also gibt es eine Funktion $\phi \in U_{\mathcal{A}}^n \to U_{\mathcal{A}}$ (Auswahlaxiom) mit:

$$\forall \sigma \in Val_{\mathcal{A}}\colon\ \mathcal{A}[\![\exists x A]\!]\sigma = 1 \ \Rightarrow\ \mathcal{A}[\![A]\!](\sigma[\phi(\sigma(x_1), \ldots, \sigma(x_n))/x]) = 1$$

Sei $\mathcal{B}$ die Struktur mit $f_{\mathcal{B}} = \phi$, die sich nur auf $\{f\}$ von $\mathcal{A}$ unterscheidet. Sei $\sigma \in Val_{\mathcal{A}}$ mit $\mathcal{A}[\![\exists x A]\!]\sigma = 1$. Da f in A nicht vorkommt, gilt:

$$\begin{aligned}\mathcal{B}[\![A[f(x_1, \ldots, x_n)/x]]\!]\sigma &= \mathcal{B}[\![A]\!](\sigma[\mathcal{B}[\![f(x_1, \ldots, x_n)]\!]\sigma/x]) \\ &= \mathcal{A}[\![A]\!](\sigma[\phi(\sigma(x_1), \ldots, \sigma(x_n))/x]) \\ &= 1\end{aligned}$$

□

Die für die Elimination des Existenzquantors eingeführte Funktion nennt man eine *Skolem-Funktion*.

Satz 10.5.2 (Skolem) *Zu jeder Formel A kann man eine Formel B in Skolemform berechnen, so dass gilt:*

1. *A ist genau dann erfüllbar, wenn B erfüllbar ist.*
2. *Jedes Modell von B ist ein Modell von A.*
3. *Zu jedem Modell $\mathcal{A}$ von A existiert ein Modell $\mathcal{B}$ von B, so dass $\mathcal{A}$ und $\mathcal{B}$ sich nur auf Funktionssymbolen unterscheiden, die in A nicht vorkommen.*

Beweis Zuerst bringt man A in Pränexform. Dann eliminiert man den äußersten Existenzquantor von A gemäß Lemma 10.5.1. Dabei sind eventuelle Allquantoren über dem Existenzquantor kein Problem, da Modelle freie Variablen implizit allquantifizieren. Dies wiederholt man solange, bis alle Existenzquantoren eliminiert sind. □

10.6 Quantorenfreie Klauseldarstellung

Wir geben jetzt ein detailliertes Verfahren zur Berechnung von Skolem-Formen an, dass mit einer prädikatenlogischen Klauseldarstellung arbeitet. Insgesamt bekommen wir ein Verfahren, mit dem Klauselmengen in quantorenfreie Klauselmengen überführt werden können. Dabei ist die Ausgangsmenge genau dann erfüllbar, wenn die quantorenfreie Ergebnismenge erfüllbar ist.

Eine *Klausel* ist eine endliche Menge von Formeln. Wir vereinbaren die folgende Notation:

$$C, D \in \mathit{Cla} \stackrel{\text{def}}{=} \mathcal{P}_{\mathit{fin}}(\mathit{For}) \qquad \textit{Klausel}$$

$$S \in \mathcal{P}_{\mathit{fin}}(\mathit{For}_\Sigma) \qquad \textit{Klauselmenge}$$

$$FV(C) \stackrel{\text{def}}{=} \bigcup_{A \in C} FV(A) \qquad \textit{Freie Variablen}$$

Wir werden mit der konjunktiven Interpretation von Klauselmengen arbeiten. Zusätzlich definieren wir die Semantik von Klauselmengen so, dass eine Klausel die Variablen, die in ihren Formeln frei vorkommen, allquantifiziert. Eine Klausel

$$\{A_1, \ldots, A_n\}$$

wird also wie die Formel

$$\forall x_1 \ \ldots \ \forall x_m \ (A_1 \vee \cdots \vee A_n)$$

interpretiert, wobei $\{x_1, \ldots, x_m\} = FV(A_1) \cup \cdots \cup FV(A_n)$.

Eine Struktur $\mathcal{A}$ heißt *Modell* einer Klausel C genau dann, wenn $\mathcal{A}$ zu jeder Formel in C passt und

$$\forall \sigma \in \mathit{Val}_\mathcal{A} \ \exists A \in C\colon \ \mathcal{A}[\![A]\!]\sigma = 1$$

gilt. Eine Struktur $\mathcal{A}$ heißt *Modell* einer Klauselmenge S genau dann, wenn $\mathcal{A}$ ein Model jeder Klausel in S ist.

Eine Klauselmenge heißt *erfüllbar* genau dann, wenn sie mindestens ein Modell hat. Eine Klauselmenge heißt *unerfüllbar* genau dann, wenn sie kein Modell hat.

Proposition 10.6.1 *Zu jeder endlichen Klauselmenge $S \subseteq \mathit{Cla}_\Sigma$ existiert eine geschlossene Formel $A \in \mathit{For}_\Sigma$, die dieselben Modelle wie S hat.*

Proposition 10.6.2 *Sei M eine Formelmenge. Dann haben M und die Klauselmenge $\{\,\{A\} \mid A \in M\,\}$ dieselben Modelle.*

$$
\begin{array}{lrcl}
(1) & S,\ (C,\ \neg\neg A) & \xrightarrow{s} & S,\ (C,\ A) \\
(2) & S,\ (C,\ \neg(A_1 \wedge A_2)) & \xrightarrow{s} & S,\ (C,\ \neg A_1,\ \neg A_2) \\
(3) & S,\ (C,\ A_1 \wedge A_2) & \xrightarrow{s} & S,\ (C,\ A_1),\ (C,\ A_2) \\
(4) & S,\ (C,\ \neg\exists x A) & \xrightarrow{s} & S,\ (C,\ \neg A[y/x]) \\
 & & & \text{falls } y \notin FV(C) \ \wedge\ (x = y \vee y \notin FV(A)) \\
(5) & S,\ (C,\ \exists x A) & \xrightarrow{s} & S,\ (C,\ A[f(x_1, \dots, x_n)/x]) \\
 & & & \text{falls } FV(\exists x A) = \{x_1, \dots, x_n\} \text{ und } f \text{ ein } n\text{-stelliges} \\
 & & & \text{Funktionssymbol ist, das in } S, C \text{ und } A \text{ nicht vorkommt}
\end{array}
$$

Abbildung 10.5: Vereinfachungsregeln für prädikatenlogische Klauselmengen

Proposition 10.6.3 *Jede Klauselmenge, die die leere Klausel enthält, ist unerfüllbar.*

Eine Klausel C heißt *trivial* genau dann, wenn es eine Formel A gibt, so dass $A \in C$ und $\neg A \in C$.

Proposition 10.6.4 (Triviale Klauseln) *Sei S eine Klauselmenge und C eine triviale Klausel. Dann haben $S \cup \{C\}$ und S dieselben Modelle.*

Ein *Literal* ist entweder eine atomare Formel $p(t_1, \dots, t_n)$ oder eine negierte atomare Formel $\neg p(t_1, \dots, t_n)$.

Eine Klausel heißt *literal* genau dann, wenn sie nur Literale enthält. Eine Klauselmenge heißt *literal* genau dann, wenn sie nur literale Klauseln enthält.

Abbildung 10.5 zeigt die Vereinfachungsregeln für prädikatenlogische Klauselmengen. Die ersten drei Regeln entsprechen den konjunktiven Vereinfachungsregeln für aussagenlogische Klauselmengen. Wir verzichten diesmal auf die Eliminationsregel für triviale Klauseln, damit bei der Vereinfachung keine Funktions- und Prädikatensymbole verloren gehen. Die vierte Regel vereinfacht interne negierte Existenzquantifizierung zu externer Allquantifizierung. Die fünfte Regel eliminiert interne Existenzquantoren durch die Einführung von Skolemfunktionen.

Wir schreiben $S \xrightarrow{s} S'$ genau dann, wenn die Klauselmenge S' aus der Klauselmenge S durch die Anwendung einer Vereinfachungsregel erhalten werden kann.

Wir schreiben $S \models\!\!\Rightarrow S'$ genau dann, wenn die folgenden Aussagen gelten:

1. Jedes Modell von S' ist ein Modell von S.
2. Zu jedem Modell $\mathcal{A}$ von S existiert ein Modell $\mathcal{B}$ von S', so dass $\mathcal{A}$ und $\mathcal{B}$ sich nur auf Funktionssymbolen unterscheiden, die in A nicht vorkommen.

Die wesentlichen Eigenschaften der Vereinfachungsregeln werden durch den folgenden Satz formuliert.

Satz 10.6.5 (Vereinfachung) *Sei S eine Klauselmenge. Dann:*

1. *Wenn $S \stackrel{s}{\rightarrow} S'$, dann $S \models\!\!\Rightarrow S'$.*
2. *S ist literal genau dann, wenn es kein S' mit $S \stackrel{s}{\rightarrow} S'$ gibt.*
3. *Wenn S endlich ist, dann gibt es keine unendliche Kette*

   $S \stackrel{s}{\rightarrow} S_1 \stackrel{s}{\rightarrow} S_2 \stackrel{s}{\rightarrow} S_3 \stackrel{s}{\rightarrow} \cdots$

Beweis Behauptung (1) folgt mit dem Skolem-Lemma 10.5.1. Behauptung (2) gilt offensichtlich. Behauptung (3) folgt mit ähnlichen Argumenten wie bei den aussagenlogischen Vereinfachungsregeln. □

Eine Klauselmenge S heißt *sparsam* genau dann, wenn zu jedem $n \in \mathbb{N}$ unendlich viele n-stellige Funktionssymbole existieren, die nicht in S vorkommen.

Lemma 10.6.6 *Zu jeder sparsamen Klauselmenge S existiert eine literale Klauselmenge S' mit $S \models\!\!\Rightarrow S'$. Zudem gibt es eine berechenbare Funktion, die zu einem Tester für S einen Tester für S' berechnet.*

Beweis Da die Menge aller Formeln abzählbar ist, hat jede Klauselmenge nur abzählbar viele Klauseln. Damit kann man die Klauseln einer Klauselmenge Schritt für Schritt vereinfachen. Diese eventuell unendliche Berechnung konvergiert gegen eine literale Klauselmenge. □

Eine Formelmenge M heißt *sparsam* genau dann, wenn die M entsprechende Klauselmenge $\{\,\{A\} \mid A \in M\,\}$ sparsam ist. Wir schreiben $M \models\!\!\Rightarrow S$ genau dann, wenn $\{\,\{A\} \mid A \in M\,\} \models\!\!\Rightarrow S$ gilt.

Satz 10.6.7 (Elimination) *Zu jeder sparsamen Formelmenge M existiert eine literale Klauselmenge S mit $M \models\!\!\Rightarrow S$. Zudem gibt es eine berechenbare Funktion, die zu einem Tester für M einen Tester für S berechnet.*

Beweis Folgt aus Proposition 10.6.2 und Lemma 10.6.6. □

Der obige Satz heißt Eliminationssatz, da er die Elimination aller Quantoren der internen Syntax erlaubt.

Proposition 10.6.8 *Mithilfe einer konsistenten Umbenennung*

$\lambda\,(0, m, n) \in \mathit{Fun}\,.\;(0, 2m, n)$

der Funktionssymbole kann man jede Formelmenge M in eine sparsame Formelmenge überführen, die bis auf diese Umbenennung dieselben Modelle wie M hat.

10.7 Der Expansionssatz

Sei S eine quantorenfreie Klauselmenge. Die Menge aller Grundinstanzen der Klauseln in S heißt die Expansion von S und ist eine aussagenlogische Klauselmenge. Wir zeigen, dass eine quantorenfreie Klauselmenge genau dann unerfüllbar ist, wenn ihre Expansion unerfüllbar ist. Mit diesem Ergebnis lassen sich die Kompaktheits- und Testbarkeitseigenschaften der Aussagenlogik auf die Prädikatenlogik übertragen. Insbesondere erhalten wir ein Testverfahren für die Allgemeingültigkeit von Formeln.

Satz 10.7.1 *Sei $S \subseteq Cla_\Sigma$ eine variablenfreie und nichtleere Klauselmenge. Dann sind die folgenden Aussagen äquivalent:*

1. *S erfüllbar.*
2. *S hat ein Σ-Termmodell $\mathcal{A}$.*
3. *$\mathcal{K}[\![S]\!] \neq \emptyset$.*

Beweis Σ enthält mindestens ein Konstantensymbol, da S variablenfrei und nichtleer ist.

(1) $\Rightarrow$ (3). Sei S erfüllbar. Dann hat S ein Σ-Modell $\mathcal{A}$ (Proposition 10.2.2). Damit gilt

$$\forall C \in S\ \exists A \in C\colon\ \mathcal{A}^{\mathrm{G}}[\![A]\!] = 1$$

Mit Lemma 10.3.1 folgt

$$\forall C \in S\ \exists A \in C\colon\ \mathcal{F}[\![A]\!]\sigma_{\mathcal{A}} = 1$$

Also $\sigma_{\mathcal{A}} \in \mathcal{K}[\![S]\!]$.

(3) $\Rightarrow$ (2). Sei $\mathcal{K}[\![S]\!] \neq \emptyset$. Dann gibt es ein $\sigma \in AGFor \to \mathbb{B}$ mit

$$\forall C \in S\ \exists A \in C\colon\ \mathcal{F}[\![A]\!]\sigma = 1$$

Lemma 10.3.2 liefert uns zu σ eine Σ-Termstruktur $\mathcal{A}$ und der Eigenschaft, dass $\sigma_{\mathcal{A}}$ und σ auf $AGFor_\Sigma$ übereinstimmen. Also gilt

$$\forall C \in S\ \exists A \in C\colon\ \mathcal{F}[\![A]\!]\sigma_{\mathcal{A}} = 1$$

Mit Lemma 10.3.1 folgt

$$\forall C \in S \; \exists A \in C: \; \mathcal{A}^{\mathrm{G}}[\![A]\!] = 1$$

Also ist $\mathcal{A}$ ein Termmodell von S.

(2) $\Rightarrow$ (1). Trivial. □

Wir definieren:

$$\mathit{QFor} \stackrel{\text{def}}{=} \{\, A \in \mathit{For} \mid A \text{ quantorenfrei} \,\}$$

Für jede Funktion $\sigma \in \mathit{Var} \to \mathit{GTer}_\Sigma$ definieren wir zwei Funktionen:

$$\sigma^{\mathrm{T}} \in \mathit{Ter}_\Sigma \to \mathit{GTer}_\Sigma$$

$$\begin{aligned} \sigma^{\mathrm{T}}(x) &= \sigma x \\ \sigma^{\mathrm{T}}(f(t_1, \ldots, t_n)) &= f(\sigma^{\mathrm{T}}(t_1), \ldots, \sigma^{\mathrm{T}}(t_n)) \end{aligned}$$

$$\sigma^{\mathrm{F}} \in \mathit{QFor}_\Sigma \to \mathit{GFor}_\Sigma$$

$$\begin{aligned} \sigma^{\mathrm{F}}(p(t_1, \ldots, t_n)) &= p(\sigma^{\mathrm{T}}(t_1), \ldots, \sigma^{\mathrm{T}}(t_n)) \\ \sigma^{\mathrm{F}}(\neg A) &= \neg(\sigma^{\mathrm{F}}(A)) \\ \sigma^{\mathrm{F}}(A_1 \wedge A_2) &= \sigma^{\mathrm{F}}(A_1) \wedge \sigma^{\mathrm{F}}(A_2) \end{aligned}$$

Lemma 10.7.2 (Substitution) *Sei $\mathcal{A}$ eine Σ-Struktur und $\sigma \in \mathit{Var} \to \mathit{GTer}_\Sigma$. Dann:*

1. $\forall t \in \mathit{Ter}_\Sigma: \; \mathcal{A}^G[\![\sigma^T(t)]\!] = \mathcal{A}[\![t]\!](\lambda\, x \in \mathit{Var}.\, \mathcal{A}^G[\![\sigma(x)]\!])$
2. $\forall A \in \mathit{QFor}_\Sigma: \; \mathcal{A}^G[\![\sigma^F(A)]\!] = \mathcal{A}[\![A]\!](\lambda\, x \in \mathit{Var}.\, \mathcal{A}^G[\![\sigma(x)]\!])$.

Beweis Durch strukturelle Induktion über $t \in \mathit{Ter}_\Sigma$ und $A \in \mathit{QFor}_\Sigma$. □

Lemma 10.7.3 (Substitution) *Sei $\mathcal{A}$ eine Σ-Termstruktur und $\sigma \in \mathit{Val}_{\mathcal{A}}$. Dann:*

1. $\forall t \in \mathit{Ter}_\Sigma: \; \mathcal{A}[\![t]\!]\sigma = \sigma^T(t)$.
2. $\forall A \in \mathit{QFor}_\Sigma: \; \mathcal{A}[\![A]\!]\sigma = \mathcal{A}^G[\![\sigma^F(A)]\!]$.

Beweis Folgt unmittelbar aus dem allgemeineren Lemma 10.7.2. □

Eine Klauselmenge heißt *quantorenfrei* genau dann, wenn alle ihre Klauseln nur quantorenfreie Formeln enthalten.

Die Σ-*Expansion* einer quantorenfreien Klauselmenge $S \subseteq \mathit{Cla}_\Sigma$ ist wie folgt definiert:

$$\mathit{Exp}_\Sigma(S) \stackrel{\text{def}}{=} \{\, \{\, \sigma^{\mathrm{F}}(A) \mid A \in C \,\} \mid C \in S \text{ und } \sigma \in \mathit{Var} \to \mathit{GTer}_\Sigma \,\}$$

Offensichtlich ist $\mathit{Exp}_\Sigma(S)$ eine variablenfreie Teilmenge von Cla_Σ. Die Klauseln in $\mathit{Exp}_\Sigma(S)$ nennt man auch die Σ-*Grundinstanzen* der Klauseln in S.

Lemma 10.7.4 *Sei $S \subseteq Cla_\Sigma$ eine quantorenfreie Klauselmenge und $\mathcal{A}$ ein Σ-Termmodell von $Exp_\Sigma(S)$. Dann ist $\mathcal{A}$ ein Modell von S.*

Beweis Sei $C \in S$ und $\sigma \in Val_{\mathcal{A}}$. Es genügt zu zeigen, dass ein $A \in C$ existiert mit $\mathcal{A}[\![A]\!]\sigma = 1$.

Offensichtlich ist $\{\, \sigma^{\mathrm{F}}(A) \mid A \in C \,\} \in Exp_\Sigma(S)$. Also ist $\mathcal{A}$ ein Modell dieser variablenfreien Klausel. Also existiert ein $A \in C$ mit $\mathcal{A}^{\mathrm{G}}[\![\sigma^{\mathrm{F}}(A)]\!] = 1$. Mit dem Substitutionslemma 10.7.3 folgt $\mathcal{A}[\![A]\!]\sigma = 1$. □

Proposition 10.7.5 *Sei $S \subseteq Cla_\Sigma$ eine erfüllbare quantorenfreie Klauselmenge. Dann ist $Exp_\Sigma(S)$ erfüllbar.*

Beweis Wenn Σ kein Konstantensymbol enthält, dann ist $Exp_\Sigma(S)$ leer und folglich erfüllbar.

Enthalte Σ mindestens ein Konstantensymbol und sei $\mathcal{A}$ ein Modell von S. Wegen Proposition 10.2.2 können wir annehmen, dass $\Sigma_{\mathcal{A}} = \Sigma$. Sei $C \in S$ und $\sigma \in Var \to GTer_\Sigma$. Es genügt zu zeigen, dass ein $A \in C$ existiert mit $\mathcal{A}^{\mathrm{G}}[\![\sigma^{\mathrm{F}}(A)]\!] = 1$.

Da $\mathcal{A}$ ein Modell von S ist, gibt es ein $A \in C$ mit

$$\mathcal{A}[\![A]\!](\lambda\, x \in Var\,.\, \mathcal{A}^{\mathrm{G}}[\![\sigma(x)]\!])$$

Also gilt $\mathcal{A}^{\mathrm{G}}[\![\sigma^{\mathrm{F}}(A)]\!] = 1$ mit Lemma 10.7.2. □

Satz 10.7.6 (Expansion) *Sei $S \subseteq Cla_\Sigma$ eine quantorenfreie Klauselmenge und enthalte Σ mindestens ein Konstantensymbol. Dann sind die folgenden Aussagen äquivalent:*

1. *S erfüllbar.*
2. *S hat ein Σ-Termmodell.*
3. *$\mathcal{K}[\![Exp(S)_\Sigma]\!] \neq \emptyset$.*

Beweis (1) $\Rightarrow$ (3). Sei S erfüllbar. Dann ist $Exp(S)_\Sigma$ erfüllbar (Proposition 10.7.5). Also $\mathcal{K}[\![Exp_\Sigma(S)]\!] \neq \emptyset$ mit Satz 10.7.1.

(3) $\Rightarrow$ (2). Sei $\mathcal{K}[\![Exp_\Sigma(S)]\!] \neq \emptyset$. Dann liefert Satz 10.7.1 ein Σ-Termodell $\mathcal{A}$ für $Exp_\Sigma(S)$. Mit Lemma 10.7.4 folgt, dass $\mathcal{A}$ ein Modell von S ist.

(2) $\Rightarrow$ (1). Trivial. □

Satz 10.7.7 (Herbrand) *Sei $S \subseteq Cla_\Sigma$ eine quantorenfreie Klauselmenge und enthalte Σ mindestens ein Konstantensymbol. Dann ist S unerfüllbar genau dann, wenn es eine endliche Teilmenge $S' \subseteq Exp_\Sigma(S)$ gibt mit $\mathcal{K}[\![S']\!] = \emptyset$.*

Beweis Folgt aus dem Expansionssatz 10.7.6 und dem aussagenlogischen Kompaktheitssatz 9.12.11. □

Eine Struktur heißt *abzählbar* genau dann, wenn ihr Universum abzählbar ist. Da die Menge aller Terme abzählbar ist, ist jedes Termmodell abzählbar.

Satz 10.7.8 (Löwenheim-Skolem) *Jede erfüllbare Formelmenge hat ein abzählbares Modell.*

Beweis Folgt aus dem Eliminationssatz 10.6.7, Proposition 10.6.8 und dem Expansionssatz 10.6.7, da jede Termstruktur abzählbar ist. □

Satz 10.7.9 (Kompaktheit) *Eine Formelmenge ist erfüllbar genau dann, wenn jede ihrer endlichen Teilmengen erfüllbar ist.*

Beweis Folgt aus dem Eliminationssatz 10.6.7, Proposition 10.6.8 und dem Satz von Herbrand (10.7.7). □

Satz 10.7.10 (Testbarkeit) *Die Unerfüllbarkeit von testbaren Formelmengen ist testbar.*

Beweis Folgt aus dem Eliminationssatz 10.6.7, Proposition 10.6.8, dem Expansionssatz 10.7.6 und dem aussagenlogischen Testbarkeitssatz 9.12.10 □

Korollar 10.7.11 *Allgemeingültigkeit von Formeln ist testbar.*

10.8 Unifikation

Unifikation ist eine Operation, die allgemeinste Lösungen für Termgleichungen berechnet. Wir benötigen Unifikation als Baustein für einen praktisch brauchbaren prädikatenlogischen Resolutionsbeweiser. Darüber hinaus ist Unifikation ein wichtiger Baustein für einen Typableiter für Standard ML.

Im Folgenden legen wir eine Signatur Σ zugrunde, die mindestens ein Konstantensymbol enthält. Wir werden ausschließlich mit syntaktischen Objekten arbeiten, in denen nur Funktions- und Prädikatensymbole aus Σ vorkommen. Um unsere Notation übersichtlich zu halten lassen wir bei Ter_Σ, For_Σ und so weiter den Index Σ weg und schreiben einfach *Ter*, *For* und so weiter.

Eine *Substitution* ist eine endliche Funktion $\theta \in \mathit{Var} \stackrel{\mathit{fin}}{\rightharpoonup} \mathit{Ter}$, die die folgende Bedingung erfüllt: $\forall x \in \mathit{Dom}\,\theta\colon\ \theta x \neq x$. Diese Bedingung sorgt dafür, dass beispielsweise $\{x \mapsto x\}$ keine Substitution ist. Sie ist notwendig, damit Proposition 10.8.1 gilt. Die Menge aller Substitutionen bezeichnen wir mit *Sub*.

Die Erweiterungen einer Substitution θ auf Terme und quantorenfreie Formeln und Klauseln ist wie folgt definiert:

$\theta^{\mathrm{T}} \in \mathit{Ter} \to \mathit{Ter}$

$$\begin{aligned} \theta^{\mathrm{T}}(x) &= \text{if } x \in \mathit{Dom}\,\theta \text{ then } \theta x \text{ else } x \\ \theta^{\mathrm{T}}(f(t_1, \dots, t_n)) &= f(\theta^{\mathrm{T}}(t_1), \dots, \theta^{\mathrm{T}}(t_n)) \end{aligned}$$

$\theta^{\mathrm{F}} \in \mathit{QFor} \to \mathit{QFor}$

$$\begin{aligned} \theta^{\mathrm{F}}(p(t_1, \dots, t_n)) &= p(\theta^{\mathrm{T}}(t_1), \dots, \theta^{\mathrm{T}}(t_n)) \\ \theta^{\mathrm{F}}(\neg A) &= \neg(\theta^{\mathrm{F}}(A)) \\ \theta^{\mathrm{F}}(A_1 \wedge A_2) &= \theta^{\mathrm{F}}(A_1) \wedge \theta^{\mathrm{F}}(A_2) \end{aligned}$$

$\theta^{\mathrm{C}} \in \mathit{QCla} \to \mathit{QCla}$

$$\theta^{\mathrm{C}}(C) = \{\, \theta^{\mathrm{F}}(A) \mid A \in C \,\}$$

Da θ immer eine Substitution bezeichnen wird, schreiben wir statt $\theta^{\mathrm{T}}(t)$, $\theta^{\mathrm{F}}(A)$ und $\theta^{\mathrm{C}}(C)$ kürzer θt, θA und θC.

Proposition 10.8.1 *Seien θ_1 und θ_2 Substitutionen. Dann gilt:*

$$\theta_1 = \theta_2 \iff \forall t \in \mathit{Ter}\colon\ \theta_1 t = \theta_2 t$$

Proposition 10.8.2 *Seien $\theta_1, \theta_2 \in \mathit{Sub}$. Dann existiert genau ein $\theta \in \mathit{Sub}$ mit $\theta^T = \lambda\, t \in \mathit{Ter}.\ \theta_1(\theta_2 t)$. Die Substitution θ heißt die* Komposition θ_1 nach θ_2 *und wird mit $\theta_1 \circ \theta_2$ bezeichnet.*

Sei $S \subseteq \mathit{Sub}$. Dann heißt $\theta \in S$ ein *allgemeinstes Element* von S genau dann, wenn $\forall \psi \in S\ \exists \phi \in \mathit{Sub}\colon\ \psi = \phi \circ \theta$.

Eine *Umbenennung* ist eine Substitution θ mit $\emptyset = \theta \circ \theta$. Wenn θ eine Umbenennung ist, dann ist θ^{T} eine bijektive Funktion $\mathit{Ter} \to \mathit{Ter}$ die sich selbst als Umkehrfunktion hat. Für eine Umbenennung θ gilt also $\forall t \in \mathit{Ter}\colon\ t = \theta(\theta t)$.

Lemma 10.8.3 (Substitution) *Für jede Σ-Struktur $\mathcal{A}$ gilt:*

(1) $\forall t \in \mathit{Ter}\ \forall \theta \in \mathit{Sub}\ \forall \sigma \in \mathit{Val}_{\mathcal{A}}$:

$$\mathcal{A}[\![\theta t]\!]\sigma = \mathcal{A}[\![t]\!](\lambda\, x \in \mathit{Var}.\ \mathcal{A}[\![\theta x]\!]\sigma)$$

(2) $\forall A \in \mathit{QFor}\ \forall \theta \in \mathit{Sub}\ \forall \sigma \in \mathit{Val}_{\mathcal{A}}$:

$$\mathcal{A}[\![\theta A]\!]\sigma = \mathcal{A}[\![A]\!](\lambda\, x \in \mathit{Var}.\ \mathcal{A}[\![\theta x]\!]\sigma)$$

Beweis Durch strukturelle Induktion über $t \in \mathit{For}$ und $A \in \mathit{For}$. □

Satz 10.8.4 (Instanzen) *Sei $\mathcal{A}$ ein Σ-Modell einer Klauselmenge $S \subseteq \mathit{Cla}$. Dann: $\forall C \in S\ \forall \theta \in \mathit{Sub}\colon\ \mathcal{A}$ ist Modell von θC.*

Beweis Sei $C \in S$, $\theta \in Sub$ und $\sigma \in Val_{\mathcal{A}}$. Da $\mathcal{A}$ Modell von S ist, existiert ein $A \in C$ mit $\mathcal{A}[\![A]\!](\lambda x \in Var.\ \mathcal{A}[\![\theta x]\!]\sigma) = 1$. Mit Lemma 10.8.3 folgt $\mathcal{A}[\![\theta A]\!]\sigma = 1$. Also ist $\mathcal{A}$ eine Modell von θC. $\square$

Eine *Gleichungsmenge* ist eine endliche Menge von Paaren (s, t), die aus zwei Termen bestehen. Wir vereinbaren die folgende Notation:

$$E \in GM \overset{\text{def}}{=} \mathcal{P}_{fin}(Ter \times Ter) \qquad \textit{Gleichungsmenge}$$

$$s \doteq t \in E \qquad \textit{Gleichung}$$

Die *Unifikatoren* einer Gleichungsmenge sind wie folgt definiert:

$$\mathcal{U} \in GM \to Sub$$

$$\mathcal{U}[\![E]\!] = \{\, \theta \in Sub \mid \forall s \doteq t \in E\colon\ \theta s = \theta t \,\}$$

Die Elemente von $\mathcal{U}[\![E]\!]$ heißen die *Unifikatoren* von E. Eine Gleichungsmenge E heißt *unifizierbar* genau dann, wenn $\mathcal{U}[\![E]\!] \neq \emptyset$.

Sei E eine Gleichungsmenge. Eine Substitution $\theta \in \mathcal{U}[\![E]\!]$ heißt *allgemeinster Unifikator* von E genau dann, wenn θ ein allgemeinstes Element von $\mathcal{U}[\![E]\!]$ ist. Eine Substitution $\theta \in \mathcal{U}[\![E]\!]$ heißt *prinzipaler Unifikator* von E genau dann, wenn die folgenden Bedingungen gelten:

1. θ ist ein allgemeinster Unifikator von E.
2. $\theta = \theta \circ \theta$ (Idempotenz).
3. In *Dom* θ und *Ran* θ kommen nur Variablen vor, die auch in E vorkommen.

Eine Gleichung heißt *widerlegt* genau dann, wenn sie eine der folgenden zwei Formen hat:

1. $f(s_1, \ldots, s_m) \doteq g(t_1, \ldots, t_n)$ mit $f \neq g$.
2. $x = f(t_1, \ldots, t_n)$ mit x kommt in einem der Terme $t_1, \ldots, t_n$. vor.

Eine Gleichungsmenge E heißt *widerlegt* genau dann, wenn sie eine widerlegte Gleichung enthält.

Proposition 10.8.5 *Widerlegte Gleichungsmengen sind nicht unifizierbar.*

Eine Gleichungsmenge E heißt *gelöst* genau dann, wenn sie die Form

$$E = \{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$$

hat mit

1. $x_1, \ldots, x_n$ paarweise verschieden.
2. $\forall i, j \in \{1, \ldots, n\}\colon\ x_i$ kommt nicht in t_j vor.

$$
\begin{array}{lrcl}
(1) & E,\ t \doteq t & \overset{\mathrm{u}}{\to} & E \\
(2) & E,\ f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n) & \overset{\mathrm{u}}{\to} & E,\ s_1 \doteq t_1,\ \dots,\ s_n \doteq t_n \\
(3) & E,\ x \doteq t & \overset{\mathrm{u}}{\to} & E[t/x],\ x \doteq t \\
 & & & \text{falls } x \text{ in } E \text{ vorkommt und in } t \text{ nicht vorkommt} \\
(4) & E,\ f(t_1, \dots, t_n) \doteq x & \overset{\mathrm{u}}{\to} & E,\ x \doteq f(t_1, \dots, t_n)
\end{array}
$$

Abbildung 10.6: Unifikationsregeln

Proposition 10.8.6 *Sei $E = \{x_1 \doteq t_1, \dots, x_n \doteq t_n\}$ eine gelöste Gleichungsmenge. Dann ist*

$$\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$$

ein prinzipaler Unifikator von E.

Abbildung 10.6 zeigt Vereinfachungsregeln für Gleichungsmengen, die man als *Unifikationsregeln* bezeichnet. Wir schreiben $E \overset{\mathrm{u}}{\to} E'$ genau dann, wenn die Gleichungsmenge E' aus der Gleichungsmenge E durch die Anwendung einer Vereinfachungsregel erhalten werden kann.

Satz 10.8.7 (Unifikation) *Sei E eine Gleichungsmenge. Dann:*

1. *Wenn $E \overset{\mathrm{u}}{\to} E'$, dann $\mathcal{U}[\![E]\!] = \mathcal{U}[\![E']\!]$.*
2. *E ist widerlegt oder gelöst genau dann, wenn es kein E' mit $E \overset{\mathrm{u}}{\to} E'$ gibt.*
3. *Es gibt keine unendliche Kette $E \overset{\mathrm{u}}{\to} E_1 \overset{\mathrm{u}}{\to} E_2 \overset{\mathrm{u}}{\to} \cdots$.*

Gegeben eine Gleichungsmenge E, kann man mit den Unifikationsregeln entscheiden, ob E unifizierbar ist. Dazu wendet man die Unifikationsregel solange auf E an, bis man eine Gleichungsmenge E' bekommt, die nicht weiter vereinfacht werden kann. Dann hat E' genau dieselben Unifikatoren wie E. Außerdem ist E' entweder widerlegt oder gelöst. Wenn E' widerlegt ist, dann ist E nicht unifizierbar. Wenn E' gelöst ist, dann liefert E' gemäß Proposition 10.8.6 einen prinzipalen Unifikator für E.

Es kann passieren, dass das vereinfachte Gleichungssystem exponentiell größer ist als das Ausgangssystem (wegen Regel (3)). Die Aufblähung kann man in der Praxis leicht vermeiden, indem man die Terme durch die Knoten eines Graphens repräsentiert. Man kann zeigen, dass man die Unifizierbarkeit einer Gleichungsmenge mit linearer Zeit entscheiden kann.

$$\frac{C \in S}{C \in Res(S)}$$

$$\frac{A \in C_1 \in Res(S) \qquad \neg B \in C_2 \in Res(S) \qquad \theta_1 A = \theta_2 B \qquad \theta_1, \theta_2 \in Sub}{(\theta_1 C_1 - \{\theta_1 A\}) \cup (\theta_2 C_2 - \{\neg\theta_2 B\}) \in Res(S)}$$

Abbildung 10.7: Naiver prädikatenlogischer Resolutionsabschluss

10.9 Prädikatenlogische Resolution

Wir verallgemeinern jetzt die aussagenlogische Resolventenbildung so, dass man damit die Unerfüllbarkeit von prädikatenlogischen Klauselmengen zeigen kann.

Wie im Abschnitt über Unifikation lassen wir den Signaturindex Σ der Übersichtlichkeit halber weg und nehmen an, dass alle syntaktischen Objekte gemäß einer Signatur Σ gebildet sind, die wenigstens eine Konstante enthält.

Äquivalenz von zwei Klauselmengen S, $S' \in Cla$ definieren wir wie folgt:

$$S \models\!\!\!\dashv S' \;\overset{\text{def}}{\Longleftrightarrow}\; S \text{ und } S' \text{ haben dieselben } \Sigma\text{-Modelle}$$

Die Menge der quantorenfreien Klauseln bezeichnen wir mit *QCla*.

Abbildung 10.7 definiert mithilfe zweier Inferenzregeln den *naiven prädikatenlogischen Resolutionsabschluss* $Res(S)$ einer quantorenfreien Klauselmenge $S \subseteq QCla$. Die durch die zweite Inferenzregel eingeführten Klauseln heißen wie im aussagenlogischen Fall *Resolventen*. Für variablenfreie Klauselmengen ist der naive prädikatenlogische Resolutionsabschluss identisch mit dem aussagenlogischen Resolutionsabschluss.

Proposition 10.9.1 *Sei S eine quantorenfreie Klauselmenge. Dann $Res(S) \models\!\!\!\dashv S$.*

Beweis Folgt aus Satz 10.8.4 und der Tatsache, dass

$$(A \vee B_1) \;\wedge\; (\neg A \vee B_2) \;\Rightarrow\; B_1 \vee B_2$$

aussagenlogisch gültig ist. □

Satz 10.9.2 (Resolution) *Eine literale Klauselmenge S ist unerfüllbar genau dann, wenn $\emptyset \in Res(S)$.*

Beweis Folgt aus Proposition 10.9.1, dem Expansionssatz 10.7.6, der Gleichung $Res(Exp_\Sigma(S)) = Res(S) \cup Exp_\Sigma(S)$ und dem aussagenlogischen Resolutionssatz 9.12.12. □

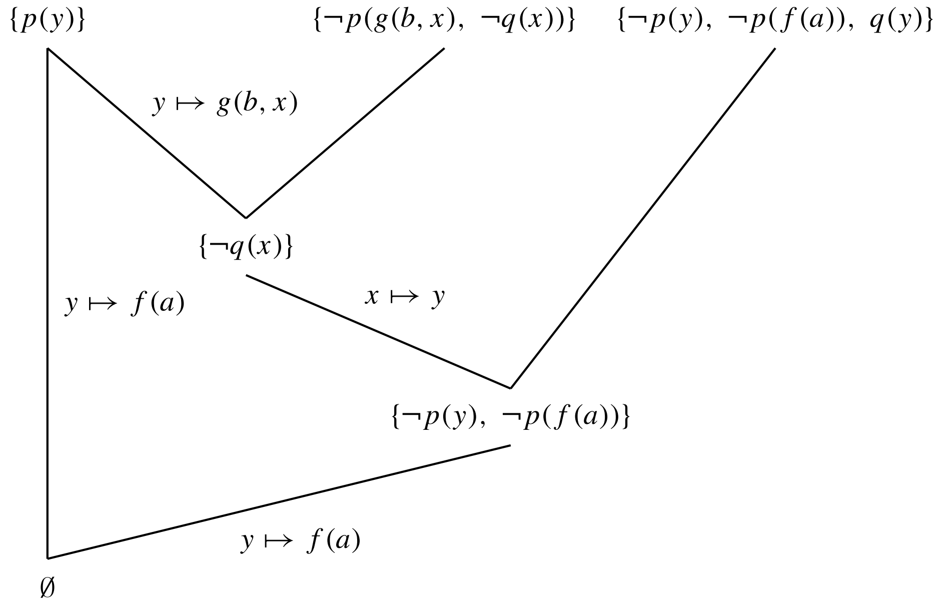


Abbildung 10.8: Ein prädikatenlogischer Resolutionsgraph

Abbildung 10.8 zeigt einen prädikatenlogischen Resolutionsgraphen. Der Resolutionssatz garantiert uns, dass die Unerfüllbarkeit jeder literalen Klauselmenge durch einen endlichen Resolutionsgraphen gezeigt werden kann.

Ein *Beweiser* ist ein Programm, das versucht, zu einer literalen Klauselmenge einen Resolutionsgraphen zu konstruieren, der zeigt, dass die Klauselmenge uenerfüllbar ist. Der Resolutionssatz sagt, dass ein solcher Graph genau dann existiert, wenn die Klauselmenge unerfüllbar ist. Die Resolutionsregel in Abbildung 10.7 hat in Bezug auf eine praktikable Realisierung des Beweisers jedoch das Problem, dass sie zu zwei Klauseln im Allgemeinen unendlich viele Resolventen generiert. Der Informatiker J.A. Robinson hat 1964 entdeckt, dass man die zu betrachtenden Resolventen auf endlich viele allgemeinste Resolventen einschränken kann, die man mithilfe von Unifikation aus den beiden Elternklauseln berechnen kann. Aufbauend auf dieser Grundlage kann man mit weiteren Techniken praktikable Beweiser realisieren.

Mit *GCla* bezeichnen wir die Menge aller variablenfreien Klauseln. Die Funktion

$$GR \in GCla \times GCla \to \mathcal{P}(GCla)$$
$$GR(C_1, C_2) = \{\, (C_1 - \{A\}) \cup (C_2 - \{\neg A\}) \mid A \in C_1,\ \neg A \in C_2 \,\}$$

liefert zu zwei variablenfreien Klauseln die Menge aller ihrer Resolventen.

Mit *LCla* bezeichnen wir die Menge aller literalen Klauseln. Ein *Resolvierer* ist

$$\frac{C \in S}{C \in Res(r, S)} \qquad \frac{C_1 \in Res(r, S) \qquad C_2 \in Res(r, S) \qquad C \in r(C_1, C_2)}{C \in Res(r, S)}$$

Abbildung 10.9: Resolvierer-basierter Resolutionsabschluss

eine Funktion

$$r \in LCla \times LCla \to \mathcal{P}(LCla)$$

die zu zwei literalen Klauseln eine Menge von literalen Klauseln wie folgt liefert:

$$Exp(r(C_1, C_2)) = \bigcup \{\, GR(C_1', C_2') \mid C_1' \in Exp\{C_1\},\ C_2' \in Exp\{C_2\} \,\}$$

Abbildung 10.9 definiert zu einem Resolvierer r und einer literalen Klauselmenge S eine literale Klauselmenge $Res(r, S)$.

Der *triviale Resolvierer*

$$\begin{aligned} r_0(C_1, C_2) = \{\, & (\theta_1 C_1 - \{\theta_1 A\}) \cup (\theta_2 C_2 - \{\neg\theta_2 B\}) \mid \\ & A \in C_1 \in Res(S),\ \neg B \in C_2 \in Res(S), \\ & \theta_1 A = \theta_2 B,\ \theta_1 \in Sub,\ \theta_2 \in Sub \,\} \end{aligned}$$

entspricht der Resolutionsregel in Abbildung 10.7. Offensichtlich gilt:

$$\forall S \subseteq LCla\colon\ Res(S) = Res(r_0, S)$$

Satz 10.9.3 (Resolution) *Sei r ein Resolvierer. Dann ist eine literale Klauselmenge S genau dann unerfüllbar, wenn $\emptyset \in Res(r, S)$.*

Eine Substitution θ heißt *Unifikator* einer literalen Klausel C genau dann, wenn θC aus genau einem Literal besteht. Ein Unifikator θ einer literalen Klausel C heißt *allgemeinster Unifikator* von C genau dann, wenn für jeden Unifikator ψ von C eine Substitution ϕ mit $\psi = \phi \circ \theta$ existiert. Offensichtlich kann man das Unifikationsproblem für Klauseln auf das Unifikationsproblem für Gleichungsmengen zurückführen. Wir wissen also, wie man die Unifizierbarkeit einer Klauselmenge entscheidet und wie man gegebenfalls einen allgemeinsten Unifikator berechnet.

Wir wählen ein Objekt $\perp \notin Sub$. Ein *Unifizierer* ist eine Funktion (*mgu* steht für most general unifier)

$$mgu \in LCla \to Sub \cup \{\perp\}$$

so dass für jede literale Klausel C gilt:

1. C unifizierbar genau dann, wenn $mgu(C) \neq \bot$
2. Wenn C unifizierbar ist, dann ist $mgu(C)$ ein allgemeinster Unifikator von C.

Ein Klausel $C \in QCla$ heißt *Variante* einer Klausel $D \in QCla$ genau dann, wenn es eine Umbenennung θ gibt mit $C = \theta D$.

Zwei Klausel heißen *variablendisjunkt* genau dann, wenn es keine Variable gibt, die in beiden Klauseln vorkommt. Ein *Umbenenner* ist eine Funktion

$$ren \in LCla \times LCla \to LCla$$

so dass für alle literalen Klauseln C_1 und C_2 die Klausel $ren(C_1, C_2)$ eine Variante von C_1 ist, die variablendisjunkt zu C_2 ist.

Satz 10.9.4 (Resolution) *Sei mgu ein Unifizierer und ren ein Umbenenner. Dann ist*

$$\begin{aligned} r(C_1, C_2) = \{\, (\theta C_1 - \theta C_1') \cup (\theta C_2 - \theta C_2') \mid \\ & \emptyset \neq C_1' \subseteq ren(C_1, C_2), \\ & \emptyset \neq C_2' = \{\neg B_1, \ldots, \neg B_n\} \subseteq C_2 \\ & \theta = mgu(C_1' \cup \{B_1, \ldots, B_n\}) \neq \bot \,\} \end{aligned}$$

ein Resolvierer.

Der in diesem Satz angegebene Resolvierer hat die für die Realisierung eines Beweisers wichtige Eigenschaft, dass er zu zwei Klausel nur endlich viele Resolventen liefert.

10.10 Allgemeingültigkeit ist unentscheidbar

Satz 10.10.1 (Unentscheidbarkeit) *Sei Σ eine Signatur mit mindestens einer Konstante, zwei einstelligen Funktionssymbolen, und einem zweistelligen Prädikatensymbol. Dann ist die Menge $\{ A \in For_\Sigma \mid A$ allgemeingültig $\}$ unentscheidbar.*

Die Satz wurde um 1936 erstmals von Alonzo Church gezeigt. Er gilt auch für Signaturen ohne Funktionssymbole, die genügend viele Prädikatensymbole haben.

Um den Unentscheidbarkeitssatz zu zeigen, benötigen wir ein unentscheidbares, aber testbares Problem, dass wir auf das Allgemeingültigkeitsproblem reduzieren können (das Allgemeingültigkeitsproblem ist testbar, siehe Satz 10.7.10). Dafür eignet sich beispielsweise das Halteproblem für IMP (Satz 8.5.3). Wir können aber viel Arbeit sparen, wenn wir ein anderes als unentscheidbar aber testbar bekanntes Problem wählen, das sich einfacher mit prädikatenlogischen Formeln codieren lässt. Zum Beispiel ist das Postsche Korrespondenzproblem dafür gut geeignet. Einen entsprechenden Beweis finden Sie in [Schöning, Kapitel 2.3].

Ein *Beweissystem* für eine Menge U ist ein Paar (B, β) wie folgt:

1. B ist eine in polynomialer Zeit entscheidbare Menge.
2. $\beta \in B \to U$ ist eine in polynomialer Zeit berechenbare surjektive Funktion.

Dabei heißt $b \in B$ *Beweis für* $u \in U$ genau dann, wenn $\beta(b) = u$. Offensichtlich ist jede Menge, zu der ein Beweissystem existiert, testbar.

Für die Menge der allgemeingültigen Formeln können wir ein Beweissystem wie folgt konstruieren. Als Beweise (die Elemente der Menge B) wählen wir Tripel wie folgt:

1. Eine Formel A.
2. Eine Ableitung $\{\{\exists\neg A\}\} \stackrel{s}{\to} \cdots \stackrel{s}{\to} S$, so dass S eine literale Klauselmenge ist.
3. Ein Resolutionsgraph, der aus S die leere Klausel ableitet.

10.11 Theorien und Axiomatisierung

In diesem Abschnitt arbeiten wir wieder mit mehr als einer Signatur.

Eine Σ-*Theorie* ist eine Formelmenge $T \subseteq For_\Sigma$, so dass

$$T = \{ A \in For_\Sigma \mid T \models_\Sigma A \}$$

gilt. Eine Theorie ist also eine unter logischer Konsequenz abgeschlossene Formelmenge. Eine Σ-Theorie T heißt *vollständig* genau dann, wenn für jede geschlossene Formel $A \in For_\Sigma$ gilt: $A \in T$ oder $\neg A \in T$.

Sei $\mathcal{A}$ eine Σ-Struktur. Dann bezeichnen wir die Formelmenge

$$Th(\mathcal{A}) \stackrel{\text{def}}{=} \{ A \in For_\Sigma \mid A \text{ gültig in } \mathcal{A} \}$$

als die *Theorie von* $\mathcal{A}$. Offensichtlich ist $Th(\mathcal{A})$ eine vollständige Σ-Theorie.

Sei $M \subseteq For_\Sigma$. Dann bezeichnen wir

$$Th_\Sigma(M) \stackrel{\text{def}}{=} \{\, A \in For_\Sigma \mid M \models_\Sigma A \,\}$$

als die *von M erzeugte Theorie*.

Proposition 10.11.1 *Sei $M \subseteq For_\Sigma$. Dann $M \subseteq Th_\Sigma(M) = Th_\Sigma(Th_\Sigma(M))$. Außerdem ist M genau dann eine Σ-Theorie, wenn $M = Th_\Sigma(M)$ gilt.*

Satz 10.11.2 (Testbarkeit) *Sei $M \subseteq For_\Sigma$ testbar. Dann ist $Th_\Sigma(M)$ testbar.*

Beweis Folgt aus Proposition 10.2.7 und Satz 10.7.10. □

Proposition 10.11.3 *Eine vollständige Theorie ist testbar genau dann, wenn sie entscheidbar ist.*

Beweis Sei T eine testbare und vollständige Σ-Theorie. Sei $A \in For_\Sigma$ eine geschlossene Formel. Wir können einen steuerbaren Tester für T abwechselnd auf A und $\neg A$ anwenden. Da T vollständig ist, wissen wir nach endlich vielen Schritten, ob A oder $\neg A$ in T ist. □

Eine *Axiomatisierung* einer Σ-Theorie T ist eine entscheidbare Formelmenge $M \subseteq For_\Sigma$ mit $T = Th_\Sigma(M)$.

Proposition 10.11.4 *Sei M eine Axiomatisierung einer Σ-Theorie T. Dann ist T testbar. Wenn T vollständig ist, dann ist T sogar entscheidbar.*

Beweis Folgt aus Satz 10.11.2 und Proposition 10.11.3. □

Von besonderem Interesse sind *minimale Axiomatisierungen*. Dabei heißt eine Formelmege $M \subseteq For_\Sigma$ *minimal* genau dann, wenn

$$\forall A \in M\colon\; Th_\Sigma(M - \{A\}) \neq Th_\Sigma(M)$$

Um 1900 glaubten viele Logiker (zum Beispiel David Hilbert), dass sich die üblichen mathematischen Theorien in Prädikatenlogik erster Stufe axiomatisieren lassen. Dieser Glaube war etwas verwegen, da man das Konzept der Berechenbarkeit noch gar nicht formal definieren konnte (das gelang erst Church und Turing um 1937). Gödel zeigte um 1933 mit seinem Unvollständigkeitssatz 8.6.1, dass die Theorie der ganzen Zahlen mit der Signatur $\{0, 1, -1, +, *, =\}$ nicht axiomatisierbar ist.

Es gibt aber durchaus interessante Strukturen, deren Theorie axiomatisierbar ist. Beispielweise ist die Theorie der natürlichen Zahlen mit der Signatur $\{0, 1, +, =\}$ entscheidbar (sogenannte Presburger Arithmetik). Dieses Ergebnis wurde von Presburger, einem Diplomanden Tarskis, mit der Methode der Quantorenelimination gezeigt.

Auch die Theorie der reellen Zahlen mit der Signatur $\{0, 1, -1, +, *, =\}$ ist entscheidbar. Dieses Ergebnis wurde von Tarski ebenfalls mit der Methode der Quantorenelimination gezeigt. Der Satz von Löwenheim-Skolem 10.7.8 liefert uns das auf den ersten Blick überraschende Ergebnis, dass die Theorie der reellen Zahlen auch ein abzählbares Modell hat.

Die gerade aufgeführten Resulte für die Theorie der reellen Zahlen sind nur auf den ersten Blick verblüffend. Sie sagen im Wesentlichen, dass die Expressivität (Aussdrucksstärke) von prädikatenlogischen Sprachen erster Stufe begrenzt ist in dem Sinne, dass ihre Formeln wesentliche Eigenschaften der reellen Zahlen nicht ausdrücken können. Diese Schwierigkeit verschwindet, wenn man mit prädikatenlogischen Sprachen höherer Ordnung arbeitet, die auch über Funktionen und Prädikate quantifizieren können.

Wir können also einerseits feststellen, dass die Expressivität der Prädikatenlogik erster Stufe eingeschränkt ist. Andererseits ist sie aus der Sicht der Berechenbarkeit nicht gerade bescheiden: die Theorie der ganzen Zahlen ist nicht testbar und Allgemeingültigkeit ist immerhin unentscheidbar. Wenn man zu expressiveren Sprachen übergeht, bezahlt man damit, dass Allgemeingültigkeit nicht mehr testbar ist.

Bisher haben wir nur die vollständigen Theorien konkreter Strukturen wie $\mathbb{Z}$ und $\mathbb{R}$ betrachtet. Beim mathematischen Arbeiten verwendet man jedoch oft Abstraktionen wie Gruppen oder partielle Ordnungen. Diese Abstraktionen entsprechen unvollständigen Theorien und lassen sich oft mit prädikatenlogischen Sprachen erster Stufe axiomatisieren. Dabei verwendet man typischerweise prädikatenlogische Sprachen mit Gleichheit. Diese stellen atomare Formeln $s = t$ bereit, die als Gleichheitsaussagen interpretiert werden. Man kann zeigen, dass man das eingebaute Gleichheitsprädikat mithilfe eines zweistelligen Prädikatensymbols axiomatisieren kann, ohne dass sich der Begriff der Gültigkeit ändert.

Kapitel 11

Rückblick

Wir haben eine Reihe von logischen Sprachen betrachtet: die Programmiersprache IMP, die Sprache AL der Aussagenlogik, und die Familie der prädikatenlogischen Sprachen erster Ordnung (zum Beispiel ASSN). Für jede dieser Sprache haben wir syntaktische und semantische Objekte definiert. Wir haben stets eine denotationale Semantik angegeben, die syntaktische Objekte auf semantische Objekte abbildet.

Syntaktische Objekte

Syntaktische Objekte haben wir als sehr einfache mathematische Objekte formalisiert, die die Struktur von Bäumen haben. Die Baumstruktur ergibt sich dadurch, das größere syntaktische Objekte durch Tupelbildung aus kleineren syntaktischen Objekten konstruiert werden. Diese Darstellung macht die hierarchische Struktur von syntaktischen Objekten explizit. Mit der Technik der Gödelisierung lassen sich syntaktische Objekte auch als natürliche Zahlen repräsentieren. Wenn man logische Sprachen auf Computern realisiert, verwendet man eine Darstellung durch Zeichenreihen und spricht von konkreter Syntax. Die Baumdarstellung bezeichnet man dagegen als abstrakte Syntax.

Denotationale Semantik

Die für eine Sprache erforderlichen syntaktischen Mengen (zum Beispiel Kommandos, Terme und Formeln) wurden durch strukturelle Rekursion definiert. Entsprechend wurden die erforderlichen Denotationsfunktionen durch strukturelle Rekursion über die Konstruktion der syntaktischen Objekte definiert. Die Denotationsfunktion für eine syntaktische Menge liefert stets eine Äquivalenzrelation

$$S \models\!\dashv S' \iff \mathcal{D}[\![S]\!] = \mathcal{D}[\![S']\!]$$

die die Ersetzungseigenschaft hat: Wenn $S \models\!\!\!\dashv S'$ und das syntaktische Objekt T' aus dem syntaktischen Objekt T erhalten wird, indem ein Auftreten des Teilobjekts S durch S' ersetzt wird, dann $T \models\!\!\!\dashv T'$. Damit erhalten wir eine abstrakte Gleichheitsrelation, mit der wir nach den wohlvertrauten Regeln für Gleichheit rechnen können.

Bis auf die Kommandos von IMP war die Definition der Denotationsfunktionen immer offensichtlich. Die Schwierigkeit bei IMP war, dass die Denotation einer Schleife eine Funktion ist, die durch eine etwas allgemeinere Rekursion definiert werden muss. Für solche rekursive Definitionen haben wir durch VPOs und Fixpunktoperatoren eine mathematische Grundlage bereitgestellt.

Operationale Semantik

Da die denotationale Semantik für IMP für den mathematisch Naiven nicht sofort verständlich ist, haben wir für IMP eine alternative operationale Semantik angegeben, die unmittelbar verständlich ist. Wir haben dann gezeigt, dass die beiden Semantiken äquivalent sind. Die denotationale Semantik hat gegenüber der operationalen Semantik den großen Vorteil, dass sie die Äquivalenz von Kommandos so direkt wie nur möglich definiert.

Hoare-Logik

Die prädikatenlogische Sprache ASSN versetzte uns in die Lage, Spezifikationen für die Kommandos von IMP zu formulieren. Unser Hauptinteresse galt der Verifikation von partieller Korrektheit. Die Hoare-Regeln versetzen uns in die Lage, die partielle Korrektheit eines Kommandos zu verifizieren, ohne auf die Definition der Semantik von IMP zurückgreifen zu müssen. Für eine Verifikation muss man geeignete Schleifeninvarianten angeben. Aus dem annotierten Programm lassen sich dann die sogenannten Verifikationsbedingungen berechnen. Dabei handelt es sich um Formeln von ASSN. Die partielle Korrektheit ist verifiziert, wenn alle Verifikationsbedingungen gültig sind.

Wir haben gezeigt, dass die durch die Hoare-Regeln gegebene Verifikationsmethode vollständig ist. Dazu haben wir einen nichttivialen Algorithmus angeben, der zu einer Nachbedingung und einem Kommando eine schwächste Vorbedingung berechnet. Die Existenz der Vorbedingung bedeutet, dass die Spezifikationssprache ASSN hinreichend expressiv ist, um die für ein Kommando und eine Nachbedingung zulässigen Eingaben zu charakterisieren.

Berechenbarkeit

Aufbauend auf IMP haben wir die grundlegenden Begriffe der Berechenbarkeit formalisiert. Mit der Eindeutigkeit der Primzahlzerlegung und dem Konzept der Gödelisierung sieht man leicht, dass IMP Turingmaschinen simulieren kann. Damit entsprechen die mit IMP definierten Berechnungskonzepte den Üblichen. Die Existenz schwächster Vorbedingungen lieferte uns dann die Erkenntnis, dass die Menge der gültigen Formeln von ASSN nicht testbar ist (Gödels Unvollständigkeitssatz).

Aussagenlogik

Bei der Aussagenlogik galt unser Hauptinteresse algorithmischen Methoden, um die Gültigkeit, Erfüllbarkeit und Äquivalenz von Formeln zu entscheiden. Dafür haben wir zwei Vereinfachungssysteme[1] angegeben, die Formeln in disjunktive beziehungsweise konjunktive Normalform überführen. Im Zusammenhang damit haben wir Tableaus und Sequenten betrachtet. Zusätzlich haben wir Resolution betrachtet, mit der wir eindeutige disjunktive und konjunktive Normalformen berechnen können (sogenannte Primdarstellungen). Schließlich haben wir die Kompaktheitseigenschaft unendlicher Formelmengen kennengelernt, die für die Prädikatenlogik eine wichtige Rolle spielt.

Prädikatenlogik

Für die Prädikatenlogik haben wir zuerst gezeigt, dass die aussagelogische Sprache eine Teilsprache jeder prädikatenlogischen Sprache ist. Damit sind alle aussagenlogischen Methoden für die Prädikatenlogik verwendbar. Als Nächstes haben wir gezeigt, dass die Menge der allgemeingültigen Formeln testbar und unentscheidbar ist. Für die Testbarkeit haben wir Unerfüllbarkeit betrachtet. Der erste Schritt beim Testen ist die Überführung einer Formel in quantorenfreie Klauselform. Dies gelingt durch die Einführung von Skolem-Funktionen. Wir haben dann gezeigt, dass eine quantorenfreie Klauselmenge genau dann unerfüllbar ist, wenn die (unendliche) Menge ihrer Grundinstanzen unerfüllbar ist. Aussagenlogische Kompaktheit und Resolution liefern uns dann den zweiten Schritt eines Testers auf Unerfüllbarkeit. Mithilfe der Unifikationsoperation kann man eine prädikatenlogische Version der Resolution angeben, die eine Grundlage für praktikable Beweiser liefert.

[1] In der Literatur heißen Vereinfachungssysteme meistens Reduktionssysteme.