Mathematische Aspekte der Programmiersprachen

Elfriede Fehr

Inhaltsangabe

0 Einleitung

Programmiersprachen sind *künstliche (formale) Sprachen*, die der Kommmunikation zwischen Mensch und Maschine dienen. Künstliche Sprachen wurden in der Mathematik und Philosophie bereits Ende des 19. Jahrhunderts entwickelt, um Aussagen und Schlußfolgerungen formal beschreiben und analysieren zu können. Im Kapitel über Algorithmen haben wir bereits verschiedene Möglichkeiten kennengelernt, Algorithmen zu formulieren. Es ist auch klar geworden, daß die dort verwendeten sprachlichen Formulierungen stark von der zugrunde liegenden "Maschine" abhängen. Wenn Sie etwa ein Registermaschinenprogramm zur Formulierung eines Algorithmus entworfen haben, dann läßt sich dieses Programm zwar auf einer Registermaschine aber nicht unmittelbar auf einer Turingmaschine ausführen. Dies liegt daran, daß die Speicherkonzepte (Register vs. Band) und die Basisfähigkeiten dieser Maschinen unterschiedlich sind und daher auch die Befehlsvorräte verschieden sind. Auch die Strukturen realer Rechner sind uneinheitlich, dennoch ist es zur Formulierung von Algorithmen wünschenswert, sich maschinenunabhängig ausdrücken zu können und möglichst dem **Menschen** intuitiv verständliche sprachliche Formulierungen verwenden zu können. Mit der Entwicklung *höherer Programmiersprachen* wurde diesen Wünschen Rechnung getragen. Ein immer noch lebendiger Zweig der Informatik beschäftigt sich mit der Konstruktion von *Übersetzern*, das sind Programme, die zu gegebenen realen Maschinen aus einem höheren Programm ein ausführbares Maschinenprogramm erzeugen. Dazu gehören natürlich formale Beschreibungsmittel sowohl für die *Syntax* (grammatikalische Struktur) als auch für die *Semantik* (Bedeutung) von Programmiersprachen.

1 Formale Syntax

Der amerikanische Linguist Noam Chomsky hat im Jahre 1959 eine allgemeine Form zur syntaktischen Definition formaler Sprachen vorgestellt:

Definition 1.0.1 *Eine* Grammatik G *für eine Sprache* L *besteht aus einem Viertupel*

$$G = (V, T, P, S)$$

mit folgenden Eigenschaften:

- V *ist eine endliche, nichtleere Menge von Zeichen, den* Nichtterminalsymbolen *(auch syntaktische Variablen genannt).*
- T *ist eine endliche Menge von Zeichen, den* Terminalsymbolen, *wobei* V *und* T *disjunkt sind, d.h.* $V \cap T = \emptyset$.
- P *ist eine endliche Menge von* Produktionen, *das sind Paare der Form* $\alpha \to \beta$, *wobei gilt:* α *und* β *sind Wörter aus Nichtterminal- und Terminalsymbolen, d.h.* $\alpha, \beta \in (V \cup T)^\star$, *wobei* α *mindestens ein Terminalsymbol enthalten muß.*
- S *ist ein Element aus* V *und heißt Startsymbol.* □

Nun läßt sich eine formale Sprache syntaktisch durch eine Grammatik $G = (V, T, P, S)$ festlegen, indem man vereinbart: alle aus S mittels P ableitbaren Wörter, die nur aus Nichtterminalsymbolen bestehen, gehören zu L und sonst keine.
Dabei läßt sich der Ableitungsbegriff wie folgt präzisieren:

Definition 1.0.2 *Sei* $G = (V, T, P, S)$ *eine Grammatik, sei* w *ein Wort aus Nichtterminal- und Terminalsymbolen, d.h.* $w \in (V \cup T)^\star$, *und sei* $\alpha \to \beta$ *eine Produktion aus* P. *Wenn* α *als Teilwort in* w *vorkommt, d.h.* $w = w_1 \alpha w_2$ *mit* $w_1, w_2 \in (V \cup T)^\star$, *dann ist* $w' = w_1 \beta w_2$ *in einem Schritt aus* w ableitbar, *geschrieben* $w \underset{G}{\to} w'$. *Die reflexive, transitive Hülle von* $\underset{G}{\to}$ *bezeichnet man mit* $\overset{\star}{\underset{G}{\to}}$ *,d.*

h. für zwei Wörter w und w' gilt $w \overset{\star}{\underset{G}{\rightarrow}} w'$ genau dann, wenn $w' = w$ gilt oder w' in einem oder mehreren Schritten aus w ableitbar ist.

Die *von G erzeugte Sprache* $L(G)$ ist die Menge aller Wörter, die aus dem Startsymbol S ableitbar sind und nur aus Terminalsymbolen bestehen, d.h.

$$L(G) = \left\{ w | S \overset{\star}{\underset{G}{\rightarrow}} w \text{ und } w \in T^{\star} \right\} \qquad \square$$

Diese allgemeine Definition von Chomsky ist bereits ebenso mächtig wie der Begriff der rekursiven Aufzählbarkeit (vergleiche Abschnitt 8 aus dem Kapitel über Algorithmen). Es gilt nämlich folgender

Satz: Zu jeder Grammatik G ist die Sprache $L(G)$ rekursiv aufzählbar, und umgekehrt gibt es zu jeder rekursiv aufzählbaren Sprache L eine Grammatik G mit $L = L(G)$.

Beispiel: Wir definieren die Sprache RM der Registermaschinenprogramme (siehe Abschnitt 2.2 des Kapitels "Was können Algorithmen?") durch folgende Grammatik:

$$\begin{aligned}
G_R &= (V_R, T_R, P_R, S_R) \text{ mit} \\
V_R &= \left\{ S_R, V, M, \underline{pz}, \ \underline{\text{befehl}}, \ \underline{\text{index}}, \ \underline{\text{ziffer}} \right\} \\
T_R &= \left\{ X, Y, Z, E, A, \underline{if}, \underline{goto}, \leftarrow, +, -, 0, 1, \neq, [,], 0, 1, \ldots, 9 \right\}
\end{aligned}$$

P_R besteht aus folgenden Produktionen:

$$
\begin{array}{l}
S_R \to \underline{pz} \\
S_R \to \underline{pz}\, S_R \\
\underline{pz} \to \underline{\text{befehl}} \\
\underline{pz} \to [M]\ \underline{\text{befehl}} \\
M \to A\ \underline{\text{index}} \\
M \to E \\
\underline{\text{befehl}} \to V \leftarrow V + 1 \\
\underline{\text{befehl}} \to V \leftarrow V - 1 \\
\underline{\text{befehl}} \to \underline{if}\, V \neq 0\, \underline{goto}\, M \\
V \to X\ \underline{\text{index}} \\
V \to Z\ \underline{\text{index}} \\
V \to Y \\
\underline{\text{index}} \to \underline{\text{ziffer}} \\
\underline{\text{index}} \to \underline{\text{ziffer}}\ \underline{\text{index}} \\
\underline{\text{ziffer}} \to 0 \\
\underline{\text{ziffer}} \to 1 \\
\vdots \\
\underline{\text{ziffer}} \to 9
\end{array}
$$

Bemerkung: Durch ein zusätzliches Terminalsymbol $\underline{nz}$ (für neue Zeile), das von einem Drucker entsprechend interpretiert wird, läßt sich auch die zeilenweise Darstellung der Registermaschinenprogramme mit der zweiten modifizierten Produktion

$$S_R \to \underline{pz}\,\underline{nz}\, S_R$$

erreichen. Der Leser möge sich davon überzeugen, daß $L(G_R) \supseteq$RM gilt. □

Chomsky ordnete je nachdem, welche Form die Produktionsregeln aufweisen, die Grammatiken in verschiedene Klassen ein. Er unterschied die regulären, die kontextfreien und die kontextsensitiven Grammatiken.
Für die Syntax von Programmiersprachen werden überwiegend *kontextfreie Grammatiken* verwendet, die dadurch charakterisert sind, daß die linken Seiten aller Produktionen jeweils aus genau einem Nichtterminalsymbol bestehen. Man beachte, daß die Beispielgrammatik G_R kontextfrei ist. Dies hat zur Folge, daß auch Befehle der Form $V \leftarrow W + 1$ bzw. $V \leftarrow W - 1$ mit $V \neq W$ ableitbar sind. Ferner fordert man, daß die Syntax einer Programmiersprache *eindeutig* ist, d.h. zu jedem syntaktisch korrekten Programm existiert genau eine Linksableitung (das jeweils am weitesten linksstehende Nichtterminalsymbol wird ersetzt) oder anders ausgedrückt: die syntaktische Struktur (z.B. gegeben durch den Syntaxbaum) eines Programms ist eindeutig.

Beispiel: Die Grammatik $G_{ne} = (V_{ne}, T_{ne}, P_{ne}, S_{ne})$, wobei ne für $n$icht $e$indeutig steht, zur Erzeugung einfacher arithmetischer Ausdrücke mit

$$\begin{array}{lcl}
V_{ne} & = & \{\underline{exp}\} \\
T_{ne} & = & \{0, 1, +, /\} \\
P_{ne} & = & \{ \begin{array}[t]{lcl} \underline{exp} & \rightarrow & \underline{exp} + \underline{exp}, \\ \underline{exp} & \rightarrow & \underline{exp}/\underline{exp} \\ \underline{exp} & \rightarrow & 0 \\ \underline{exp} & \rightarrow & 1\} \end{array} \\
S_{ne} & = & \underline{exp}
\end{array}$$

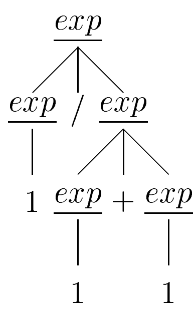
ist **nicht** eindeutig, denn das Terminalwort $1/1+1$ läßt sich durch zwei verschiedene Linksableitungen gewinnen:

a) $\underline{exp} \rightarrow \underline{exp}/\underline{exp} \rightarrow 1/\underline{exp} \rightarrow 1/\underline{exp} + \underline{exp} \rightarrow 1/1 + \underline{exp} \rightarrow 1/1+1.$

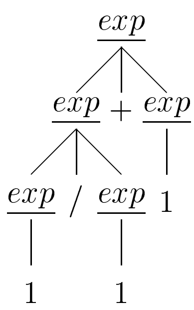
b) $\underline{exp} \rightarrow \underline{exp} + \underline{exp} \rightarrow \underline{exp}/\underline{exp} + \underline{exp} \rightarrow 1/\underline{exp} + \underline{exp} \rightarrow 1/1 + \underline{exp} \rightarrow 1/1+1.$

Die syntaktischen Strukturen gemäß dieser Ableitungen lassen sich durch folgende Syntaxbäume darstellen:

a)



b)



Es ist offensichtlich, daß bei Vereinbarung der bekannten Semantik arithmetischer Ausdrücke, der Wert der beiden Strukturen verschieden ist, nämlich a) erhält den Wert 1/2 und b) den Wert 2.

Mehrdeutigkeiten in kontextfreien Grammatiken von Programmiersprachen werden meist durch Vereinbarung von Prioritäten vermieden. Es existieren aber auch *inhärent mehrdeutige* Sprachen, d.h. alle kontextfreien Grammatiken, die eine solche Sprache erzeugen, sind mehrdeutig. Die Sprache $L = \{a^k b^m c^n / k = m \text{ oder } m = n\}$ ist ein Beispiel für eine inhärent mehrdeutige Sprache. Oft werden zur Syntaxdefinition von Programmiersprachen noch weitere Einschränkungen vorgenommen, um das Verfahren der Syntaxanalyse zu vereinfachen. Die Notation der Produktionen wurde von J. Backus und P. Naur noch vereinfacht und erstmals zur syntaktischen Beschreibung der Programmiersprache `ALGOL 60` (Algorithmic Language 1960) verwendet. Diese Notation, nach ihren Erfindern auch BNF (Backus-Naur-Form) genannt, zeichnet sich durch folgende Merkmale aus:

- Nichtterminalsymbole stehen stets in spitzen Klammern.
- Der Ableitungspfeil (oft auf einer Tastatur nicht vorhanden) wird durch das Zeichen ::= ersetzt.
- Gibt es mehrere rechte Seiten für ein Nichtterminalsymbol, so werden diese durch einen senkrechten Strich getrennt, direkt hintereinander geschrieben, in einer Regel zusammengefaßt.

Beispiel: Die Sprache $L(G_{ne})$ gegeben durch eine BNF Grammatik:

<exp> ::= <exp> + <exp> | <exp> / <exp> |0|1. □

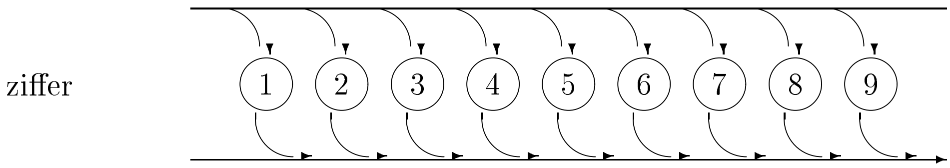
Später wurde diese Notation noch weiter vereinfacht:

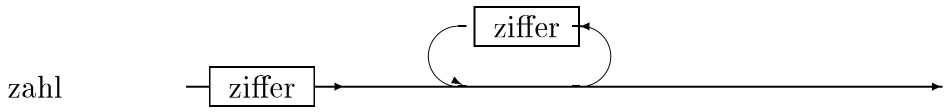
1. Optionale Symbole oder Wörter werden in eckige Klammern eingeschlossen.
2. Symbole oder Wörter, die beliebig oft wiederholt werden können, werden in geschweifte Klammern eingeschlossen.

Als Beispiel für die erweiterte BNF definieren wir die formale Syntax des Kerns einer typischen höheren Programmiersprache:

< program >	::=	begin < stat-seq > end
< stat-seq >	::=	[< statement >]{; < statement >}
< statement >	::=	< var >:=< exp > |
		if < bool-exp > then {< stat-seq >}
		else {< stat-seq >} fi |
		while< bool-exp > do {< stat-seq >} od

Der Leser möge sich geeignete Regeln für arithmetische und boolesche Ausdrücke, die aus den Nichtterminalsymbolen < exp > bzw. < bool-exp > erzeugt werden, als Übung aufschreiben.
Noch anschaulicher als die erweiterte BNF ist die Verwendung von Syntagdiagrammen, z.B. die Syntaxdiagramme für natürliche Zahlen lauten





2 Formale Semantik

Zur Formalisierung der *Semantik* höherer Programmiersprachen haben sich drei Methoden durchgesetzt:

1. Die *operationelle Semantik* verwendet eine abstrakte Maschine und definiert die Semantik eines Programms als schrittweise Zustandsänderung dieser Maschine. Die operationelle Methode eignet sich besonders gut dazu, höhere Programmiersprachen auf vorhandenen Rechnersystemen schnell verfügbar zu machen, indem die entsprechende abstrakte Maschine simuliert wird.

2. Die *denotationelle Semantik* abstrahiert von der schrittweisen Zustandsänderung einer Maschine und ordnet in statischer Weise jedem Programm die entsprechende Ein–/Ausgabefunktion als Semantik zu. Dies geschieht induktiv über den Aufbau des Programms, indem jeder Programmkomponente ein geeignetes mathematisches Objekt als Wert zugeordnet wird. Diese Objekte sind im allgemeinen diskrete Strukturen, stetige Funktionen oder Funktionale höherer Ordnung, aus denen dann die Semantik des Programms aufgebaut wird. Der entscheidende Gedanke dabei ist, daß Programmschleifen auf der Ebene der Semantikfunktionen durch Fixpunktbildung modelliert werden können; daher spricht man von *Fixpunktsemantik.*

3. Die *axiomatische Methode* abstrahiert noch einen Schritt weiter. Hier werden keine konkreten Zustände transformiert, sondern nur noch logische Aussagen über Zustände.

   Die Wirkung eines Programms P wird durch eine möglichst schwache Vorbedingung *(weakest precondition)*, gegeben durch eine logische Formel Q, und eine möglichst starke Nachbedingung, gegeben durch eine weitere Formel R, charakterisiert. Man notiert diesen Zusammenhang in der Klausel

$$\{Q\}P\{R\}$$

und liest:

Wenn die Bedingung Q für einen Anfangszustand erfüllt ist, dann gilt nach Ausführung von P die Bedingung R für den Endzustand.

Die Vorgehensweise der axiomatischen Semantikspezifikation einer Programmiersprache ist nun wie folgt: Jedem atomaren Bestandteil einer Sprache wird ein Axiom, eine elementare Relation zwischen Vor- und Nachbedingungen, zugeordnet. Jedem zusammengesetzten Programmteil wird eine Ableitungsregel zugeordnet, mit deren Hilfe sich aus den Vor- und Nachbedingungen seiner Komponenten die Vor- und Nachbedingung dieses Programmteils gewinnen läßt. Diese Methode wurde von Hoare [Ho69] entwickelt und eignet sich besonders gut zur Verifikation von Programmen. Eine ausführliche Behandlung der Programmverifikation findet man in dem Lehrbuch von Apt und Olderog [Ap94] .

Da die denotationelle Semantik die Bedeutung von Programmen besonders übersichtlich und vollständig formalisiert, soll die Semantik unserer kleinen Beispielsprache aus Abschnitt 2 hier denotationell spezifiziert werden. Der Einfachheit halber befassen wir uns hier nicht mit den Konzepten für die Ein-/Ausgabe und legen ein simples Zustandskonzept zugrunde, in dem ein Zustand ausschließlich durch die Zuordnung von Variablen zu Werten gegeben ist. Mathematisch formuliert:

$$\mathtt{ZUSTAND} = (\mathtt{VAR} \to \mathtt{VALUE}),$$

wobei `VAR` die Menge der Variablen und `VALUE` die Menge der Werte (z.B. darstellbare Zahlen) bezeichnet.
Angenommen, wir kennen die Bedeutung der Ausdrücke und der booleschen Ausdrücke zu gegebenem Zustand z in diesem Wertebereich `VALUE` und bezeichnen den Wert eines Ausdruckes e mit $[\![e]\!]z$, dann können wir die Semantik der Anweisungen induktiv als Zustandstransformationen definieren:

$[\![v := e]\!]z \quad = \quad z[v/[\![e]\!]z]$ (Hier bezeichnet $z[v/[\![e]\!]z]$ den Zustand, der allen Variablen außer v den gleichen Wert zuordnet, wie z und der Variablen v gerade den Wert des Ausdrucks e zuordnet.)

$$[\![\text{if } b \text{ then } s_1 \text{ else } s_2]\!]z = \begin{cases} [\![s_1]\!]z, & \text{falls } [\![b]\!]z \text{ wahr ist} \\ [\![s_2]\!]z, & \text{falls } [\![b]\!]z \text{ falsch ist} \end{cases}$$

$$[\![\text{while } b \text{ do } s \text{ od}]\!]z = \begin{cases} [\![s;\ \text{while } b \text{ do } s \text{ od}]\!]z, & \text{falls } [\![b]\!]z \text{ wahr ist} \\ z, sonst \end{cases}$$

$$[\![s_1; s_2]\!]z = [\![s_2]\!]\,([\![s_1]\!]z)\,.$$

Der Leser möge auch hier mit derselben Methode die Semantik der Ausdrücke induktiv als Übung definieren. Zur denotationellen Semantik weiterer Konzepte höherer Programmiersprachen sei auf [Fe89] verwiesen.

3 Grundlagen der Logik-Programmierung

In diesem Abschnitt soll in informeller Weise der Übergang von der Prädikatenlogik 1. Stufe über *Hornklausen* zur Sprache `PROLOG` und ihrer Auswertungsmechanismen geschildert werden. Dabei soll nur der mit der Prädikatenlogik konsistente Teil von `PROLOG`, manchmal auch Kern-`PROLOG` genannt, berücksichtigt werden.

3.1 Von der Prädikatenlogik 1. Stufe zu PROLOG

In der Sprache erster Stufe benutzt man

- Prädikatssymbole $p, q, \ldots$ mit Stelligkeit > 0
- Funktionssymbole $f, g, \ldots$ mit Stelligkeit > 0
- Konstanten $a, b, c, \ldots$
- Variablen $x, y, z, \ldots$
- Junktoren $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ (nicht, und, oder, folgt, genau dann wenn)
- Quantoren $\forall$ und $\exists$ (für alle, es gibt).

Aus Variablen, Konstanten und Funktionssymbolen werden wie üblich *Terme* aufgebaut, aus jeweils einem Prädikatssymbol und Termen erhält man *atomare Formeln*. Die *aussagenlogischen Formeln* gewinnt man durch Verwendung der Junktoren und schließlich erhält man die *prädikatenlogischen Formeln* durch Abschluß unter Quantoren, die jeweils auf eine Variable und eine Formel angewendet werden.

Bemerkung: Häufig wird die Gleichheit als fest interpretiertes Standardprädikat hinzugenommnen. *Klausen* sind Formeln spezieller Gestalt. Sie werden notiert in der Form

$$B_1, \ldots, B_m \leftarrow A_1, \ldots, A_n$$

wobei die A_i und B_i atomar sind. Eine solche Klause steht als Abkürzung für

$$\forall \vec{x}.\ (A_1 \wedge \cdots \wedge A_n \rightarrow B_1 \vee \cdots \vee B_m)$$

oder auch

$$\forall \vec{x}.\ (B_1 \vee \cdots \vee B_m \vee \neg A_1 \vee \cdots \vee \neg A_n)$$

wobei $\vec{x}$ alle Variablen unfaßt, die in den A_i, B_i auftreten.
Sonderfälle sind:

$m \leq 1$:	*Hornklausen* benannt nach Alfred Horn, der solche Formeln 1951 in algebraischem Kontext untersucht hat.
$m = n = 0$:	Die *leere Klause* $\square$ mit der Konvention, daß $\square$ in keiner Interpretation gilt.

In `PROLOG` werden ausschließlich Hornklausen, mit $m = 1$ verwendet, wobei man für den Fall $n = 0$ von *Fakten* und für den Fall $n > 0$ von *Regeln* spricht. Eine Reduktion allgemeiner Formeln erster Stufe auf Klausen leistet der folgende

Satz: Für jede Formel ϕ erster Stufe gibt es eine Menge Φ von Klausen, so daß gilt:

$$\phi \text{ erfüllbar } \Leftrightarrow \Phi \text{ erfüllbar.}$$

Der **Beweis** erfolgt durch Angabe eines Verfahrens zur Gewinnung von Φ:

1. Ersetze $\phi_0 \rightarrow \phi_1$ durch $\neg\phi_0 \vee \phi_1$, analog $\leftrightarrow$.

2. Ziehe $\neg$ nach innen, durch Übergang von

$$\begin{array}{rcl} \neg\neg\phi_0 & \text{zu} & \phi_0 \\ \neg(\phi_0 \wedge \phi_1) & \text{zu} & \neg\phi_0 \vee \neg\phi_1 \\ \neg(\phi_0 \vee \phi_1) & \text{zu} & \neg\phi_0 \wedge \neg\phi_1 \\ \neg\forall x.\phi_0 & \text{zu} & \exists x.\neg\phi_0 \\ \neg\exists x.\phi_0 & \text{zu} & \forall x.\neg\phi_0 \end{array}$$

3. Benenne quantifizierte Variablen um, sodaß verschiedene Quantoren auf verschiedene Variablen wirken.

4. Eliminiere $\exists$ durch Skolemfunktionen (zuerst verwandt in einer Arbeit des Norwegers Thoralf Skolem 1920).

Idee: Folgende Formeln sind äquivalent:

$$\forall x \exists y \forall z \exists w . P(x, y, z, w)$$
$$\exists f \forall x \forall z \exists w . P(x, f(x), z, w)$$
$$\exists f \exists g \forall x \forall z . P(x, f(x), z, g(x, z))$$

Nun gilt:

$\forall x \exists y \forall z \exists w . P(x, y, z, w)$ erfüllbar $\Leftrightarrow \forall x \forall z . P(x, f(x), z, g(x, z))$ erfüllbar.

Falls $\exists$ nicht unter $\forall$ auftritt, genügt die Einführung von Konstanten:

$\exists x \exists y . P(a, a, x, y)$ erfüllbar $\Leftrightarrow P(a, a, b, c)$ erfüllbar.

5. Stelle konjunktive Normalform her.

Ergebnis: Konjunktion $\chi_0 \wedge \cdots \wedge \chi_k$, wobei χ_i von der Form

$$\forall \vec{x}_1 . B_1 \left(\vec{x}_1\right) \vee \cdots \vee \forall \vec{x}_m . B_m \left(\vec{x}_m\right) \vee \forall \vec{y}_1 . \neg A_1 \left(\vec{y}_1\right) \vee \cdots \vee \forall \vec{y}_n . \neg A_n \left(\vec{y}_n\right)$$

ist mit A_i, B_j atomar. Dies ist (bei Verschiedenheit der Variablen in den Blöcken $\vec{x}_i$, $\vec{y}_j$) äquivalent bzgl. Erfüllbarkeit zu

$$\forall \vec{x}_1 \cdots \vec{\forall} y_1 \cdots \Big(B_1(\vec{x}_1) \vee \cdots \vee B_m(\vec{x}_m) \vee \neg A_1(\vec{y}_1) \vee \cdots \vee \neg A_n(\vec{y}_n)\Big)$$

und gemäß Konvention können die Allquantoren bei Klausen weggelassen werden. Entsteht auf diese Weise aus χ_i die Formel Ψ_i, so leistet

$$\Phi := \Psi_0, \ldots, \Psi_k$$

das Gewünschte.

3.2 Unifikation und Resolution

Formale Regeln des Schließens sind ein uraltes Thema der Logik, und so gibt es auch für die Sprache erster Stufe eine Anzahl verschiedenartiger Kalküle, die korrekt und vollständig sind. Das Resolutionsprinzip (formuliert von J.A. Robinson) ist anwendbar auf Klausen und hat besondere Bedeutung dadurch, daß es einen korrekten und vollständigen Kalkül mit einer einzigen Regel begründet und eine vergleichsweise effiziente Implementierung automatischer Beweisfahren erlaubt. Im folgenden schreiben wir Klausen in der Form

$$B_1 \vee \cdots \vee B_m \vee \neg A_1 \vee \cdots \vee \neg A_n$$

und identifizieren sie jeweils mit der Menge

$$\{B_1, \ldots, B_m, \neg A_1, \ldots, \neg A_n\}.$$

Im *aussagenlogischen Fall ohne Variablen* lautet die Resolutionsregel so: Sind C_1 und C_2 Klausen, so daß A in C_1 und $\neg A$ in C_2 auftritt, so darf man zur Klause $(C_1 - \{A\}) \cup (C_2 - \{\neg A\})$ übergehen.
Die *Korrektheit* dieser Regel ist klar: Aus $A \vee \phi$ und $\neg A \vee \psi$ folgt $\phi \vee \psi$. Auch die *Vollständigkeit* läßt sich nachweisen.
Sonderfälle dieser Regel entsprechen klassischen Gesetzen der Logik:

$$1)\quad \frac{\begin{array}{l}\neg A \vee B\\ \neg B \vee C\end{array}}{\neg A \vee C} \quad \begin{array}{l}\text{entspricht der}\\ \text{Kettenschlußregel:}\end{array} \quad \frac{\begin{array}{l}A \to B\\ B \to C\end{array}}{A \to C}$$

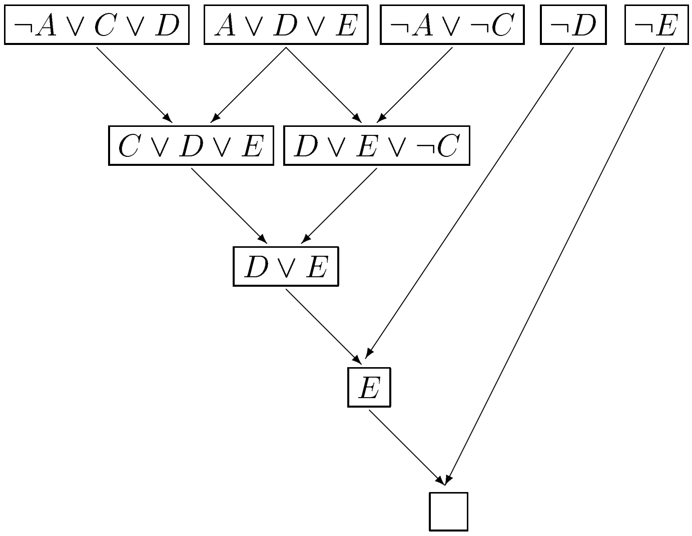
$$2)\quad \frac{\begin{array}{l}A\\ \neg A \vee B\end{array}}{B} \quad \begin{array}{l}\text{entspricht dem}\\ \text{Modus ponens:}\end{array} \quad \frac{\begin{array}{l}A\\ A \to B\end{array}}{B}$$

Beispiel:

$$\text{Aus} \left\{ \begin{array}{ll} 1) & A \to C \vee D \\ 2) & A \vee D \vee E \\ 3) & A \to \neg C \end{array} \right. \quad \text{folgt 4) } D \vee E.$$

Nach dem Resolutionsprinzip wird eine Ableitung von $\square$ aus denjenigen Klausen gesucht, die den Formeln 1) ,2) ,3) und $\neg$4) entsprechen. Abbildung 1 veranschaulicht eine solche Ableitung.
Treten in den Klausen Variablen auf, so setzt die Anwendung der Resolutionsregel i.a. die *Unifikation* voraus. Hierbei werden Variablen durch *Grundterme* ersetzt, d.h. durch Terme, in denen keine Variablen mehr auftreten. Zwei atomare Formeln A_1, A_2 sind *unifizierbar*, wenn es eine Substitution θ gibt, so daß die gemäß θ entstandenen Formeln $A_1\theta$ und $A_2\theta$ übereinstimmen. Man kann zeigen, daß die *Existenz* einer solchen Substitution entscheidbar ist, und daß in diesem Falle zu A_1, A_2 eine allgemeinste Substitution θ_0 (most general unifier *mgu*) effektiv bestimmbar ist. Das entsprechende Verfahren heißt *Unifikationsalgorithmus*. Dabei heißt $\theta_0 mgu$ zu A_1, A_2, wenn $A_1\theta_0 = A_2\theta_0$ und außerdem alle anderen Substitutionen θ, für die $A_1\theta = A_2\theta$ gilt, darstellbar sind als $\theta_0\theta'$ für geeignetes θ'. Die *allgemeine Resolutionsregel* erlaubt nun den Übergang von zwei Klausen C_1 und C_2, so daß A in C_1 und $\neg B$ in C_2 existieren mit *mgu* θ für A und B, zur Klause $(C_1\theta - \{A\theta\}) \cup (C_2\theta - \{\neg B\theta\})$.

Abbildung 1:



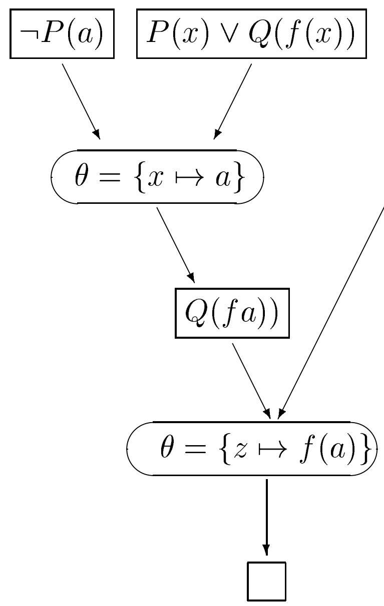
Die *Korrektheit* der allgemeinen Regel ergibt sich wie zuvor, wenn man berücksichtigt, daß die Gültigkeit einer Klause C sicherlich die Gültigkeit von $C\theta$ nach sich zieht. Weniger offensichtlich ist die *Vollständigkeit*. Das Problem, das in der Variablensubstitution steckt, lautet in anschaulicher Formulierung: “Kann man im allgemeinen eine Inkonsistenz, die in einer Menge von Allaussagen steckt, schon anhand von Einzelbeispielen für diese Allaussagen nachweisen?” Eine solche Reduktion ist in der Tat möglich. Dies zeigt der *Satz von Herbrand* (1930), der den Schlüssel zum Vollständigkeitssatz liefert. Eine Version dieses Satzes besagt: Eine Menge S von Klausen ist nicht erfüllbar, genau dann, wenn es eine *endliche* nicht erfüllbare Menge S' von Klausen gibt, die aus S entsteht durch Ersetzen der Variablen durch Grundterme, dies sind Terme, in denen keine Variable mehr auftreten.

3.3 Suchstrategie von `PROLOG`

Die erste, der im vorigen Abschnitt erwähnten Einschränkungen, die in `PROLOG` bzgl. der Prädikatenlogik 1. Stufe zugunsten der Effizienz vorgenommen wurde, ist die ausschließliche Verwendung von Hornklausen. Die Standardsituation betrifft den Vergleich einer Regel $B \leftarrow A_1, \ldots, A_n$ mit Zielen (goals) $G_1, \ldots, G_m$ d.h. man versucht einen Beweis herzuleiten indem man die Klausen

$$(*) \quad B \vee \neg A_1 \vee \cdots \vee \neg A_n \text{ und } \neg G_1 \vee \cdots \vee \neg G_m$$

Abbildung 2: Beispiel einer Ableitung von $\Box$



zum Widerspruch führt, d.h. eine Ableitung von $\Box$ aus $(*)$ sucht. Offensichtlich ist die Anwendung der Resolutionsregel nur über die Unifizierung von B mit einem der G_i möglich. Die Einschränkung auf Klausenpaare der Form $(*)$ bedeutet, daß man auf die Resolution von gegebenen Fakten und Regeln verzichtet. Dieses Verfahren nennt man *lineare Resolution*. In `PROLOG` nimmt man die weitere Einschränkung vor, daß die Resolutionsregel der Reihe nach auf die $G_1, \ldots G_m$ angewendet wird. Man spricht dann von *linearer Input-Resolution*. Selbst die Einschränkung auf lineare Input-Resolution läßt noch die Wahl offen, mit welcher der gegebenen Fakten und Regeln man ein Ziel resolviert. Die *depth-first Strategie* in `PROLOG` beginnt mit der ersten Klause, die eine Resolution zuläßt, und verwendet die weiteren Klausen nur, wenn über ein *backtrack-Verfahren* dieses Ziel erneut bewiesen werden muß. Bei all diesen Einschränkungen wird auf Vollständigkeitseigenschaften zugunsten von mehr Effizienz verzichtet. Eine weitere gravierende Einschränkung in `PROLOG` ist die Verwendung des *cut-Operators*, welcher in einer Klause als Prädikat verwendet werden kann und ein "backtracking" über diese Stelle verhindert. Darüber hinaus gibt es in `PROLOG` weitere *Systemprädikate*, bei deren "Beweis" die aktuelle Klausenmenge durch Hinzufügen oder Streichen einer Klause modifiziert wird. Hierbei handelt es sich um Sprachelemente, die zwar eine einheitliche Notation sowohl für Datenbankmanipulation als auch für logische Anfragen erlauben, die aber nicht mehr konsistent sind mit der Semantik der Prädikatenlogik 1. Stufe. Daher wollen wir hier diese Elemente von `PROLOG` nicht behandeln.

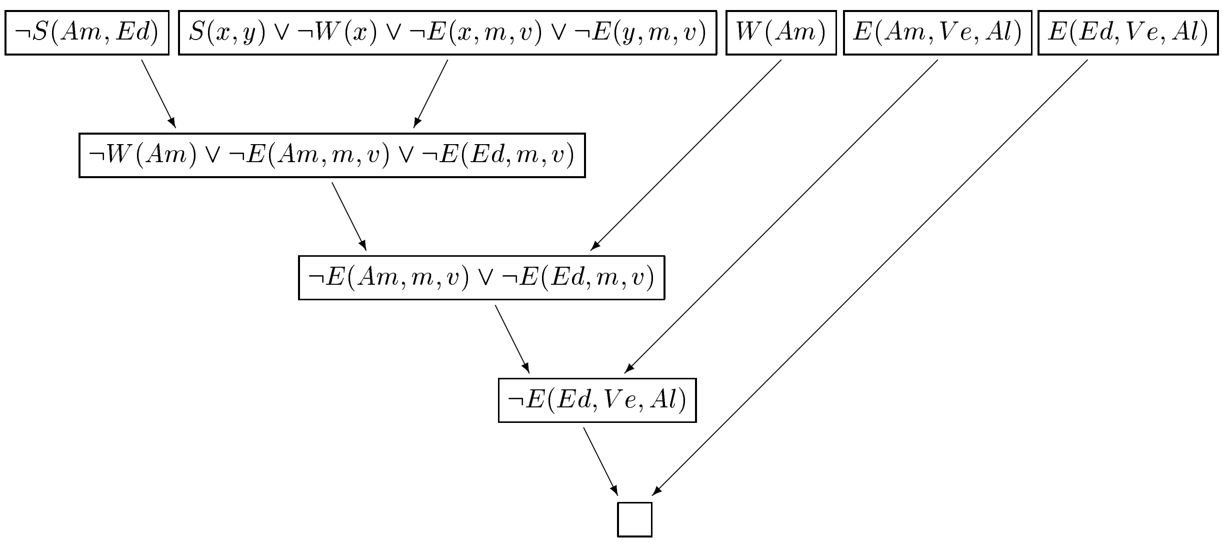
Beispiel: Sei D folgende Klausenmenge aus einer Datenbank mit Familieneinträgen:

$$\{ \quad \begin{array}{l} \textit{Weiblich(Amelie)}, \\ \textit{Eltern(Amelie, Veronika, Albert)}, \\ \textit{Eltern(Eduard, Veronika, Albert)}, \\ \textit{Schwester}(x,y) \leftarrow \textit{Weiblich}(x) \wedge \textit{Eltern}(x,m,v) \wedge \textit{Eltern}(y,m,v) \quad \} \end{array}$$

Eine Anfrage der Art: " Ist Amelie Schwester von Eduard?" lautet in `PROLOG`:

$$?\textit{Schwester(Amelie,Eduard)}.$$

Gemäß der depth-first Strategie beantwortet der `PROLOG`-Interpreter diese Frage positiv nach Erstellen folgenden Beweisbaumes zur Ableitung eines Widerspruches aus D und $\neg$*Schwester(Amelie,Eduard)* (Prädikats-und Konstantennamen wurden abgekürzt):



Zu einer ausführlichen Darstellung des Gebietes der Logik-Programmierung sei auf das Buch von Lloyd verwiesen [Ll87].

4 Grundlagen der Funktionalen Programmierung

Die mathematischen Methoden zur konstruktiven Definition von Funktionen stellen die Basis der funktionalen Programmierung dar. Alonzo Church hat diese Methoden in den 30er Jahren mit der Entwicklung des λ-Kalküls formalisiert. Die Grundidee des λ-Kalküls ist die Abstraktion eines Ausdruckes nach einer Variable, so daß eine einstellige Funktion gewonnen wird.

Beispiele:

- Aus dem Ausdruck $x + 3$ entsteht durch Abstraktion nach x die Funktion $\lambda x.x + 3$, die angewendet auf eine Zahl z, den Wert $z + 3$ ergibt.
- Aus dem Ausdruck 1 entsteht durch Abstraktion nach x die konstante Funktion $\lambda x.1$, die jedes Argument auf den Wert 1 abbildet.

Natürlich lassen sich die Abstraktion und die Applikation beliebig tief verschachteln und man kann beobachten, daß durch wiederholte Abstraktion Funktionen beliebig hoher Ordnung definierbar sind und daß durch sukzessive Applikation diese Ordnung wieder schrittweise reduziert werden kann.

Beispiel: Der Ausdruck $\lambda x.\lambda y.\lambda z.((x + y) \dot{-} z)$ definiert eine Funktion des Typs: $I\!N \rightarrow (I\!N \rightarrow (I\!N \rightarrow I\!N))$. □

Ferner erlaubt derselbe Mechanismus auch die Abstraktion nach Funktionsvariablen.

Beispiel: Der Ausdruck $\lambda f.f(3)$ läßt sich als Funktion vom Typ $(I\!N \rightarrow I\!N) \rightarrow I\!N$ auffassen. Wendet man nun diese Funktion an, etwa auf $\lambda x.0$ oder $\lambda x.x + 1$, so erhält man wieder Ergebnisse im Bereich $I\!N$ der natürlichen Zahlen. □

In den Ausdrücken des reinen λ-Kalküls verzichtet Church sogar auf die Verwendung gegebener Ausgangsfunktionen und setzt ausschließlich eine abzählbare, unendliche Menge V von Variablen voraus und schließt die Menge der Ausdrücke unter Applikation und Abstraktion ab:

Definition 4.0.1 *Die Menge Λ der* λ-Ausdrücke *ist induktiv durch (i) bis (iv) gegeben:*

(i) Jede Variable

$$x \in V$$

ist ein atomarer Ausdruck. *Die Variable x kommt in dem Ausdruck x frei vor. Keine andere Variable kommt in x frei vor.*

(ii) Wenn F und A λ-Ausdrücke sind, so ist auch

$$(FA)$$

ein λ-Ausdruck, der die Applikation *von F auf A bezeichnet. Eine Variable x kommt genau dann in (FA) frei vor, wenn sie in F oder in A frei vorkommt.*

(iii) Wenn A ein λ-Ausdruck und x eine Variable ist, dann ist auch

$$\lambda x.A$$

ein Ausdruck, der die Abstraktion *von A nach x bezeichnet. Eine Variable y kommt genau dann in $\lambda x.A$ frei vor, wenn y in A frei vorkommt und y von x verschieden ist.*

(iv) Die Menge Λ enthält sonst keine Ausdrücke. □

Diese einfache Syntax des λ-Kalküls ist gepaart mit einer zentralen Regel zur Auswertung von λ-Ausdrücken, der *β-Reduktion*, die den oben bereits angedeuteten Mechanismus präzisiert: Wenn eine Funktion, die durch Abstraktion nach x aus einem Ausdruck M gewonnen wurde, auf ein Argument A angewendet wird, so hat diese Applikation den gleichen Wert, wie der Ausdruck M, in dem jedes freie Vorkommen von x durch A ersetzt wurde (vorausgesetzt, daß ein M nach keiner Variablen abstrahiert wird, die in A frei vorkommt), d.h.

$$(\lambda x.MA) \underset{\beta}{\rightarrow} \$^x_A M,$$

wobei der Substitutionsoperator \$ induktiv wie folgt definiert wird:

(i) $\$^x_A\, y = \begin{cases} A, & \text{falls } x = y \\ y, & \text{sonst} \end{cases}$

(ii) $\$^x_A\,(A_1, A_2) = (\$^x_A\, A_1 \$^x_A\, A_2)$

(iii) $\$^x_A(\lambda y.B) = \begin{cases} \lambda y.B, & \text{falls } x = y \\ \lambda y.\$^x_A B, & \text{sonst} \end{cases}$.

Damit die β-Reduktion immer anwendbar ist, wurde zur systematischen Umbenennung gebundener Variablen die *α-Konversion* eingeführt. Falls y nicht in M vorkommt, so gilt:

$$\lambda x.M \underset{\alpha}{\rightarrow} \lambda y.\$^x_y M$$

.

Beispiel: Der Ausdruck $(\lambda x.\lambda y.x\ y)$ läßt sich nicht auf $\lambda y.y$ reduzieren, weil im Argument die Variable y frei vorkommt und im Ausdruck $\lambda y.x$ fälschlicherweise gebunden würde. Daher wenden wir auf den Ausdruck $\lambda x.y$ zunächst eine α-Konversion an, z. B. $\lambda y.x \underset{\alpha}{\rightarrow} \lambda z.x$ und erhalten:

$$(\lambda x.\lambda y.x\ y) \underset{\alpha}{\rightarrow} (\lambda x.\lambda z.x\ y) \underset{\beta}{\rightarrow} \lambda z.y\ .$$ □

Vereinbart man nun, daß man die β-Reduktion und die α-Konversion auch auf beliebige Teilausdrücke anwenden darf, so erhält man den λ-Kalkül. (Wir bezeichnen die reflexive, transitive Hülle von $\underset{\beta}{\rightarrow}$ vereinigt mit $\underset{\alpha}{\rightarrow}$ durch $\underset{\lambda}{\rightarrow}$.) Erstaunlicher Weise ist dieser simple Kalkül bereits genauso mächtig wie die Turingmaschinen, denn bei geeigneter Kodierung der natürlichen Zahlen in die Menge der λ-Ausdrücke ist jede berechenbare Funktion f darstellbar durch einen λ-Ausdruck $f\lambda$, und der Wert von f auf eine Zahl n läßt sich durch die Anwendung von β-Reduktion und α-Konversion "ausrechnen".
Church stellte die natürlichen Zahlen im λ-Kalkül als n-fache Iteration einer Funktion dar:

$$\begin{array}{lcll}
\underline{0} & := & \lambda f.\lambda x.x & \text{steht für } 0 \\
\underline{1} & := & \lambda f.\lambda x.(fx) & \text{steht für } 1 \\
\underline{2} & := & \lambda f.\lambda x.(f(fx)) & \text{steht für } 2 \\
 & & \vdots & \\
\underline{n} & := & \lambda f.\lambda x(f^n x) & \text{steht für } n.
\end{array}$$

Eine Funktion $f : I\!N \to I\!N$ heißt λ-*definierbar*, wenn es einen λ-Ausdruck F gibt, so daß für alle $n \in I\!N$ gilt: $(F\underline{n}) \underset{\lambda}{\rightarrow} \underline{m}$ gdw $f(n) = m$.

Satz: Eine Funktion $f : I\!N \to I\!N$ ist berechenbar, genau dann wenn sie λ definierbar ist. □

Der Beweis bezieht sich auf die μ-rekurvisen Funktionen und macht Gebrauch von der Tatsache, daß sowohl die Ausgangsfunktionen als auch der Rekursionsoperator selbst λ-definierbar sind.

Beispiel: Die Nachfolgefunktion wird durch den λ-Term $N_\lambda := \lambda r.\lambda f.\lambda x.\,(f((rf)x))$ dargestellt, denn es gilt:

$$\begin{aligned}
(N_F\underline{n}) &\underset{\beta}{\rightarrow} \lambda f.\lambda x.\,(f((\underline{n}f)x)) \underset{\beta}{\rightarrow} \lambda f.\lambda x.\,(f(\lambda x.(f^n x)x)) \underset{\beta}{\rightarrow} \\
&\lambda f.\lambda x.\,(f(f^n x)) = \lambda f.\lambda x.(f^{n+1}x) = \underline{n+1} \text{ für alle } n \in I\!N.
\end{aligned}$$ □

Auf der Basis dieses einfachen Kalküls wurden, beginnend mit der Entwicklung von `LISP` [Ca60], die funktionalen Programmiersprachen entwickelt. Wesentliche Hilfen für die Entwicklung von Programmen waren:

– Bereitstellung geeigneter vordefinierter Funktionen,

– Übergang zu rekursiven Gleichungssystemen,

– Einsatz flexibler Typsysteme.

Eine leicht verständliche Einführung in die funktionale Programmierung bietet das Buch von Bird und Wadler [Bi88].
Ein kleines Programmbeispiel zur Berechnung des größten gemeinsamen Teilers in `Miranda` (s. [Tu86]) lautet:

$$\begin{aligned} ggT\,a\,b &= a,\ \text{if } a = b \\ &= ggT\,(a \bmod b)b,\ \text{if } a > b \\ &= ggT\,a\ (b \bmod a),\ \text{otherwise.} \end{aligned}$$ □

Literatur

[Ap94] V. Apt/E.-R. Olderog: Programmverifikation, Springer-Lehrbuch (1994).

[Bi88] R. Bird/P. Wadler: Introduction to Functional Programming, Prentice-Hall (1988).

[Ca60] J. McCarthy et. al.: LISP 1.5 Programmer's Manual, MIT-Press, Cambridge, Massachusetts (1960).

[Fe89] E. Fehr: Semantik von Programmiersprachen, Springer Verlag (1989).

[Ho69] C. A. R. Hoare: An Axiomatic Basis for Computer Programming, CACM 12, 10 (1969).

[Ll87] J. W. Lloyd: Foundations of Logic Programming, Springer Verlag (1987).

[Tu86] D. A. Turner: Miranda: A non-strict functional language with polymorphic types, in Springer LNCS, vol. 201 (1985).