Proseminararbeit

Theorie der logischen Programmierung mit Negation

Richard Möhn

4. Dezember 2012

Davon ausgehend, dass dem Leser logische Programmierung ohne Negation ausreichend bekannt ist, wird das Grundkonzept der logischen Programmierung mit Negation eingeführt: Die Closed World Assumption. Programmvervollständigung als Rechtfertigung gegenüber der klassischen Logik wird betrachtet. Schließlich werden SLDNF-Resolution als Resolutionsverfahren für logische Programme mit Negation und ihre Implementierung in Prolog vorgestellt.

Inhaltsverzeichnis

1 Einführung 2

2 Allgemeine Programme 2

3 Programmvervollständigung 3

4 Stratifikation 5

5 Closed World Assumption und Negation as Failure 6

6 SLDNF-Resolution 8

Literatur 11

This work is licensed under the Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit `http://creativecommons.org/licenses/by/3.0/` or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Tabelle 1: Ein Fahrplan des 19. Jahrhunderts

Edinburgh	Dundee
05:00	07:32
10:00	12:32
13:00	15:32
18:00	20:32

1 Einführung

Ein Fahrplan ist ein Stück Papier, auf dem steht, wann Züge, Straßenbahnen etc. von wo nach wo fahren, und auf dem nicht steht, wann keine Züge oder Straßenbahnen von wo nach wo fahren. Reisende nehmen gewöhnlich an, dass im Fahrplan alle Verbindungen enthalten sind, dass es also keine weiteren Verbindungen gibt, dass also Züge und Straßenbahnen nur dann fahren, wenn es dasteht. Dem Fahrplan in Tabelle 1 würde man so entnehmen, dass um 5 Uhr, 10 Uhr, 13 Uhr und 18 Uhr Züge von Edinburgh nach Dundee fahren, aber nicht um 16:30 Uhr oder 3:46 Uhr.

Diese Denkweise nennt man in der logischen Programmierung *Closed World Assumption (CWA)*: Jedes Programm beschreibt eine Welt; es enthält alle Informationen über diese Welt; damit ist alles falsch, was nicht aus dem Programm folgt. Das Programm enthält also positive Informationen (alles Wahre) explizit, negative Informationen (alles Falsche) implizit. Genauso der Fahrplan, den man als logisches Programm auffassen kann.

Fahrpläne können aber auch Regeln enthalten, beispielsweise »An Feiertagen fahren Sonderzüge« oder »Züge verkehren nur, wenn es nicht stürmt«. Ohne diese Regeln wusste man, dass ein Zug nicht fährt, wenn er nicht vermerkt ist. Nun aber kann man das nicht mehr so leicht feststellen. Das Mittel der logischen Programmierung ist dann Resolution. – SLD-Resolution für definite Programme ohne Negation, SLDNF-Resolution sobald Negation ins Spiel kommt.

2 Allgemeine Programme

Die zweite Regel in dem Fahrplanbeispiel lautete »Züge verkehren nur, wenn es nicht stürmt«. Das könnte man als Klausel $\texttt{abfahrt}(X, Y) \leftarrow \neg\texttt{sturm_zwischen}(X, Y)$. schreiben. Wegen der Negation ist das keine definite Klausel. Weil aber Negation oft praktisch und natürlich ist, definiert man:

Definition 1. Seien A Atom und $L_1, \ldots, L_n$ Literale. Eine *allgemeine Klausel* (*general clause*) ist eine Klausel der Form

$$A \leftarrow L_1, \ldots, L_n.$$

Definition 2. Ein *allgemeines Programm* (*general program*) ist eine endliche Menge allgemeiner Klauseln.

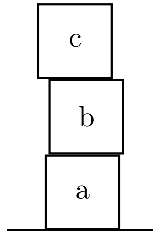

Abbildung 1: Drei aufeinanderstehende Kisten

Beispiel 1. [11, S. 67]

$$\begin{aligned} P\colon \qquad \texttt{fundament}(X) &\leftarrow \texttt{auf}(Y,X), \texttt{auf_boden}(X).\\ \texttt{auf_boden}(X) &\leftarrow \neg\texttt{weiter_oben}(X).\\ \texttt{weiter_oben}(X) &\leftarrow \texttt{auf}(X,Y).\\ \texttt{auf}(c,b)&.\\ \texttt{auf}(b,a)&. \end{aligned}$$

Dieses allgemeine Programm beschreibt die Anordnung der drei Kisten a, b und c in Abbildung 1. Ohne das negative Literal $\neg\texttt{weiter_oben}(X)$ wäre es auch ein definites Programm.

3 Programmvervollständigung

Betrachtet man definite und auch allgemeine Programme, findet man vor den Implikationspfeilen nur positive Literale. Wie soll man daraus folgern, dass etwas nicht gilt, dass also beispielsweise $P \models \neg A$ (P Programm, A Atom)?

Die Closed World Assumption ist das Prinzip, SLDNF-Resolution der Mechanismus dazu und die (modell-)theoretische Rechtfertigung erhalten sie mit der Annahme, dass mit Implikationen eigentlich Äquivalenzen gemeint sind.

Beispiel 2.

$$\begin{aligned} P\colon \qquad \texttt{kann_fallen}(X) &\leftarrow \texttt{auf}(X,Y).\\ \texttt{auf}(c,b)&.\\ \texttt{auf}(b,a)&. \end{aligned}$$

Aus der ersten Zeile folgt nur, dass c und b herunterfallen können; über a wird keine Aussage gemacht. Aus der Äquivalenz $\texttt{kann_fallen}(X) \leftrightarrow \texttt{auf}(X,Y)$ folgte hingegen, dass a nicht herunterfallen kann. Das ist konsistent mit der Closed World Assumption: Falsch ist, was nicht aus einem Programm gefolgert werden kann. – $\texttt{kann_fallen}(a)$ folgt nicht aus P, also gilt $\neg\texttt{kann_fallen}(a)$.

Programmvervollständigung ist das Verfahren, mit dem man in jedem allgemeinen Programm Implikationen in Äquivalenzen verwandeln kann. Keith Clark [4] hat es vorgeschlagen, weshalb es auch *Clark's completion* heißt; die folgende Darstellung orientiert sich an [11, S. 62].

Definition 3. Seien P allgemeines Programm, $k, l, m, n \in \mathbb{N}$ in jedem Schritt neu. Die nach folgendem Verfahren erhaltene Menge von Formeln ist die *Vervollständigung von* P, $comp(P)$:

1. Für jedes Prädikatensymbol $\mathtt{p}$ aus P, ersetze jede Klausel

   $$\mathtt{p}(t_1, \dots, t_m) \leftarrow L_1, \dots, L_n.$$

   aus P durch

   $$\mathtt{p}(X_1, \dots, X_m) \leftarrow \exists Y_1, \dots, Y_k : (X_1 = t_1, \dots, X_m = t_m, L_1, \dots, L_n).,$$

   wobei $Y_1, \dots, Y_k$ alle Variablen aus $L_1, \dots, L_n$ sind und $X_1, \dots, X_n$ in P noch nicht verwendet wurden.

2. Für jedes Prädikatensymbol $\mathtt{p}$ aus P mit $l > 0$, fasse alle Formeln

   $$\begin{gathered} \mathtt{p}(X_1, \dots, X_m) \leftarrow B_1. \\ \vdots \\ \mathtt{p}(X_1, \dots, X_m) \leftarrow B_l. \end{gathered}$$

   zusammen zu

   $$\forall X_1, \dots, X_m : (\mathtt{p}(X_1, \dots, X_m) \leftrightarrow B_1 \vee \dots \vee B_l).$$

3. Für alle nur in Regelrümpfen vorkommenden Prädikate $\mathtt{p}(X_1, \dots, X_m)$ aus P schreibe

   $$\forall X_1, \dots, X_m : \neg \mathtt{p}(X_1, \dots, X_m).$$

4. $=$ ist die Identität. Außerdem gelte (alle Formeln implizit universell quantifiziert):

   $\neg(f(X_1, \dots, X_n) = g(Y_1, \dots, Y_m))$ $\quad f/n$, g/m verschieden
   $f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n) \rightarrow X_1 = Y_1 \wedge \dots \wedge X_n = Y_n$ $\quad f$ beliebig
   $\neg(t = X)$ $\quad t$ Term, in dem X ungebunden vorkommt

Beispiel 3. Formalisierung der Aussagen aus Beispiel 2:

$$\begin{aligned} P &\models \mathtt{kann_fallen}(b) \\ P &\models \mathtt{kann_fallen}(c) \\ P &\not\models \mathtt{kann_fallen}(a) \\ P &\not\models \neg\mathtt{kann_fallen}(a) \\ comp(P) &\models \neg\mathtt{kann_fallen}(a) \end{aligned}$$

Die zwei ersten Aussagen gelten natürlich auch für $comp(P)$.

Beispiel 4. [11, S. 61] Folgendes Programm soll vervollständigt werden.

$$
\begin{aligned}
P: \quad & \texttt{über}(X,Y) \leftarrow \texttt{auf}(X,Y). \\
& \texttt{über}(X,Y) \leftarrow \texttt{auf}(X,Z), \texttt{über}(Z,Y). \\
& \texttt{auf}(c,b). \\
& \texttt{auf}(b,a).
\end{aligned}
$$

Schritt 1 sorgt dafür, dass auch Regeln mit unterschiedlichen Köpfen trotz gleichem Prädikat im nächsten Schritt zusammengefasst werden können.

$$
\begin{aligned}
\texttt{über}(X_1,X_2) &\leftarrow \exists X,Y : (X_1 = X, X_2 = Y, \texttt{auf}(X,Y)). \\
\texttt{über}(X_1,X_2) &\leftarrow \exists X,Y,Z : (X_1 = X, X_2 = Y, \texttt{auf}(X,Z), \texttt{über}(Z,Y)). \\
\texttt{auf}(X_1,X_2) &\leftarrow X_1 = c, X_2 = b, \texttt{auf}(c,b). \\
\texttt{auf}(X_1,X_2) &\leftarrow X_1 = b, X_2 = a, \texttt{auf}(b,a).
\end{aligned}
$$

In Schritt 2 werden Regeln mit gleichem Kopf zusammengefasst und die Implikationen durch Äquivalenzen ersetzt:

$$
\begin{aligned}
\forall X_1,X_2 : \texttt{über}(X_1,X_2) &\leftrightarrow (\exists X,Y : (X_1 = X, \dots)) \vee (\exists X,Y,Z : (X_1 = X, \dots)) \\
\forall X_1,X_2 : \texttt{auf}(X_1,X_2) &\leftrightarrow (X_1 = c, X_2 = b, \texttt{auf}(c,b)) \vee (X_1 = b, X_2 = a, \texttt{auf}(b,a))
\end{aligned}
$$

Schritt 3 ist ein direkter Bezug zur Closed World Assumption: Wenn ein Prädikat in keinem Regelkopf steht, kann es nicht aus dem Programm gefolgert werden. Es wird also explizit als falsch markiert. Der Schritt entfällt hier, weil alle Prädikatensymbole in Regelköpfen auftreten.

Ergänzt man die gewonnenen Formeln noch durch die Übersetzung von Schritt 4 in Formeln, erhält man $comp(P)$.

4 Stratifikation

Nach dem vorhergehenden Abschnitt kann man für jedes Programm P $comp(P)$ konstruieren. Befindet sich in P allerdings die für $p = \mathsf{false}$ wahre Aussage $\texttt{p} \leftarrow \neg\texttt{p}.$, enthält $comp(P)$ die falsche Aussage $\texttt{p} \leftrightarrow \neg\texttt{p}$. Das bedeutet, die Vervollständigung eines konsistenten Programmes kann inkonsistent sein.

Man benötigt aber die Vervollständigung, um aus einem Programm formal korrekt negative Informationen folgern zu können. Dazu muss dessen Vervollständigung konsistent sein. Ob sie es ist oder nicht, ist allerdings nicht entscheidbar. Krzysztof Apt, Howard Blair und Adrian Walker [2] führten deshalb den Begriff der *Stratifizierbarkeit* ein (lateinisch *stratum* heißt so viel wie *Decke* [13], frei übersetzt *Schicht*). Allen Van Gelder [6] kam unabhängig zu den gleichen Ergebnissen.

Definition 4. Ein Programm P ist *stratifizierbar* (*stratifiable*), falls eine Partition $P = P_1 \dot\cup \dots \dot\cup P_n$ existiert, sodass für alle $k = 1, \dots, n$ gilt:

- Alle in P_k positiv vorkommenden Prädikate sind in $\bigcup_{i \leqq k} P_i$ definiert.
- Alle in P_k negativ vorkommenden Prädikate sind in $\bigcup_{i \lneqq k} P_i$ definiert.

Bemerkung. Ein Prädikat ist definiert, wenn es im Kopf einer allgemeinen Klausel vorkommt.

Anschaulich ist ein Programm dann stratifizierbar, wenn man seine Regeln schichtweise aufschreiben kann. Kommt ein Prädikat positiv in einer Schicht vor, muss es dort oder darunter definiert sein. Kommt ein Prädikat negativ in einer Schicht vor, muss es in einer darunterliegenden Schicht definiert sein.

Beispiel 5. [11, S. 69] Das Programm aus Beispiel 1 (S. 3) lässt sich wie folgt stratifizieren.

P_2:	`fundament`$(X) \leftarrow$ `auf`(Y,X), `auf_boden`(X). `auf_boden`$(X) \leftarrow \neg$`weiter_oben`(X).
P_1:	`weiter_oben`$(X) \leftarrow$ `auf`(X,Y). `auf`(c,b). `auf`(b,a).

Beispiel 6. [2]

$$\texttt{r} \leftarrow \neg\texttt{p}.$$

$$\texttt{p} \leftarrow \texttt{r}.$$

Dieses Programm (Argumente weggelassen) lässt sich nicht stratifizieren: Wegen ¬`p` muss die erste Zeile eine Schicht über der zweiten liegen. Das geht aber nicht, weil dann `r` in der Schicht unter seiner Definition verwendet würde.

Die Vervollständigung stratifizierbarer Programme ist immer konsistent. Die Vervollständigung eines nicht stratifizierbaren Programmes muss aber nicht inkonsistent sein. Dafür kann man effektiv algorithmisch feststellen, ob ein Programm stratifizierbar ist *oder nicht.* Außerdem haben stratifizierbare Programme für die Modelltheorie günstige Eigenschaften [2].

5 Closed World Assumption und Negation as (Finite) Failure

In der klassischen Logik gilt: Ein Programm, also eine Menge von Formeln, enthält explizit alles Wissen über die beschriebene Welt. Es gibt kein implizites Wissen. Über alles was nicht dasteht, wird keine Aussage gemacht. Demnach ist der Wahrheitswert einer Aussage A bezogen auf ein Programm P undefiniert, wenn weder A noch $\neg A$ aus P folgt. Dies ist die *Open World Assumption.*

In der logischen Programmierung verwendet man dagegen die bereits erwähnte Closed World Assumption: Eine Aussage A ist bezogen auf ein Programm P genau dann falsch, wenn sie nicht aus diesem folgt. Oder formal:

$$P \nvDash A \leftrightarrow P \vdash_{\text{CWA}} \neg A \leftrightarrow \mathit{comp}(P) \vDash \neg A$$

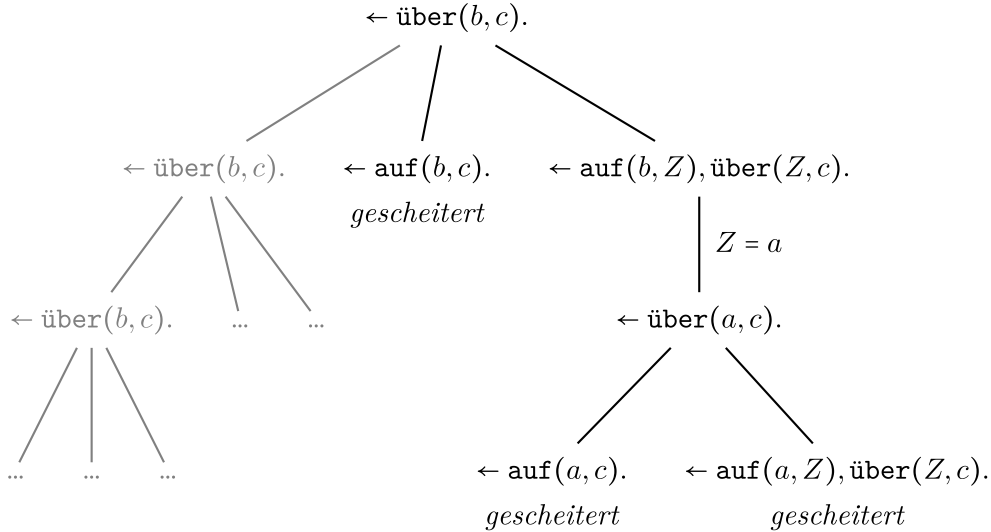

Abbildung 2: SLD-Baum für die Abfrage ← `über`(b, c). bezüglich des Programms aus Beispiel 4 (S. 5). [11, S. 60]

Beispiel 7. Um Aussagen aus definiten Programmen zu folgern, benutzt man meist SLD-Resolution. Der schwarz gedruckte Teil der Abbildung 2 zeigt den SLD-Baum der Abfrage ← `über`(b, c). bezüglich des Programms P aus Beispiel 4 (S. 5). Man sieht, dass alle Zweige gescheitert sind. $A \equiv$ `über`(b, c) konnte also nicht aus P gefolgert werden. Damit ist es nach CWA falsch, $P \vdash_{\text{CWA}} \neg A$.

Ergänzt man das Programm um die Regel `über`(X, Y) ← `über`(X, Y)., erhält man zusätzlich den grauen Zweig, der offensichtlich unendlich ist. Es gilt immer noch $P \nvDash A$ und daher $P \vdash_{\text{CWA}} \neg A$.

Der zweite Fall des Beispiels ist problematisch: Wie soll man algorithmisch herausfinden, dass eine Aussage nicht aus einem Programm folgt, wenn die entsprechende Ableitung unendlich ist? Ob eine Aussage nach Closed World Assumption falsch ist, ist also nicht entscheidbar. Man benötigt eine schwächere Regel, die der *Negation as (Finite) Failure (NAF)*: Bezogen auf ein Programm P ist eine Aussage A falsch, wenn der Versuch, A aus P zu folgern, nach endlich vielen Schritten scheitert. Formal bleibt gegenüber der Closed World Assumption übrig:

$$P \vdash_{\text{NAF}} \neg A \rightarrow \mathit{comp}(P) \vDash \neg A$$

Die Verwandtschaft kann man auch im Wortlaut herstellen. – CWA parallel zur NAF-Regel formuliert: Bezogen auf ein Programm P ist eine Aussage A genau dann falsch, wenn der Versuch, A aus P zu folgern, scheitert.

Beispiel 8. Im schwarz gedruckten Teil des SLD-Baums aus Abbildung 2 sind alle Zweige nach endlich viele Schritten gescheitert. Also ist A auch nach NAF-Regel falsch, $P \vdash_{\text{NAF}} \neg A$.

Nimmt man aber wieder $\texttt{über}(X,Y) \leftarrow \texttt{über}(X,Y)$. dazu, sodass sich der unendliche Zweig ergibt, folgt nach der NAF-Regel nicht mehr, dass A falsch ist, $P \nvdash_{\text{NAF}} \neg A$.

6 SLDNF-Resolution

Ergänzt man die SLD-Resolution um die Negation as (Finite) Failure-Regel, erhält man die *SLDNF-Resolution* (etwa *l*inear *r*esolution for *d*efinite programs with *n*egation as (finite) *f*ailure rule and *s*election rule). Für allgemeine Programme sind weitere Anpassungen notwendig, den Namen behält man bei.

Zunächst braucht man SLDNF-Wälder, die, anschaulich, aus SLD-Bäumen zusammengesetzt sind (die gesamte Darstellung von SLDNF-Resolution folgt [3]):

Definition 5. Seien P allgemeines Programm, Q_0 beliebige Abfrage, $\mathcal{F} = (V, E)$ Wald, dessen Knoten Abfragen sind, und *subs* Funktion, $\textit{subs}\colon V \to V$. Knoten können als *gescheitert*, *erfolgreich* oder *verhaspelt*, Literale als *gewählt* markiert sein.

1. $\mathcal{F} = (\{Q_0\}, \varnothing)$ ist *Prä-SLDNF-Wald* und Q_0 dessen Wurzel.
2. Jede Erweiterung $\mathcal{F}' = E(\mathcal{F})$ eines Prä-SLDNF-Waldes $\mathcal{F}$ ist Prä-SLDNF-Wald.

SLDNF-Resolution ist dann eine Folge von Erweiterungen eines Prä-SLDNF-Baumes.

Definition 6. Seien $\mathcal{F}$ Prä-SLDNF-Wald und P Programm. Berechne die *Erweiterung* $E(\mathcal{F})$ bezogen auf P folgendermaßen.

Für jede Abfrage Q, die ein nichtmarkiertes Blatt in $\mathcal{F}$ ist: Markiere ein nichtmarkiertes Literal L aus Q als *gewählt*.

- L ist positiv.
  - Für jede Klausel C aus P, die auf L anwendbar ist (im Sinne von Unifizierung): Wähle einen Resolventen (α, Q') von Q in Bezug auf L und C und hänge ihn als Kind an Q an. C darf keine Variablen aus der Wurzel Q_0 von $\mathcal{F}$ oder aus Klauseln C früherer Erweiterungsschritte des aktuellen Astes enthalten.
  - Falls es keine solche Klausel gibt: Markiere Q als *gescheitert* (failed).
- $L = \neg A$ ist negativ.
  - A enthält Variablen: Markiere Q als *verhaspelt* (floundered).
  - A ist variablenfrei:
    - $\textit{subs}(Q)$ ist undefiniert: Füge neuen Baum T mit Wurzel $\leftarrow A$. zu $\mathcal{F}$ hinzu und setze $\textit{subs}(Q) = T$.
    - $\textit{subs}(Q)$ ist definiert und erfolgreich: Markiere Q als *gescheitert*.

- $subs(Q)$ ist definiert und nach endlich vielen Schritten gescheitert: Hänge $(\epsilon, Q \setminus L)$ als Kind an Q an.
- $subs(Q)$ ist definiert, aber weder gescheitert noch erfolgreich: Mache mit anderem Blatt weiter.

Markiere leere Abfragen als *erfolgreich*.

Bemerkung 1. Anschaulich ist ein Prä-SLDNF-Wald $\mathcal{F}$ SLDNF-Wald, wenn seine Erweiterung $\mathcal{F}'$ gleich $\mathcal{F}$ ist.

Bemerkung 2. Man findet heraus, ob eine Aussage L bezüglich eines Programms P wahr ist, indem man Erweiterungen eines Prä-SLDNF-Waldes mit der Wurzel $Q_0 = \leftarrow L.$ bezüglich P berechnet. Wenn dessen Hauptbaum einen als *erfolgreich* markierten Zweig enthält, ist L wahr, $comp(P) \vDash L$. Wenn alle Zweige als *gescheitert* markiert sind, ist L falsch, $comp(P) \vDash \bar{L}$.

Für eine formalere Darstellung und weitere Eigenschaften von SLDNF-Resolutionen siehe [3] oder [5].

Beispiel 9. Abbildung 3 zeigt einen SLDNF-Wald für die Abfrage $\leftarrow \texttt{fundament}(X).$ bezüglich des Programms aus Beispiel 1 (S. 3).

Er entsteht wie folgt. Zuerst wird $Q_0 = \leftarrow \texttt{fundament}(X).$ als Wurzel gesetzt. Dann wird resolviert wie bei SLD-Resolution bis zu den negativen Abfragen mit den Literalen $\neg A_1 \equiv \neg\texttt{weiter_oben}(b)$ und $\neg A_2 \equiv \neg\texttt{weiter_oben}(a)$. Weil für diese *subs* noch nicht definiert ist, werden neue Bäume mit A_1 und A_2 als Wurzeln hinzugefügt. *subs* verweist auf diese. Die Resolution des A_1-Baumes ist erfolgreich; der Zweig des Hauptbaumes mit $\neg A_1$ wird als *gescheitert* markiert. Die Resolution des A_2-Baumes ist gescheitert; der Zweig des Hauptbaumes mit $\neg A_2$ wird als *erfolgreich* markiert. Also gibt es ein Fundament, namentlich a, $comp(P) \vDash \texttt{fundament}(a)$.

Beispiel 10. [11, S. 73]

$$
\begin{aligned}
P\colon \quad \texttt{obenauf}(X) &\leftarrow \neg\texttt{blockiert}(X).\\
\texttt{blockiert}(X) &\leftarrow \texttt{auf}(Y, X).\\
\texttt{auf}(b, a).&
\end{aligned}
$$

Laut Definition 6 (S. 8) dürfen Variablen enthaltende negative Literale nicht ausgewählt werden und die entsprechende Abfrage wird als *verhaspelt* markiert. Abbildung 4 (S. 11) zeigt einen SLDNF-Wald für die Abfrage $\leftarrow \texttt{obenauf}(X).$ bezüglich des Programms oben. Das Verbot missachtend wurde ein negatives Literal mit Variablen gewählt; man erhält $P \vdash_{\text{SLDNF}} \neg\texttt{obenauf}(X)$ – keine Kiste ist obenauf. Das ist offensichtlich falsch.

SLDNF-Resolution ohne das Verbot, Variablen enthaltende negative Literale zu wählen, ist also inkorrekt. Leider benutzt Standard-Prolog eine sehr primitive Implementierung von Negation:

$$
\begin{aligned}
&\texttt{not}(X) \leftarrow \texttt{call}(X), !, \texttt{fail}.\\
&\texttt{not}(X).
\end{aligned}
$$

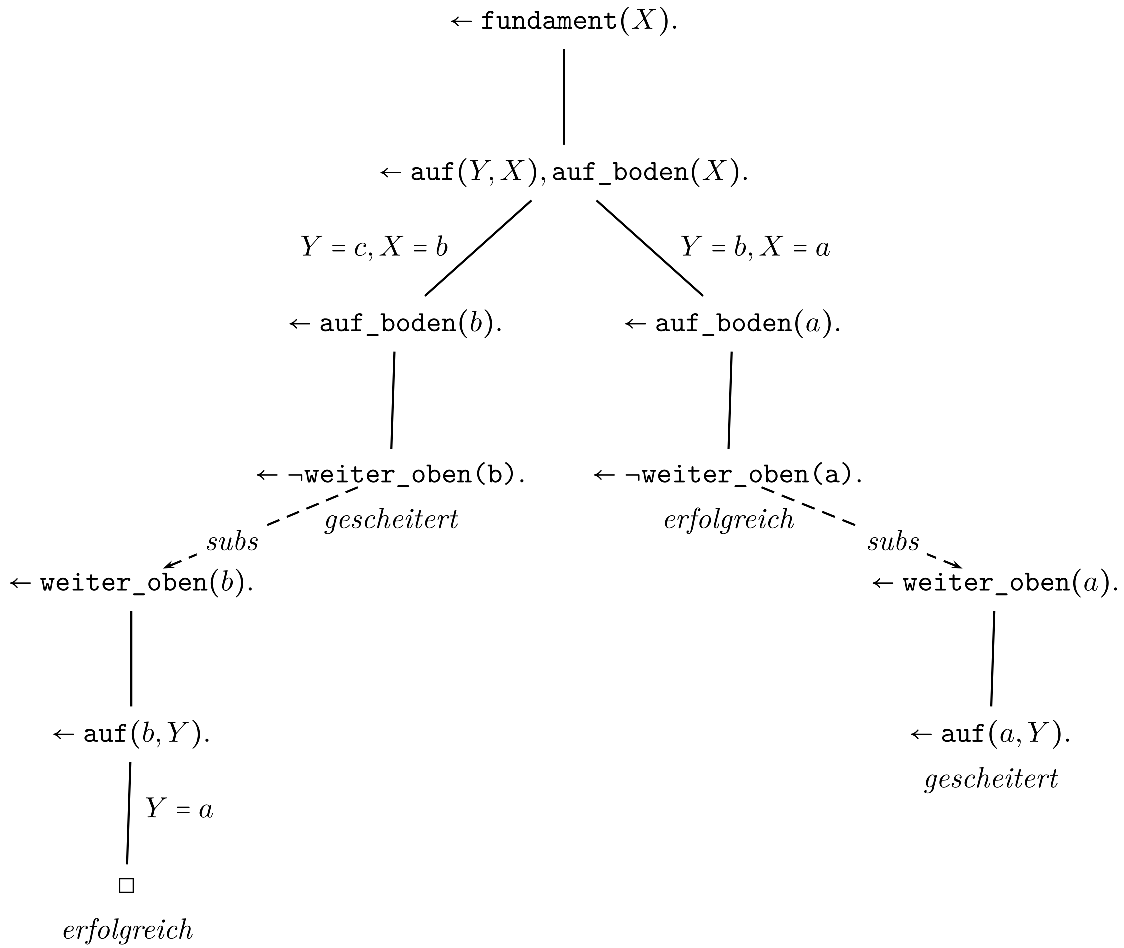

Abbildung 3: SLD-Wald für die Abfrage ← `fundament`(X). bezüglich des Programms aus Beispiel 1 (S. 3) [11, S. 72]. *gewählt*-Markierungen ausgelassen. Substitutionen zur besseren Übersicht nicht streng aufgeschrieben.

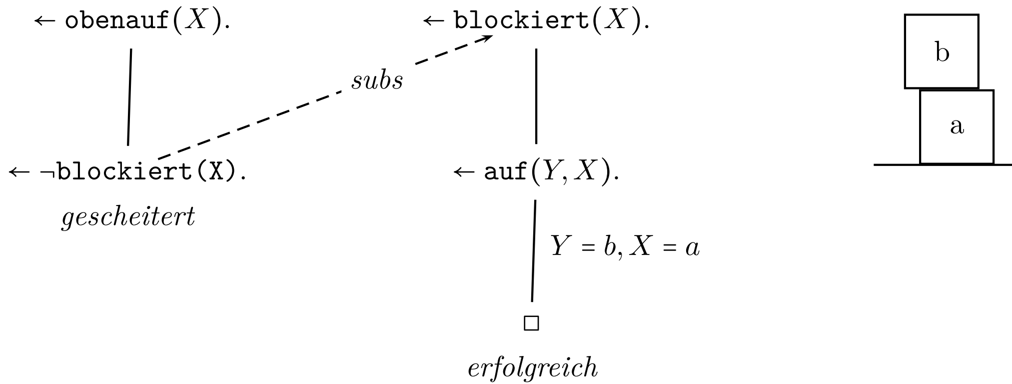

Abbildung 4: SLDNF-Wald für die Abfrage ← `obenauf`(X). bezüglich des Programms aus Beispiel 10 (S. 9), bei dem trotz Verbot ein nicht variablenfreies negatives Literal gewählt wurde [11, S. 73].

In Prolog heißt das vordefinierte Prädikat `\+` statt `not`, aber es arbeitet genau so: X wird aufgerufen; wenn es erfolgreich ist, wird der Rest der Klausel abgeschnitten und der betroffene Zweig als *gescheitert* angesehen. Dann wird die zweite Regel probiert und es gilt `not`(X).

Wenn Prolog $\neg X$ bzw. `\+X` wählt, prüft es also nicht, ob dieses Variablen enthält und liefert unter Umständen falsche Ergebnisse. Programmierer müssen das beachten und möglicherweise elegante logische Programme zum Schlechten verändern, um die Unzulänglichkeiten von Prolog auszugleichen. Aus diesem Grund wurden auf Prolog beruhende Programmierumgebungen wie NU-Prolog [10] entwickelt, die unter anderem Negation korrekt implementieren, ohne sich dabei Theorembeweisern ungebührlich anzunähern. Das Ideal heißt schließlich deklaratives Programmieren und nicht Prädikatenlogik.

Literatur

[1] Krzysztof R. Apt. »Logic Programming«. In: *Handbook of Theoretical Computer Science*. Bd. B: *Formal Models and Semantics*. Hrsg. von Jan van Leeuwen. 2 Bde. Elsevier Science Publishers, 1990, S. 493–574.

[2] Krzysztof R. Apt, Howard A. Blair und Adrian Walker. »Towards a Theory of Declarative Knowledge«. In: *Foundations of Deductive Databases and Logic Programming*. Hrsg. von Jack Minker. Los Altos, CA: Morgan Kaufman Publishers, 1988, S. 89–148.

[3] Krzysztof R. Apt und Roland N. Bol. »Logic Programming and Negation: a Survey«. In: *Journal of Logic Programming* 19, 20 (1994), S. 9–71.

[4] Keith L. Clark. »Negation as Failure«. In: *Readings in Nonmonotonic Reasoning*. Hrsg. von Matthew L. Ginsberg. Morgan Kaufman Publishers, 1978, S. 311–326.

[5] Kees Doets. *From Logic to Logic Programming*. Foundations of Computing. Cambridge, MA und London, England: The MIT Press, 1994.

[6] Allen Van Gelder. »Negation as Failure Using Tight Derivations for General Logic Programs«. In: *Foundations of Deductive Databases and Logic Programming*. Hrsg. von Jack Minker. Los Altos, CA: Morgan Kaufman Publishers, 1988, S. 149–176.

[7] Matthew L. Ginsberg, Hrsg. *Readings in Nonmonotonic Reasoning*. Morgan Kaufman Publishers, 1987.

[8] John Wylie Lloyd. *Foundations of Logic Programming*. 2. Aufl. Symbolic Computation. Artificial Intelligence. Berlin u. a.: Springer-Verlag, 1987.

[9] Jack Minker, Hrsg. *Foundations of Deductive Databases and Logic Programming*. Los Altos, CA: Morgan Kaufman Publishers, 1988.

[10] Lee Naish. »Negation and Quantifiers in NU-Prolog«. In: *Third International Conference on Logic Programming*. Hrsg. von Ehud Shapiro. Bd. 225. Lecture Notes in Computer Science. Springer Berlin und Heidelberg, 1986, S. 624–634.

[11] Ulf Nilsson und Jan Małuszyński. *Logic, Programming and Prolog (2ED)*. 2. Aufl. 2000. URL: `http://www.ida.liu.se/~ulfni/lpp/bok/bok.pdf`.

[12] Raymond Reiter. »On Closed World Data Bases«. In: *Readings in Nonmonotonic Reasoning*. Hrsg. von Matthew L. Ginsberg. Morgan Kaufman Publishers, 1978, S. 300–310.

[13] Josef M. Stowasser, Michael Petschenig und Franz Skutsch. *Der kleine Stowasser*. Hrsg. von Hubert Reitterer und Wilfried Winkler. Bearb. von Robert Pichl. 3. unveränd. Aufl. München: Oldenbourg, 1991.

Kolophon

Diese Arbeit wurde mit XƎLaTeX erstellt; die Dokumentenklasse ist scrartcl aus dem KOMA-Script-Paket von Markus Kohm und anderen.

Für das Literaturverzeichnis wurde biblatex von Philipp Lehman verwendet, zusammen mit biber von Philip Kime und François Charette für die Sortierung. Die Graphiken wurden erstellt mit Hilfe von PSTricks von Timothy van Zandt (und vielen anderen) und pst-tree, ebenfalls von Timothy van Zandt und Herbert Voß.

Weiterhin wurden die Pakete amsmath, amssymb, amsthm, booktabs, csquotes, fontspec, mathabx, MnSymbol, polyglossia und xltxtra benutzt.

Alle Pakete und Programme kamen mit der Distribution TeX Live 2012, betreut von Karl Berry.