Prolog Tutorial

Norbert E. Fuchs

Institut für Informatik

Universität Zürich

Inhalt

Vom deklarativen Wissen zum prozeduralen Programm

Logik verwendet faktische Aussagen, z.B.

Zürich hat einen Bahnhof.

Bern hat einen Bahnhof.

und Implikationen, wie

Wenn zwei Städte X und Y einen Bahnhof haben, dann sind X und Y miteinander verbunden.

In Prolog werden diese Aussagen formal als *Klauseln*

```
bahnhof(zürich).

bahnhof(bern).

verbunden(X, Y) :- bahnhof(X), bahnhof(Y).
```

geschrieben.

Man beachte, dass Konstanten (*zürich*) klein, Variable (*X*) gross geschrieben werden.

Vom deklarativen Wissen zum prozeduralen Programm

Klauseln kommen in zwei Formen: *Fakten* und *Regeln*.

Das Faktum

bahnhof(zürich).

steht für den Satz

"Zürich hat einen Bahnhof."

Die Regel

verbunden(X, Y) :- bahnhof(X), bahnhof(Y).

steht für die Implikation

"Wenn die Stadt X einen Bahnhof hat und wenn die Stadt Y einen Bahnhof hat, dann sind X und Y miteinander verbunden."

Die linke Seite einer Regel wird Kopf, die rechte Seite Körper genannt.

Deklarative Interpretation

Eine Regel wird so interpretiert, dass der Kopf gilt, wenn alle Bedingungen des Körpers wahr sind.

Ein Faktum ist bedingungslos wahr.

Fakten können deshalb formal als Regeln mit dem Körper *true* betrachtet werden.

bahnhof(zürich) :- true.

Prozedurale Interpretation

Prolog-Klauseln haben auch eine prozedurale Lesart, die den Ablauf des Beweises widerspiegelt.

Die Beispiele werden folgendermassen gelesen.

bahnhof(zürich).

Um zu beweisen, dass Zürich einen Bahnhof hat, ist nichts zu beweisen.

und

verbunden(X, Y) :- bahnhof(X), bahnhof(Y).

Um zu beweisen, dass X und Y verbunden sind, beweise, dass X einen Bahnhof hat, und dann, dass Y einen Bahnhof hat.

Vom deklarativen Wissen zum prozeduralen Programm

Die Menge aller Prolog-Klauseln bildet das Prolog-Programm.

bahnhof(zürich).

bahnhof(bern).

verbunden(X, Y) :- bahnhof(X), bahnhof(Y).

Ein Prolog-Programm beschreibt durch seine Fakten und Regeln einen bestimmten Sachverhalt.

Man kann sich über diesen Sachverhalt informieren, indem man Anfragen stellt, wie z.B.

"Sind Zürich und Bern miteinander verbunden?"

formal in Prolog

?- verbunden(zürich, bern).

Der Prolog-Interpreter (oder Prolog-Compiler) beweist nun, dass die Anfrage eine logische Konsequenz des gegebenen Prolog-Programms ist, und antwortet

yes

Vom deklarativen Wissen zum prozeduralen Programm

Noch einmal das Programm.

bahnhof(zürich).

bahnhof(bern).

verbunden(X, Y) :- bahnhof(X), bahnhof(Y).

Auf die Anfrage

?- verbunden(zürich, basel).

antwortet der Interpreter

no

denn aus dem gegebenen Programm lässt sich nicht beweisen, dass Zürich und Basel miteinander verbunden sind.

Vom deklarativen Wissen zum prozeduralen Programm

Noch[2] einmal das Programm.

bahnhof(zürich).

bahnhof(bern).

verbunden(X, Y) :- bahnhof(X), bahnhof(Y).

Die Beweise sind konstruktiv und können deshalb auch für Anfragen nach Unbekanntem verwendet werden.

Auf die Anfrage

?- verbunden(bern, S).

antwortet der Interpreter

S = zürich

Die Variable *S* wird an die Konstante *zürich* gebunden und als Resultat der Berechnung ausgegeben.

Vom Programm zur Berechnung

Um eine Anfrage zu beantworten, sucht der Prolog-Interpreter im gegebenen Programm von *oben nach unten* nach einer Klausel, deren Kopf sich mit der Anfrage syntaktisch zur Deckung bringen lässt.

Handelt es sich um ein Faktum, ist der Beweis gelungen.

Bei einer Regel wird die Anfrage durch die Bedingungen des Klauselkörpers ersetzt, die der Interpreter dann nacheinander von *links nach rechts* auf die gleiche Weise zu beweisen versucht.

Stellt man z.B. die Anfrage

?- verbunden(bern, S).

so findet der Interpreter die Klausel

verbunden(X, Y) :- bahnhof(X), bahnhof(Y)

deren Kopf sich mit der Anfrage deckt, wenn die Variable *X* an die Konstante *bern* gebunden wird und wenn die Variablen *S* und *Y* miteinander identifiziert werden.

Vom Programm zur Berechnung

Nach der Ersetzung durch den Körper der Klausel lautet die neue Anfrage mit zwei Teilanfragen

?- bahnhof(bern), bahnhof(Y).

Der Interpreter versucht, diese beiden Teilanfragen – auch *Ziele* genannt – von links nach rechts zu beweisen. Das erste Ziel wird durch das gleichlautende Fakt

bahnhof(bern)

im Programm bewiesen, und es bleibt noch die Anfrage

?- bahnhof(Y).

Wieder oben im Programm beginnend, findet der Interpreter das Fakt

bahnhof(zürich)

das sich mit dem Ziel durch Bindung von *Y* an *zürich* zur Deckung bringen lässt.

Dadurch wird auch die Variable *S* der ursprünglichen Anfrage an *zürich* gebunden. Nun ist keine weitere Anfrage übrig. Der Beweis ist gelungen und der Wert von *S* wird als Ergebnis ausgegeben.

S = zürich

Logische Variable

Prologs Begriff der Variablen entspricht — im Gegensatz zu prozeduralen Sprachen — in weiten Teilen jenem der Mathematik: Die Variable ist entweder noch unbekannt, oder sie erhält während einer Berechnung einen Wert, den sie für den Rest dieser Berechnung behält.

Es gibt also keine Zuweisung eines Wertes, insbesondere keine mehrfache, sondern nur die (einmalige) Bindung einer Variablen an einen Wert.

Unifikation

Die Bindung von Variablen geschieht durch *Unifikation*.

Diese Operation versucht, zwei Terme syntaktisch zur Übereinstimmung zu bringen.

Dabei werden Variablen, die in den Termen enthalten sind, an ihr Gegenüber gebunden.

Unifikation von

f(a,X,g(X)) *f(a,b,Y)*

durch Bindungen

X an b *Y an g(b)*

geschrieben

{X/b, Y/g(b)}

Backtracking

Prolog kann mehrere Lösungen für eine Anfrage liefern.

Bei der Anfrage

?- bahnhof(Y).

wird das zuerst gefundene Faktum

bahnhof(zürich).

für die Berechnung verwendet.

Nach Ausgabe einer Lösung können weitere Lösungen verlangt werden. Dazu macht der Interpreter die Entscheidung für

bahnhof(zürich)

rückgängig und sucht die nächste Klausel, deren Kopf sich mit dem Ziel zur Deckung bringen lässt.

Dieser Vorgang wird *Backtracking* genannt.

Im Beispielprogramm findet der Interpreter

bahnhof(bern).

und führt die Berechnung mit der Bindung *Y=bern* fort.

Backtracking

Kann keine Alternative gefunden werden, so wird Backtracking auf der vorherigen Stufe versucht, d.h. für den Beweis der ursprünglichen Anfrage

?- verbunden(bern, S).

wird eine alternative Klausel mit dem Kopf *verbunden*/2 gesucht.

Findet der Prolog-Interpreter keine weiteren Alternativen, dann antwortet er

No (more) answers

Wenn eine Anfrage aus mehreren Zielen besteht, z.B.

?- p, q, r, ...

beweist der Prolog-Interpreter die Ziele von links nach rechts. Falls einer der Teilbeweise fehlschlägt, wird automatisch Backtracking zum vorhergehenden Ziel ausgelöst.

Elemente eines Prolog-Programms

Die syntaktischen Elemente von Prolog werden *Terme* genannt. Terme sind

Konstante	*zürich*	*342*	*1.23*	*'Anna Meier'*
Variablen	*X*	*Bahnhof*	*_1234*	
Strukturen	*zug(zürich, Stadt, 112)*			

Eine wichtige Struktur ist die *Liste*.

Liste mit den Elementen a, b, c	[a, b, c]
Kopf (erstes Element)	a
Schwanz (restliche Elemente)	[b, c]

Alternative Darstellungen.

Liste mit den Elementen a, b, c als Liste mit dem Kopf a und dem Schwanz [b, c]	[a | [b, c]]
Liste mit den Elementen a, b, c als Liste mit erstem Element a, zweitem Element b, Schwanz [c]	[a, b | [c]]
leere Liste	[]

Zugverbindungen

Das Zugnetz der Schweiz sei durch die Menge aller direkten Züge zwischen zwei Städten beschrieben.

Die Verbindung einer Ausgangsstadt *A* mit einer Zielstadt *Z* lässt sich durch die folgenden Aussagen beschreiben.

Die Verbindung von A nach A besteht trivialerweise.

Die Verbindung von A nach Z besteht,
wenn es einen direkten Zug von A nach T gibt, und
wenn eine Verbindung von T nach Z besteht.

In Prolog sind die direkten Züge durch Fakten

```
zug(zürich, bern).
zug(bern, lausanne).
...
```

dargestellt. Dann ist die Verbindung zweier Städte gegeben durch

```
verbindung(A, A).
verbindung(A, Z) :-
        zug(A, T),
        verbindung(T, Z).
```

Zugverbindungen

Zwei Erweiterungen

nur Anschlusszüge mit einer Abfahrtszeit später als die Ankunftszeit des vorherigen Zuges

Berechnung der Verbindung

Direkte Züge zwischen zwei Städten *StadtA* und *StadtZ* mit Abfahrtszeit *AbfZ*, Ankunftszeit *AnkZ* und Zugnummer *Num* werden durch Fakten *zug(StadtA, StadtZ, AbfZ, AnkZ, Num)* definiert.

StadtA	**StadtZ**	**AbfZ**	**AnkZ**	**Num**
zug(zürich,	bern,	8.03,	9.15,	712).
zug(bern,	lausanne,	9.18,	10.26,	712).
zug(lausanne,	genève,	10.28,	11.02,	712).
zug(zürich,	olten,	8.06,	8.44,	512).
zug(olten,	biel,	8.47,	9.33,	512).
zug(biel,	neuchâtel,	9.35,	9.54,	512).
zug(neuchâtel,	genève,	9.55,	11.06,	512).
zug(olten,	bern,	8.48,	9.36,	2515).
zug(bern,	lausanne,	10.18,	11.26,	716).
zug(lausanne,	genève,	11.28,	12.02,	716).

Zugverbindungen

Anfrage

"Wie komme ich ab 8.00 Uhr von Zürich nach Genf?"

In Prolog

```
?- verbindung(zürich, genève, 8.00, Wie).
```

Erweiterte Version

```
verbindung(StadtA, StadtA, GAbfZ, []).
verbindung(StadtA, StadtZ, GAbfZ, [StadtT | Wie]) :-
        zug(StadtA, StadtT, AbfZ, AnkZ, Num),
        AbfZ ≥ GAbfZ,
        verbindung(StadtT, StadtZ, AnkZ, Wie).
```

GAbfZ ist die gewünschte Abfahrtszeit. Die Liste *Wie* enthält die Zwischenstationen der gefundenen Verbindung.

```
Wie = [bern, lausanne, genève]
Wie = [bern, lausanne, genève]
Wie = [bern, lausanne, genève]
Wie = [olten, biel, neuchâtel, genève]
Wie = [olten, bern, lausanne, genève]
No more solutions
```

Zugverbindungen

Noch zwei Erweiterungen.

Um zu verhindern, dass man im Kreise herumfährt. merkt man sich die bei der Suche bereits besuchten Städte in einer zusätzlichen Liste *S* und prüft bei jeder neuen Stadt, ob sie nicht bereits besucht wurde, also in dieser Liste verzeichnet ist.

Um zu wissen, welche Züge man nehmen muss, um das Ziel zu erreichen, nimmt man die gewählten Züge ebenfalls in die Verbindungsliste auf.

Neues Programm

```
verbindung(StadtA, StadtA, GAbfZ, S, []).
verbindung(StadtA, StadtZ, GAbfZ, S, [(Num,StadtT)|Wie]):-
        zug(StadtA, StadtT, AbfZ, AnkZ, Num),
        \+ member(StadtT, S),
        AbfZ ≥ GAbfZ,
        verbindung(StadtT, StadtZ, AnkZ, [StadtT|S], Wie).
```

Zugverbindungen

Schnittstellenprozedur

verbirgt Städteliste nach aussen

initialisiert Städteliste (*StadtA* ist die erste besuchte Stadt)

```
verbindung(StadtA, StadtZ, GAbfZ, Wie) :-
        verbindung(StadtA, StadtZ, GAbfZ, [StadtA], Wie).
```

Nun erhält man

```
?- verbindung(zürich, genève, 8.00, Wie).
```

Wie = [(712, bern), (712, lausanne), (712, genève)]

Wie = [(712, bern), (712, lausanne), (716, genève)]

Wie = [(512, olten), (512, biel), (512, neuchâtel), (512, genève)]

Wie = [(512, olten), (2515, bern), (716, lausanne), (716, genève)]

Ideen für weitere Verbesserungen

Meistens gibt es mehrere Möglichkeiten, an den Zielort zu gelangen. Welche ist die beste?

Zwei Möglichkeiten, die Sie ausprobieren können.

1. Angenommen, man versucht zuerst, möglichst lange mit dem gleichen Zug zu fahren, d.h. möglichst wenig umzusteigen. Dazu kann die Hauptprozedur so erweitert werden, dass jene Verbindungen bevorzugt werden, auf denen der gleiche Zug wie der zuletzt gewählte fährt. Ist man also von *StadtA* nach *StadtB* mit Zug *Num* gefahren, so versucht man zuerst, auch von *StadtB* nach *StadtC* mit Zug *Num* zu fahren.
(Hinweis: man spalte die rekursive Klausel des Prädikats *verbindung* in zwei Klauseln auf; eine für den gleichen Zug, eine für einen neuen Zug.)

2. Um die kürzeste Fahrzeit zu finden, könnten z.B. alle Lösungen gesammelt und die kürzeste ausgewählt werden.
(Hinweis: man verwende das Systemprädikat *setof*.)

Entwicklung von Prolog-Programmen

Die entwickelten Programme wurden im sogenannten reinen Prolog geschrieben, in dem Prädikate nur eine logische Bedeutung haben. Für reale Programme wird das volle Prolog benötigt, in dem weitere Prädikate — insbesondere für Seiteneffekte wie die Ein- und Ausgabe — zur Verfügung stehen.

Zwei Aspekte sind typisch für die Programmentwicklung in Prolog. Erstens lassen sich Programme in Prolog abstrakt und deklarativ in den Begriffen des Anwendungsbereichs schreiben. Prolog-Programme sind quasi Spezifikationen des Problems. Das wirkt sich positiv auf die Entwicklung und auf den Unterhalt der Programme aus. Zweitens werden Programme inkrementell entwickelt, d.h. Prolog ist ideal für Prototyping. Die Entwicklung des Programms für Zugverbindungen ist dafür ein gutes Beispiel.

Prolog-Programme sind merklich kürzer als vergleichbare Programme in anderen Programmiersprachen. Auch dafür ist das entwickelte Programm ein gutes Beispiel, da es selbst durch Optimierungen und benutzerfreundliche Ein- und Ausgabe nur wenig länger wird. Bezüglich Effizienz und Komfort der Programmierumgebung haben moderne Prolog-Implementationen mit herkömmlichen Sprachen praktisch gleichgezogen.