Metaprogrammierung und extralogische Prädikate

Leonid Budkov, Benjamin Kuske

6. Juli 2017

1 Vorwort und Motivation

Anfang der 70er Jahre entwickelte Alain Colmerauer die Programmiersprache Prolog. Als deklarative Programmiersprache unterscheidet sie sich grundlegend in Konzepten und Paradigmen, sowie der Auswertung ihrer Programmklauseln von prozeduralen Programmiersprachen. Einsatz fand die Programmiersprache vor allem in den 80er Jahren, zur Entwicklung erster Expertensysteme. Mit der Weiterentwicklung und Verbreitung von prozeduralen und objektorientierten Programmiersprachen rückte die Logikprogrammierung in den Hintergrund. Welche Bedeutung kommt ihr heute noch zu?
In den vergangenen Jahren führte eine Reihe neuer Ansätze zu einer Weiterentwicklung auf dem Gebiet der künstlichen Intelligenz. Maschinelles Lernen könnte sich als eine der Schlüsseltechnologien kommender Jahre auszeichnen. Realisieren lassen sich diese Konzepte mittels logischer Programmierparadigmen. Prolog als gegenwärtig verbreitetste logische Programmiersprache kommt dabei grundlegende Bedeutung zu.

In der folgenden Ausarbeitung werden wir Konzepte der Metaprogrammierung, extralogischer Prädikate und reiner Logikprogrammierung voneinander abgrenzen, um anschließend meta- und extralogische Prädikate zu klassifizieren. Die neu vorgestellten Paradigmen wollen wir anhand eines gegebenen Tic-Tac-Toe Spiels erläutern, indem wir dieses beispielhaft durch anschauliche und einprägsame Prädikate erweitern, die hier betrachtete Konzepte verwenden. Dadurch schaffen wir einen möglichst praktischen Bezug, um die vorgestellten Erweiterungen besser nachvollziehen zu können. Abschließend gehen wir auf die Funktionsweise des fertigen Spiels ein und geben einen Ausblick auf die eingehende Frage, welche Relevanz Prolog in der heutigen Zeit noch hat.

2 Unterscheidung zwischen Metaprogrammierung und extralogischen Prädikaten

Da es keine einheitliche Definition logischer Prädikate gibt, die den Kern logischer Programmiersprache ausmachen, kann man diese nur anhand signifikanter Merkmale festlegen und gegenüber der Metaprogrammierung und extralogischen Prädikaten abgrenzen. Reine logische Programmierung geht vom Programmcode als Wissensbasis aus, in der Fakten und Regeln definiert sind. Dem Programm können Anfragen gestellt werden, worauf der Interpreter automatisiert mögliche Antworten aus dem festgelegten Wissensbestand herleitet. Die automatisierte Herleitung von Antworten mittels Resolution bildet den Hauptbestandteil von Prolog. Rein logische Programme verwenden keine Seiteneffekte zur Realisierung.

Extralogische Prädikate dagegen beschreiben Konzepte, welche zwar auf dem Prinzip von Wissensbasis und Anfrage funktionieren, allerdings mit Hilfe von Seiteneffekten ermöglicht werden. Ein eingängiges Beispiel sind Ein- und Ausgaben, die mittels extralogischer Prädikate möglich sind. Durch die Auswertung eines zu beweisenden Prädikates können textuelle Ausgaben als Seiteneffekt des Prädikates auf dem Bildschirm ausgegeben werden. Anwendung finden extralogische Prädikate vor allem in der Realisierung von Konzepten, die aus prozeduralen Programmiersprachen bekannt sind, wie beispielsweise if-Bedingungen und Schleifen, Ein- und Ausgabe. Da reine Logik-Programme ohne Seiteneffekte arbeiten, gehen extralogische Prädikate über die grundlegenden Konzepte der Logikprogrammierung hinaus und erweitern die Programmiersprache Prolog um einige Vorteile der prozeduralen Programmierung. Da viele der hier beschriebenen Beispiele aus Java bekannt sind, werden wir Java zur Gegenüberstellung heranziehen.

Weitergehend werden Schnittstellen, die zur Kommunikation mit dem zugrundeliegenden Betriebssystem dienen, als extralogische Prädikate aufgefasst. Aufgrund der Systemnähe und der unterschiedlichen gegebenen Voraussetzungen werden diese im folgenden außen vorgelassen, um sich auf allgemeingültige Konzepte zu fokussieren.

Metalogische Prädikate ermöglichen Entwürfe aus der Metaprogrammierung. Diese beschreiben Programme, die auf anderen Programmen bzw. Programmabschnitten arbeiten und diese manipulieren können. Ein Compiler beispielsweise bekommt als Eingabe einen in einer Programmiersprache verfassten Programmcode, verarbeitet diesen syntaktisch und führt gegebenenfalls semantische Optimierungen durch, um eine schnellere Ausführung des ursprünglichen Programms auf der gegebenen Hardware des Rechners zu ermöglichen. Neben der Verarbeitung von fremdem Programmcode, der als Eingabe dem Programm übergeben wird, fasst man unter der Metaprogrammierung auch Programme auf, die den eigenen Quelltext während der Ausführung verändern können. Möglich wird dies bei Prolog, da diese eine interpretierte Programmiersprache ist, die nicht vor der eigentlichen Ausführung vollständig in Maschinencode übersetzt wird. Dadurch können Konzepte realisiert werden, die maschinelles Lernen ermöglichen.

Bei der Implementierung metalogischer Prädikate in Prolog werden oftmals Ansätze extralogischer Prädikate verwendet.

Eine genaue Abgrenzung metalogischer von extralogischen Prädikaten ist oft nicht möglich oder sinnvoll, aufgrund der Wirkung verwendeter Konzepte lässt sich allerdings oftmals eine Zuteilung vornehmen.

3 Anwendung Extralogischer Prädikate

Um ein einfaches Tic-Tac-Toe Spiel zu implementieren, muss das Programm mit dem Benutzer interagieren können. Einfache Anfragen und Antwortsubstitutionen reichen dafür nicht aus. Es gibt eine Reihe vordefinierter extralogischer Prädikate, die Operationen und Formatierungen auf Strings ermöglichen, darunter auch Zeichenfolgen auf der Standardausgabe ausgeben. Diese ist standardmäßig der Bildschirm des Rechners.

```
% Start des Hauptspiels
spiel :-
  nl,
  write('Willkommen zum Tic-Tac-Toe Spiel'), nl,
  auswahlSpielstein.
```

Listing: 1

Das Prädikatensymbol *write*/1 (mit der Stelligkeit 1) bekommt eine Zeichenfolge übergeben, die durch Apostrophe gekennzeichnet wird. Der Prolog-Interpreter handhabt die Zeichenfolge als einen übergebenen Term, der hier als Seiteneffekt ausgegeben wird. Das vordefinierte Prädikat *nl*/0 steht für newline und arbeitet vergleichbar wie das Stringkonvertierungszeichen \n, welches aus Java bekannt ist und einen Zeilenumbruch ermöglicht. Im Anschluss wird das Prädikat *auswahlSpielstein* aufgerufen, wodurch der Spieler festlegen kann, mit welchen Spielsteinen er spielen möchte.

Möchte man in Prolog die Ausgabe von Ergebnissen einer Berechnung oder Auswertung ermöglichen, so ist auch dies durch Variablenumbenennung möglich, indem dem Prädikat *write* eine Variable übergeben wird, die der Interpreter unifizieren kann.

```
% Ausgabe Gesamtanzahl aller gespielten Spiele
anzSpiele(GewonneneSpiele, VerloreneSpiele, Gesamt) :-
      Gesamt is GewonneneSpiele + VerloreneSpiele, write(Gesamt).
```

Listing: 2

Das obige Prädikat berechnet aus der Anzahl der gewonnenen und verlorenen Spiele die gesamte Anzahl gespielter Spiele und gibt diese über *write* auf dem Bildschirm aus.

Möchte man das in *Listing* 1 verwendete Prädikat *auswahlSpielstein* implementieren, benötigt man unweigerlich Eingaben des Benutzers, da diese den fortlaufenden Programmfluss entscheiden können.

```
% Abfrage, mit welchem Spielstein der Spieler beginnen moechte
auswahlSpielstein :-
    nl,
    write('Waehle deinen Spielstein (x oder o)'), nl,
    read(Spieler), nl,
    initSpiel(Spieler).
```

Listing: 3

Das vordefinierte Prädikat *read*/1 liest einen String aus der Standardeingabe. Dieser String wird von Prolog als Term aufgefasst und der Interpreter versucht die Variable *Spieler* mit dem übergebenen Term zu unifizieren. In diesem einfachen Beispiel sollte der Interpreter keine Probleme haben, jede beliebige Eingabe des Benutzers mit der Variable *Spieler* zu unifizieren und an das Prädikat *initSpiel*/1 weiter zu geben. Unerwünschtes Verhalten dürfte erst später, beim Beweisen von *initSpiel* auftreten. Daher ist es sinnvoll die Eingabe des Benutzers im Prädikat *auswahlSpielstein* zu überprüfen, um große Beweisbäume auf falschen Eingaben zu vermeiden, die möglicherweise nicht terminieren.

```
% Abfrage des Spielsteins mit Ueberpruefung der Eingabe
auswahlSpielstein :-
    nl,
    write('Waehle deinen Spielstein (x oder o)'), nl,
    read(Spieler), nl,
    (
        Spieler \= o, Spieler \=x, !,
        write('Fehlerhafte Eingabe!'), nl,
        auswahlSpielstein;
        initSpiel(Spieler)
    ).
```

Listing: 4

Auch in diesem Programm wird wieder versucht die Variable *Spieler* mit übergebenem Term zu unifizieren. Bevor der Prolog-Interpreter allerdings versucht *initSpiel*(*Spieler*) zu beweisen, werden zuerst die vorangestellten Nebenbedingungen versucht zu unifizieren. Das vordefinierte Prädikat \= überprüft zwei Terme auf Ungleichheit und gibt true

zurück, wenn beide Terme nicht identisch sind. Das Prädikat *initSpieler*(*Spieler*) wird also nur versucht zu beweisen, wenn der Nutzer für die Variable Spieler entweder *o* oder *x* eingegeben hat. Hinter den Bedingungen für die Eingabe in Zeile 6 wird ein Cut "!" ausgeführt. Cuts werden insbesondere für die Programmierung von Meta-Prädikaten verwendet. Mittels Cuts können bereits vorhandener Prädikate negiert, oder aussagenlogisch miteinander verknüpfung werden. Im Programm *Listing* 4 wird der Cut dazu verwendet, das Backtracking im Beweisbaum des Prolog-Interpreters und die Suche nach weiteren Beweisen zu verhindern. Wenn der Prolog-Interpreter in Zeile 7 die Variable *Spieler* weder mit *x* noch mit *o* unifizieren kann, wird die weitere Instantiierung mit möglichen Variablenbelegungen für *Spieler* unterbrochen und Zeile 8 und 9 ausgeführt, in denen der Nutzer über eine fehlerhafte Eingabe informiert wird und das Prädikat *auswahlSpielstein* erneut aufgerufen wird.
Aufgrund der Auswertungsstrategie des Prolog-Interpreters benötigt das Programm aus *Listing* 4 weder eine while-Schleife noch eine if-Bedingung, um den Inhalt der Variablen *Spieler* zu überprüfen und bei falscher Eingabe eine erneute Eingabeanfrage zu starten. Da die Strategie des Interpreters vorsieht, alle möglichen Variablenbelegungen von Spieler durchzugehen, diese mit *x* oder *o* zu unifizieren, bis der Beweisbaum durch der Cut unterbrochen wird, kommt das Programm ohne eine Schleife aus.
Dank extralogischer Prädikate ist es allerdings möglich ein Prädikat mit gleicher Funktionalität zu implementieren, wie es aus Java bekannt vorkommt.

```
1  % Listing 4 mit while-loop und if-Bedingung
2  auswahlSpielstein3 :-
3      write('Waehle deinen Spielstein (x oder o)'), nl,
4      repeat,
5      read(Spieler), nl,
6      (
7          if(
8              (Spieler == o; Spieler == x),
9              initSpiel(Spieler) ,
10             fail
11         )
12     ).
13
14 if(A,B,C) :- A,!,B.
15 if(A,B,C) :- write('Falsche Eingabe'),nl, C.
```

Listing: 5

In dem Programm aus *Listing* 5 wird ein selbst geschriebenes if-Prädikat verwendet, welches ähnlich wie eine if-Bedingung in Java funktioniert. Wenn die Bedingung *A* erfüllt ist, wird der Term aus *B* ausgeführt. Der Term *C* wird nur im else Fall ausgeführt. Auch hier wird wieder ein Cut verwendet, um ein Backtracking nach einer gültigen Unifizierung mit *A* zu verhindern.
Da es keine while-Schleife wie in Prozeduralen Programmiersprachen gibt, kann man

ein Prädikat wie in *Listing* 4 rekursiv erneut aufrufen, oder man kann das vordefinierte Prädikat *repeat*/0 verwenden, um den anschließenden Block zu wiederholen. Das Prädikat *repeat* funktioniert wie eine while-Schleife aus prozeduralen Sprachen, die immer *true* bleibt (bspw. *while*(1)).

Nicht nur einfache Ein- und Ausgaben auf dem Bildschirm können logische Programmiersprachen erweitern, auch das Schreiben und Auslesen aus Dateien ist dank extralogischer Prädikate in Prolog möglich. Mittels der vordefinierten Prädikate *see*/1 und *tell*/1 kann die Standardein- und -ausgabe auf eine gewünschte Datei gesetzt werden, um auf dieser Schreibe- und Leseoperationen auszuführen.
In unserem Tic-Tac-Toe Spiel könnte eine sinnvolle Anwendung das Speichern von Spielständen oder Speichern der Spielstatistik sein, sodass die Anzahl aller gewonnenen und verlorenen Spiele, sowie aller unentschiedenen Spiele gezählt werden kann.

```
1  % Praedikat zur permanenten Speicherung der Spielstatistik
2  statistikSpeichern(GewSpiele, UnSpiele, VerSpiele) :-
3      see('Eingabedatei'),
4      tell('Ausgabedatei'),
5      speichern(GewSpiele),
6      speichern(UnSpiele),
7      speichern(VerSpiele),
8      seen,
9      told.
10
11 speichern(AktSpiele) :-
12     read(GespSpiele),
13     anzSpiele(GespSpiele,AktSpiele, GesamtSpiele),
14     write('. ').
```

Listing: 6

In dem Prädikat *statistikSpeichern* in *Listing* 6 werden zwei Variablen mitgeliefert, die lokale Zählungen der gewonnenen und verlorenen Spiele seit der letzten permanenten Speicherung beinhalten. Diese sollen zu der permanent gespeicherten, Spielstatistik einzeln hinzuaddiert werden, um diese aktuell zu halten. In Zeile 3 wird die Standardeingabe auf die Datei des übergebenen Terms gesetzt (Eingabedatei) und in Zeile 4 wird die Standardausgabe auf *Ausgabedatei* gesetzt. Dem Prädikat *speichern* wird die aktuelle Anzahl der Spiele übergeben. Das Prädikat *read* in Zeile 12 liest die Eingabe aus der Eingabedatei aus. Eine Beispiel-Eingabedatei könnte den Inhalt '7. 28. 5.' haben. Das Prädikat *read* in Zeile 12 liest die Datei bis zum '.' aus. In der Beispieldatei würde die 7 ausgelesen werden, die hier für die bereits gezählten und gespeicherten, gewonnenen Spiele steht. Das Prädikat *anzSpiele*/3 ist bereits aus *Listing* 2 bekannt und addiert die ersten beiden übergebenen Terme und speichert das Ergebnis im dritten Term. In *Listing* 6 wird an dieser Stelle die in der Datei gespeicherten Werte mit den aktuellen Werten addiert und das Ergebnis durch die Ausgabe mit write im Prädikat *anzSpiele* in

die Ausgabedatei geschrieben. Das Prädikat in Zeile 6 erfüllt die gleiche Funktionalität mit den aktuellen Werten unentschiedener Spiele mit dem 2. Eintrag, bzw. den verlorenen Spielen mit dem dritten Eintrag der Eingabedatei. Die vordefinierten Prädikate *seen*/0 und *told*/0 schließen die Ein-/ausgabe und setzen diese wieder auf die Standard Ein-/Ausgabe zurück.

Dank extralogischer Prädikate lässt sich eine Vielzahl an Funktionalität aus prozeduralen Sprachen einfach in Prolog übernehmen. Die meisten bisherigen Programmbeispiele und darin enthaltenen Konzepte sind bereits aus Java bekannt und finden durchaus auch Anwendung in Metaprädikaten. Im Folgenden werden Konzepte aus der Metaprogrammierung vorgestellt, die über prozedurale Programmkonstrukte hinausgehen.

4 Anwendungen Metalogischer Prädikate

In der Metaprogrammierung in Prolog unterscheidet man zwischen zwei unterschiedlichen Paradigmen. Es gibt eine Vielzahl vordefinierter Prädikate in Prolog, die einerseits übergebene Terme manipulieren können, andererseits Operationen auf dem momentan ausgeführten Programmcode ausführen können.

4.1 Manipulation von Termen

In Prolog gibt es viele vordefinierte Prädikate, die Terme und atomare Formeln auf bestimmte Eigenschaften prüfen können. Ein Beispiel hierfür ist das Prädikat *number*/1. Dieses bekommt einen Term übergeben und überprüft, ob dieser eine Zahl ist. Sinnvoller Einsatz dieses Prädikates könnte eine Navigation in einem einfachen Menü sein, die durch numerische Benutzereingaben verwaltet wird. Vergleichbar mit *Listing* 4 können durch das Prädikat *number* falsche Eingaben schnell abgefangen werden und so kann ein mögliches Fehlverhalten des Programms verhindert werden. Es gibt in Prolog weitere elementare Prädikate, die übergebene Terme auf bestimmte Merkmale überprüfen können. Das Prädikat *var*/1 gibt *true* zurück, wenn der übergebene Term eine nicht-instantiierte Variable ist. Die Anfrage "$?- V = 42, var(V)$." ergibt $false$, da die Variable V hier instantiiert wurde. Gegenteiliges Verhalten liefert das Prädikat *nonvar*/1, welches *true* zurückgibt, wenn der übergebene Term keine Variable ist.

4.2 Manipulation von Programmen

Betrachten wir zuerst Prädikate, die es ermöglichen Prädikate während der Laufzeit aus der Wissensbasis auszulesen und neue hinzuzufügen. In Prolog bilden die Fakten und Regeln eine Wissensbasis, die auch als Datenbank betrachtet werden kann. Inhalte dieser Datenbank kann man mittels dem vordefinierten Prädikat *clause*/2 auslesen.

```
1 % Programm zur Multiplikation
2 mult(_,0,0).
3 mult(A,B,C) :- A > 0, A1 is A-1, mult(A1, B, C1), C is C1+B.
```

Listing: 7

Eine Anfrage $? - clause(term1, term2)$ lässt sich beweisen, wenn es im Programmcode eine Regel der Form "Konklusion :- Prämisse1, ... , PrämisseN"gibt, die mit *clause* $(konklusion, (praemisse1, ..., praemisseN))$ $unifiziert$ werden kann. Die Anfrage gibt dann *true* zurück.
Das Beispiel würde auf die Anfrage "$? - clause(mult(A, B, C), Preamissen).$" zuerst eine Antwortsubstitution ausgeben, die auf die erste Regel des Prädikates passt $(Y = 0, Z = 0, Praemissen = true)$, bei Anfrage weiterer Antworten mittels ";" erhält man eine Antwortsubstitution, die auf die zweite Regel des entsprechenden Prädikates passt $(Preamissen = A > 0, A1 \; is \; A-1, mult(A1, B, C1), C \; is \; C1+B.)$. Prolog gibt also die Antwortsubstitutionen für $clause/2$ entsprechend der Reihenfolge aus, wie die Regeln zum entsprechenden Prädikat im Quelltext stehen.
Mittels $clause/2$ können also ganze Abschnitte aus dem Quelltext während der Ausführung ausgelesen werden. Dies ist möglich, da Prolog Programme nicht vor der Ausführung in Maschinencode übersetzt, sondern interpretiert werden.
Die Analogie eines Prolog-Programms als Datenbank aus Fakten und Regeln lässt vermuten, dass es in Prolog auch vordefinierte Prädikate gibt, um diese zu erweitern oder Objekte aus dieser zu löschen. Mittels $assert/1$ und $retract/1$ kann man dieses Verhalten ermöglichen.

```
% Anfrage mit Erweiterung der Wissensbasis um Listing2
?- assert(anzSpiele(GewonneneSpiele, VerloreneSpiele, Gesamt) :-
      Gesamt is GewonneneSpiele + VerloreneSpiele, write(Gesamt)).
```

Listing: 8

Angenommen man hat ein einfaches Tic-Tac-Toe Spiel geschrieben und möchte die Funktionalität seines Spiels um das aus *Listing* 2 bekannt Prädikat *anzSpiele* erweitern. Durch *assert* wird der übergebene Term aus der Anfrage heraus im Programmcode erweitert.

```
% Anfrage mit Erweiterung von anzSpiele
?- assert(anzSpiele(0, 0, 0)).
```

Listing: 9

Listing 9 zeigt, dass auch ein bereits vorhandenes Prädikat um eine weitere Regel ergänzt werden kann. Absicht der Anfrage aus *Listing* 9 ist eine schnellere Terminierung zu gewährleisten, für den Fall, dass die Anzahl gewonnener und verlorener Spiele 0 ist. Abgeschlossene Spiele, die unentschieden ausgegangen sind werden hier vernachlässigt. In diesem Fall wäre auch die Anzahl insgesamt gespielter Spiele 0. Prädikate, die eine rekursive Regel haben, könnten somit vor der Ausführung um garantiert terminierende

Fälle ergänzt werden, sodass ein möglicher unendlicher Beweisbaum verhindert wird. Leider reicht das Prädikat *assert* nicht aus, um gewünschtes Programmverhalten hervorzurufen, da *assert* die übergebenen Terme immer am Ende des Programms anfügt.

```
% erweitertes Praedikat im Programmcode (nach Listing 8 & 9)
anzSpiele(GewonneneSpiele, VerloreneSpiele, Gesamt) :-
      Gesamt is GewonneneSpiele + VerloreneSpiele, write(Gesamt).
anzSpiele(0, 0, 0).
```

Listing: 10

Obiger Programmausschnitt zeigt, dass die Programmklauseln nicht in gewünschter Reihenfolge im Quellcode stehen. Da Prolog Klauseln beim Beweisen immer von oben nach unten abarbeitet, würde die 2. Regel für *anzSpiele* niemals verwendet werden. Um dennoch gewünschtes Programmverhalten hervorzurufen, gibt es in Prolog ein vordefiniertes Prädikat *asserta*/1, welches analog zu *assert*/1 arbeitet, mit der Ausnahme übergebene Terme am Anfang des Programms anzufügen.
Um Klauseln aus dem Programm zu löschen, kann *retract*/1 verwenden werden, welches ebenfalls analog zu *assert* angewendet wird.

```
% Anfrage zur Loeschung einer Klausel
?- retract(anzSpiele(0, 0, 0)).
```

Listing: 11

Der Prolog Interpreter versucht das in *retract* übergebene Prädikat mit einer Programmklausel zu unifizieren. Wenn der Beweis gelingt, wird die Klausel aus dem Quellcode gelöscht.

```
% Anfrage zur Loeschung fuer mehrere Klauseln
?- retract(anzSpiele(X, Y, Z):- praemissen).
```

Listing: 12

Wenn *Listing* 12 als Anfrage an unser Tic-Tac-Toe Programm gestellt wird, löscht der Interpreter die erste Klausel, die mit dem übergebenen Term unifiziert werden kann. In unserem erweiterten Programmcode aus *Listing* 10 wäre das die Programmklausel *anzSpiele*/3, die bereits in *Listing* 2 vorgestellt wurde. Durch Eingabe von ";" würde der Interpreter weitere Programmklauseln suchen, die mit dem übergebenen Term von *retract*/1 unifiziert werden können.

4.3 Prolog als Interpreter

Durch das Vordefinierte Prädikat *clause* können Abschnitte, bzw. ganze Programme selbst wieder als Terme verwendet werden. Prolog eignet sich daher gut zum schreiben von Meta-Programmen.

```
1 %Meta-Interpreter um Beweislaenge erweitert
2 beweis(true,0) :- !.
3 beweis((Z1,Ziel2),N) :- !, beweis(Z1,N1), beweis(Z2,N2), N is N1+N2.
4 beweis(Z,N) :- clause(Z, Rest), beweis(Rest,N1), N is N1+1.
```

Listing: 13

Listing 13 zeigt einen möglichen Meta-Interpreter, also einen Interpreter für eine Programmiersprache, der in selbiger verfasst wurde. Zweck eines Meta-Interpreters könnte es sein, den vorhandenen Interpreter um Funktionalität zu erweitern. Obiger Interpreter baut auf der gegebenen Auswertungsstrategie von Prolog auf und zählt die einzelnen Beweisschritte in der Variablen N. Mit Hilfe von Cut verhindert man unerwartetes Verhalten beim Backtracking im Beweisbaum. Zeile 3 verzweigt mehrgliedrige Beweisziele durch rekursive Aufrufe, in Zeile 4 wird explizit nach einer Übereinstimmung der zu unifizierenden Klausel in der Wissensbasis gesucht und N um 1 inkrementiert.

5 Zusammenfassung und Ausblick

Wir sind in den letzten beiden Kapiteln von einem bereits vorhandenen Tic-Tac-Toe Spiel ausgegangen und haben beispielhafte Prädikate gezeigt, wie wir Erweiterungen für dieses Spiel schaffen können, die sich mit Hilfe von extralogischen und metalogischen Prädikaten einfach Implementieren lassen. Man kann somit nicht nur einfacher Interaktionen mit dem Nutzer ermöglichen und Verzweigungen im Programmablauf vornehmen, sondern auch auf den bestehenden Prädikaten und Programm(-abschnitten) arbeiten. Während der Programmausführung kann man die bestehende Wissensbasis um Berechnungsergebnisse erweitern, wodurch das Programm selber Verhalten anlernen kann. Beispielsweise kann man ein metalogisches Prädikat schreiben, welches Abfolgen von verlorenen Spielen in seiner Wissensbasis aufnimmt und auswertet, damit ein vom Computer gesteuerter Spieler künftig bessere Entscheidungen treffen kann.

Literatur

[1] Leon Sterling, *The Art of Prolog: Advanced Programming Techniques*, The MIT Press, Massachusetts, 2nd edition, 1999.

[2] J. Giesl, *Logikprogrammierung*, Skript zur Vorlesung, RWTH Aachen, 2015.