Extralogische Prädikate und Metaprogrammierung

Valentino Geuenich, Florian Tümmers

1 Einleitung

Prolog ist eine logische Programmiersprache, welche, aufgrund einer Wissensdatenbank, die Anfragen eines Benutzers versucht logisch abzuleiten. Doch Prolog kann noch einiges mehr. Es ist in der Lage ganze Sprachen zu erzeugen oder effizient allgemeine Probleme zu lösen. Wir kennen Prolog bisher als Konsole, die auf unsere Fragen sogar zum Teil intelligente Antworten geben kann. Eine wirkliche Interaktion zwischen Benutzer und Programm findet allerdings noch nicht statt. Wir können nur Anfragen stellen, welche uns Prolog mit Variablenbelegungen, *true* oder *false* beantworten kann. Wollen wir tiefer in den Beweisbaum vordringen, gelingt uns dies nur mit viel Verständnis von Prolog oder mit viel Geduld, um Schritt für Schritt den Beweisbaum nachzubauen. Viel einfacher wäre es, der Entwickler eines Programms könnte zusätzliche Aussagen einbauen, um den Benutzer darüber aufzuklären, was gerade passiert, mit ihm interagieren, ihn fragen, was er wissen will, und davon abhängig sein Problem lösen. Folglich wäre der Benutzer dann auch im Stande, wenn der Entwickler dies unterstützt, schon auf der Konsole neue Fakten und Regeln zu beschreiben, bzw. sein Problem zu deklarieren, ohne sich mit der Syntax und Semantik von Prolog auseinanderzusetzen.
Im Zuge dieser Ausarbeitung werden wir anhand eines Spieles die Grenzen des prädikatenlogischen Modells von Prolog kennenlernen. Für dieses Spiel abstrahieren wir ein Tier auf drei prägnante Eigenschaften, die es vollständig beschreiben. Das ist in Prolog kein Problem. Wir definieren mehrere Fakten mit dem Namen *tier*, welche die drei Eigenschaften eines Tieres mit dessen Namen verknüpfen. Auch die Eigenschaften implementieren wir als Fakten mit dem Namen *property* und zur besseren Verwaltung erhält jeder von ihnen eine individuelle Nummer.

```
property(1,hoch).       property(4,gross).
property(2,klein).      property(5,africa).
property(3,tief).       property(6,schuppen).

tier(hoch,klein,tief,'Biber').
tier(gross,africa,schuppen,'Krokodil').
```

Nun können wir Fragen zu unseren Tieren stellen, zum Beispiel, welche Eigenschaften einen *Biber* ausmachen. Diese Tiere sind, in unserem Fall, *klein*, *hoch* und *tief*. Auch die

Eigenschaften können wir vom System erfragen. So wissen wir, dass die Eigenschaft *klein* der Zahl 2 zugeordnet wird, während *africa* die Zahl 5 erhalten hat. Wie ein Spiel fühlt sich das Ganze allerdings noch nicht an. Im Laufe der Ausarbeitung werden wir dieses Beispiel um eine Interaktion mit dem Benutzer erweitern, welche es ermöglicht, ihn nach drei Eigenschaften zu fragen, woraufhin entweder ein Tier gefunden und ausgegeben wird, oder ein weiteres Tier dem Programm hinzugefügt wird.

2 Extralogische Prädikate

2.1 Seiteneffekte

Ein Seiteneffekt tritt auf, wenn zum Beispiel Funktionen globale Variablen verändern. Dies kann man sich zunutze machen. Es wird allerdings davon abgeraten, da Seiteneffekte weder guter Programmierstil sind noch die Lesbarkeit des Codes verbessern. Durch sie wird das Programm unverständlicher und sollte sich ein Seiteneffekt als Problem herausstellen, ist es bei großen Projekten äußerst schwierig diesen ausfindig zu machen, da sie vom Compiler nicht erkannt werden. Dadurch sind Seiteneffekte ein großes Ärgernis und man versucht sie wenn möglich zu vermeiden.

```
public class JavaExample {
    int key = 1234;
    // Als Seiteneffekt wird key auf den uebergebenen Wert x gesetzt.
    public int getSquare(int x) {
        key = x;
        return x * key;
    }
    public int getKey() {
        return key;
    }
}
```

In der oben beschriebenen Klasse haben wir eine globale Variable *key* und die Funktionen *getSquare(int x)* und *getKey()*. Durch das Aufrufen der Funktion *getSquare(int x)* setzen wir *key* auf den Wert von x. Möchten wir nun die Funktion *getKey()* aufrufen, erhalten wir nicht mehr dasselbe Ergebnis wie vor dem Ausführen von *getSquare(int x)*. Obwohl also alle Funktionen für sich selbst funktionieren, liefert eine bestimmte Kombination dieser falsche Ergebnisse. Somit wird es umso schwieriger den Fehler zu finden. Um solche Probleme zu vermeiden, gibt es zwei Möglichkeiten. Entweder man überlässt dem Programmierer die Aufgabe auf Seiteneffekte zu achten oder man schließt sie von vornherein aus.
Letzteres ist bei funktionalen und logischen Programmiersprachen der Fall. Diese haben keine Seiteneffekte. Rein funktionale Sprachen profitieren von ihrer Wirkungsfreiheit, da ein Ausdruck immer von seiner Umgebung abhängt und nicht von einem bestimmten Zeitpunkt oder einer bestimmten Reihenfolge der Auswertung. Code in funktionalen Sprachen wird somit sehr klar und eindeutig. Ein paar Zeilen Code beschreiben die Strategie, anstatt wie imperative Sprachen einen vollständigen Algorithmus zu implementieren. Somit ist es für den Programmierer leichter die Funktion zu verstehen und nachzuvollziehen. Diese

Eigenschaft wird referenzielle Transparenz genannt und wirkt sich auch positiv auf die Lesbarkeit der Sprache im Allgemeinen aus. Ebenso verhält es sich mit logischen Programmiersprachen. Da logische Programme, zum Beispiel in Prolog, nur aus Fakten und Regeln bestehen, die unsere Wirklichkeit beschreiben, und das Programm selbst diese Datenbank nur nach einer Lösung des Problems durchsucht, kommen wir nicht mit Seiteneffekten in Berührung.

2.2 Seiteneffekte in Prolog

Nun allerdings kommen wir zu dem Problem, welches Prolog mit sich bringt. Sein prädikatenlogisches Modell verhindert die Verwendung von Seiteneffekten. Bisher kann Prolog uns nur zurückgeben, ob ein Term *true* oder *false* ist, oder wie Variablen belegt werden sollen. Was aber, wenn wir mit dem Benutzer über die Konsole interagieren wollen? Wenn wir versuchen analog zu Java die Ausgabe in Prolog zu programmieren, so werden wir ohne Seiteneffekte scheitern. Sei *read(X)* ein Prädikat zum Einlesen eines Strings, welcher vom Benutzer eingegeben wurde, und *write(X)* ein Prädikat zum Ausgeben eines Strings. Benutzen wir *write(X)* nun um den Benutzer aufzufordern eine Eingabe zu tätigen. Für den Term *write('Bitte gib etwas ein.')*, wird Prolog uns *true* zurückgeben. Die Konsole möchte damit sagen: "Ja, X ist ein String." Mehr wird auf der Konsole nicht erscheinen. Unsere gewünschte Ausgabe ist also nur dafür verwendet worden, zu prüfen, ob es sich bei dieser um einen String handelt. Genauso verhält es sich mit *read(X)*. Gibt der Benutzer einen String ein, so wird *true* zurückgegeben, gibt der Benutzer keinen String ein, *false*. Das Fehlen von Seiteneffekten führt aber nicht nur bei der Ein- oder Ausgabe auf der Konsole zu Problemen. Es existieren noch viele weitere gewünschte oder auch benötigte Funktionen, die ohne Seiteneffekte ebenfalls nicht realisierbar sind. So ist z.B. das Öffnen einer Datei nur mit extralogischen Prädikaten möglich. Zusätzlich kann man mit extralogischen Prädikaten Prolog wesentlich effizienter gestalten.
Da Seiteneffekte innerhalb des prädikatenlogischen Modells nicht möglich sind, wurden die *extralogischen Prädikate* eingeführt. Diese liegen außerhalb des prädikatenlogischen Modells und ermöglichen uns somit die Nutzung von Seiteneffekten. Diese Prädikate sind vordefiniert. Wir können demnach keine eigenen definieren, ohne selbst auf bereits definierte extralogische Prädikate zurückzugreifen. Sie ermöglichen uns erweiterte Funktionen und ebenfalls Möglichkeiten den Beweisbaum zu modifizieren und ihn nach unseren Wünschen zurechtzuschneiden.
Nun allerdings stellt sich die Frage, wie die Seiteneffekte helfen, etwas auf der Konsole mit *write(X)* auszugeben. Genauso wie zuvor wird immer noch geprüft, ob X ein String ist. Wenn X ein String ist, wird *write(X)* zu *true* ausgewertet. Anschließend wird als Seiteneffekt der String X auf der Konsole ausgegeben.

2.3 Vorstellung verschiedener extralogischer Prädikate

Extralogische Prädikate bieten eine Reihe von Eigenschaften, welche mit Logik allein nur schwer möglich wären. Für das Programm selbst sind es weitere Fakten oder Regeln, die es auswertet. Doch während die Prädikate ausgewertet werden, modifizieren sie das Programm oder kreieren neue Konsolen-Einträge. Im Folgenden stellen wir ein paar der

extralogischen Prädikaten vor. Doch vorher erläutern wir die verwendete Schreibweise:
praed(+Term) bedeutet das Term vollständig instanziiert sein muss.
praed(-Term) bedeutet das Term zur Ausgabe vorgesehen ist.
praed(@Term) bedeutet das Term nicht instanziiert sein muss.

write/1 **write**(+Term) realisiert die Ausgabe, indem es *Term* in der Konsole ausgibt. Das Prädikat selbst wird immer zu *true* ausgewertet.

nl/0 **nl** fügt der aktuellen Ausgabe in der Konsole einen Absatz hinzu und beginnt so eine neue Zeile.

read/1 **read**(-Term) implementiert die Eingabe, indem es den in die Konsole eingegebenen Prolog-Term mit *Term* unifiziert. Schlägt dies fehl, so wertet das Prädikat zu *false* aus.

var/1 **var**(@Term) wertet zu *true* aus, wenn *Term* aus freien Variablen besteht. So werten die Ausdrücke **var**(X), **var**(Giesl) zu *true* aus, während Ausdrücke wie **var**(1) oder **var**(juergen) zu *false* ausgewertet werden.

nonvar/1 **nonvar**(@Term) ist das Pendant zu **var**(@Term). Die Ausdrücke **nonvar**(X), **nonvar**(Giesl) werden zu *false* ausgewertet, während Ausdrücke wie **nonvar**(1) oder **nonvar**(juergen) zu *true* ausgewertet werden.

atomic/1 **atomic**(@Term) ist wahr, falls *Term* ein nullstelliges Prädikats- oder Funktionssymbol oder eine Zahl ist. Beispielsweise führen die Anfragen:
"?- **atomic**(a).", "?- **atomic**(-).", "?- **atomic**(2)." oder "?- **atomic**(-2)." zu *true*
und "?- atomic(X)." oder "?- atomic(a(a))." zu *false*.

compound/1 **compound**(@Term) bildet das Pendant zu **atomic/1**. Daher ist **compound**(@Term) wahr, falls *Term* ein Term oder eine atomare Formel ist, die nicht nur aus einem nullstelligen Funktions- oder Prädikatssymbol oder einer Zahl besteht. So ergeben die Anfragen:
"?- **compound**(a).", "?- **compound**(2)." oder "?- **compound**(-2)." *false*
und "?- **compound**(X)." oder "?- **compound**(a(a))." *true*. [4]

All diese Prädikate sind ein fester Bestandteil der Prolog-Syntax und werden nicht anders als normale Regeln oder Fakten abgearbeitet.

2.4 Beispiele für extralogische Prädikate

Hilfreich sind die Prädikate zum Beispiel für folgendes bekanntes Programm:

```
start :- write('hello world!'), nl.
```

Mit dem Aufruf von *start*. wertet Prolog das Prädikat *write* aus. Als Nebeneffekt wird `hello world!` in der Konsole ausgegeben.
Eine kleine Variation des obigen Beispiels soll, wenn ein Name übergeben wird, diesen grüßen und ansonsten nach einem Namen fragen und diesen grüßen.

```
start(X) :- nonvar(X), write('hello '), write(X), nl, !;
            write('What is your name?'), nl, read(X),
            start(X).
```

Unter Verwendung des Prädikats *nonvar* können wir überprüfen, ob der übergebene Term bereits definiert ist. Wenn dem so ist, grüßt das Programm den Term über eine einfache Ausgabe. Mit Hilfe eines Cuts wird die weitere Auswertung des Prädikates unterbunden, sodass nicht zusätzlich der zweite Teil ausgewertet wird. Der zweite Teil wird aktiv, sollte die Auswertung von *nonvar* scheitern. In diesem Fall fragt das Programm nach unserem Namen, liest diesen über das Prädikat *read* ein und gibt ihn, indem es sich selbst rekursiv aufruft, aus. X ist somit im erneuten Aufruf ein Wert zugewiesen, wodurch die Auswertung des Prädikats *nonvar* nicht scheitert und der Name gegrüßt wird. Sollte der Benutzer keine Belegung eingeben und somit X nicht instanziiert sein, so fragt das Programm erneut nach einer Belegung.

Die Prädikate sind ein mächtiges Mittel in und für Prolog, auch wenn die gezeigten Beispiele nur einen kleinen Einblick in den Gebrauch dieser zeigen. Ihr tatsächlicher Nutzen wird im folgenden Kapitel deutlich.

2.5 Implementierung der extralogischen Prädikate

Implementieren wir nun unsere erlangten Kenntnisse in unser kleines Spiel:

```
game :- bed1(1).
bed1(X) :- translate(X), property(X,_), X1 is X + 1,
           (read(Y), Y = ja, bed2(X,X1), !;
           bed1(X1)), !; true.
bed2(A,X) :- translate(X), property(X,_), X1 is X + 1,
             (read(Y), Y = ja, bed3(A,X,X1), !;
             bed2(A,X1)), !; true.
bed3(A,B,X) :- translate(X), property(X,_), X1 is X + 1,
               (read(Y), Y = ja, searchAnimal(A,B,X), !;
               bed3(A,B,X1)), !; true.
translate(X) :- property(X,Y),
                write('Hat dein Tier die Eigenschaft : '),
                write(Y), write(" ?\n");
                write('Ich kenne keine weitere Eigenschaft.\n').
searchAnimal(A,B,C) :- property(A,A1), property(B,B1), property(C,C1),
                       tier(A1,B1,C1,X), write('Ist dein Tier ein '),
                       write(X), write('?\n'), read(Y), Y = ja,
                       write('Das System macht keine Fehler!'), !;
                       write('Ich kenne dein Tier nicht.\n').
property(1,hoch).          property(4,gross).
property(2,klein).         property(5,africa).
property(3,tief).          property(6,schuppen).
tier(hoch,klein,tief,'Biber').
tier(gross,africa,schuppen,'Krokodil').
%and so on
```

Mit den extralogischen Prädikaten können wir nun die Interaktion mit dem Benutzer realisieren. Wird "?- *game.*" in der Konsole aufgerufen, wird *bed1(X)* für $X = 1$ ausgewertet. Darin wird weiterhin das Prädikat *translate(X)* aufgerufen. Dieses sucht mit *property(X, Y)* den Namen der Eigenschaft des Tieres für den Wert X und unifiziert diesen mit Y. Wenn

X allerdings zu keiner Eigenschaft passt, also zu groß ist, wird zurückgegeben, dass das Programm keine weiteren Eigenschaften kennt und es wird beendet. Sollte dies nicht der Fall sein, so wird daraufhin der Benutzer gefragt, ob dessen ausgesuchtes Tier die Eigenschaft *Y* hat. Wenn ja, wird die Zahl dieser Eigenschaft weiter mitgeführt und *bed2(X,X + 1)* aufgerufen. Wenn nein, wird *bed1(X + 1)* ausgeführt, um die nächste Eigenschaft zu überprüfen. Wenn das Programm nun alle drei Eigenschaften des Tieres ausfindig gemacht hat, wird *searchAnimal(A,B,C)* aufgerufen, wobei *A* die Nummer der ersten Eigenschaft, *B* die Nummer der zweiten Eigenschaft und *C* die Nummer der dritten Eigenschaft ist. Dadurch werden die jeweiligen Eigenschaften herausgesucht und mit *A1*, *B1* und *C1* unifiziert. Mit diesen Eigenschaften wird nun das gewünschte Tier gesucht. *tier(A1,B1,C1,X)* unifiziert *X* mit den Tieren, die jeweils zu *A1*, *B1* und *C1* im Programm registriert sind. Daraufhin wird der Benutzer gefragt, ob eines dieser Tiere seinem entspricht. Wenn dies nicht der Fall ist, wird *"Ich kenne dein Tier nicht."* in der Konsole ausgegeben und das Programm wird terminiert. Falls eines der Tiere das gesuchte ist, wird das Programm terminiert. Unser Spiel ist jetzt schon viel interaktiver, jedoch ist das Erweitern der Eigenschaften, sowie der Tiere recht umständlich und wird nicht vom Programm übernommen. Trotzdem wäre es schön, wenn Prolog uns diese Aufgabe abnimmt.

3 Metaprogrammierung

3.1 Anwendungsgebiete der Metaprogrammierung

Künstliche Intelligenz ist sicher das Anwendungsgebiet mit der größten Zukunft für die Metaprogrammierung. Aber auch schon einfache Sachen wie das Merken von Ergebnissen ist Metaprogrammierung und diese lässt sich besonders gut in Prolog unterbringen.
Wenn ein Mensch ein bestimmtes Problem mit einem Computerprogramm lösen will, so geschieht dies häufig, indem er es auf bereits bekannte Probleme vereinfacht. Diese Aufgabe ist durch Metaprogrammierung effizienter und verständlicher lösbar, da durch Automation in der Programmierung die Fehleranfälligkeit gesenkt und die Produktivität gesteigert werden kann. Gerade dieser Prozess kann zudem durch Metaprogrammierung vereinfacht werden, weil diese es ermöglicht, die verwendete Sprache um neue Konstruktionen zu erweitern und so die Sprache dem Problem anzupassen. Daher eignen sich Sprachen, welche die Metaprogrammierung einfach umsetzen können, besonders gut zur Erstellung Domänenspezifischer Sprachen. Aber was ist nun bereits ein Metaprogramm? Wir sprechen nämlich nicht nur bei künstlicher Intelligenz von Metaprogrammen. Selbst kleinste Programme können Metaprogramme sein. So ist das automatische Hinzufügen von Gettern und Settern in Java durch die IDE bereits eine metalogische Funktion.

3.2 Vorstellung verschiedener metalogischer Prädikate

Es existieren zusätzlich zu den bereits erwähnten extralogischen Prädikaten noch andere Prädikate jenseits des logischen Modells. Die folgenden Prädikate werden besonders zur Metaprogrammierung verwendet. Prolog ist zudem eine homoikonische Sprache. Mit dieser Eigenschaft, die besagt, dass Programme Strukturen der eigenen Sprache sind, ist

es leicht den Code zu modifizieren, während er ausgewertet wird. Besonders einfach ist dies mit den Prädikaten **assert** und **retract**. Es folgt eine Übersicht über verschiedene Meta-Prädikate. Wir verwenden die bereits bekannte Schreibweise.

assert/1 **assert**(+Term) überprüft, ob *Term* ein gültiger Fakt oder eine gültige Regel in Prolog ist und fügt diesen *Term* dann ans Ende des Programms ein.
Eine Variante ist **asserta**(+Term). Diese fügt den *Term* an den Anfang des Programms ein.

retract/1 **retract**(+Term) entfernt den ersten Term, der mit dem übergebenen *Term* unifiziert werden kann. Varianten sind **retractall**(+Term), welches alle Terme entfernt, die mit *Term* unifiziert werden können, und **retractz**(+Term), welches den letzten Term entfernt, der mit *Term* unifiziert werden kann.

freeze/1 **freeze**(+Term) friert das Backtracking auf *Term* ein und ermöglicht es so, bestimmte Zweige des Beweisbaumes abzuschneiden, jedoch anders als beim *Cut* ist es möglich, diese wieder aufzutauen. Dies kann hilfreich sein, wenn aus bestimmten Variablen-Belegungen ersichtlich ist, in welche Richtung Prolog suchen soll.

melt/1 **melt**(+Term) bildet das Pendant zu **freeze**(+Term). Daher taut es eingefrorene Zweige wieder auf. [4]

Zusammen mit den bereits bekannten extralogischen Prädikaten können wir nun noch mächtigere Programme erstellen, die sich unter Umständen sogar selber weiter schreiben.

3.3 Beispiele für Metaprogramme

Möchte man in Prolog alle Ergebnisse zu einer Anfrage erhalten, ohne mehrfach nach weiteren Lösungen zu fragen, so kann man das Prädikat **findall** benutzen, welches sich in folgendem Beispiel mithilfe der Metaprogrammierung implementieren lässt.[1]

```
findall(X, Anfrage, Xlist) :- Anfrage,
                              assert(loesung(X)),
                              fail;
                              sammleLoesungen(Xlist).

sammleLoesungen([X | Rest]) :- retract(loesung(X)),
                               !,
                               sammleLoesungen(Rest).
sammleLoesungen([]).
```

Dies ist hilfreich, wenn die Berechnung eines einzelnen Ergebnisses mehrere Minuten in Anspruch nimmt und somit in den Hintergrund verschoben werden kann, da Prolog nun nicht nach jedem Ergebnis auf eine erneute Benutzer-Eingabe wartet. Dies funktioniert indem Prolog eine Lösung findet, diese in die dynamische Datenbank schreibt und die weitere Auswertung mit dem Prädikat **fail** unterbindet, sodass nach einer weiteren Lösung gesucht wird. Sollte es keine Lösungen mehr geben sammelt Prolog alle gefunden Lösungen ein und gibt diese gebündelt aus.
Ein weiteres Beispiel ist ein Mittel zur Steigerung der Effizienz. Ein sogenanntes Memo merkt sich ein bereits errechnetes Ergebnis und kann so bei der Berechnung weiterer

Lösungen beitragen. In Prolog kann dies mit dem bekannten Prädikat **assert** implementiert werden. Wenn nun ein Ausdruck zu seiner Korrektheit ein bereits vorher berechnetes Ergebnis benötigt, so wurde dieses gespeichert und wird verwendet, anstelle es erneut zu berechnen.

```
:- dynamic savedfib(_,_).
fib(1,1).
fib(2,1).
fib(X,Y) :- savedfib(X,Y), !;
            X1 is X - 1, fib(X1,Y1),
            X2 is X - 2, fib(X2,Y2),
            Y is Y1 + Y2,
            assert(savedfib(X,Y)), !.
```

Das Beispiel zeigt die Fibonacci-Funkion, welche zu ihrer Bestimmung rekursiv die vorangegangenen Folgenglieder bestimmt und die beiden letzten addiert. Gerade für sehr große Anfragen benötigt es exponentiell viele weiteren Aufrufe und diese zum Teil sogar mehrfach. Wird zum Beispiel *fib(5,X)* aufgerufen, so werden rekursiv die Aufrufe *fib(4,X)* und *fib(3,X)* getätigt. *fib(4,X)* tätigt nun die Aufrufe *fib(3,X)* und *fib(2,X)*. Somit muss für die Berechnung eines Aufrufs mehrfach derselbe Ausdruck ausgewertet werden. Doch mithilfe der Metaprogrammierung fällt dieses Problem weg, da ein Ausdruck, der bereits einmal ausgewertet wurde, nicht erneut berechnet werden muss.
Auch im normalen Studenten-Alltag ist Prolog zusammen mit seinen extralogischen Prädikaten sehr hilfreich. Schon ein kurzes Programm reicht aus, um das zu leisten, was andere Sprachen nur mit viel mehr Code ausführen können.

3.4 Implementierung der metalogischen Prädikate

Implementieren wir nun unsere erlangten Kenntnisse in unser kleines Spiel:

```
game :- bed1(1).
bed1(X) :- translate(X), property(X,_), X1 is X + 1,
           (read(Y), Y = ja, bed2(X,X1), !;
           bed1(X1)), !;
           addBed(X).
bed2(A,X) :- translate(X), property(X,_), X1 is X + 1,
             (read(Y), Y = ja, bed3(A,X,X1), !;
             bed2(A,X1)), !;
             addBed(A,X).
bed3(A,B,X) :- translate(X), property(X,_), X1 is X + 1,
               (read(Y), Y = ja, searchAnimal(A,B,X), !;
               bed3(A,B,X1)), !;
               addBed(A,B,X).
```

Fortsetzung auf der nächsten Seite

```
translate(X) :- property(X,Y),
                write('Hat dein Tier die Eigenschaft: '),
                write(Y), write(" ?\n");
                write('Ich kenne keine weitere Eigenschaft.\n').
searchAnimal(A,B,C) :- property(A,A1), property(B,B1), property(C,C1),
                       tier(A1,B1,C1,X), write('Ist dein Tier ein '),
                       write(X), write('?\n'), read(Y), Y = ja,
                       write('Das System macht keine Fehler!'), !;
                       write('Ich kenne dein Tier nicht.\n'),
                       addAnimal(A,B,C).
% new Code below
addBed(X) :- write('Brauche eine weitere Eigenschaft.\n'), read(Y),
             assert(property(X,Y)), X1 is X + 1, addBed(X,X1).
addBed(A,X) :- write('Brauche eine weitere Eigenschaft.\n'), read(Y),
               assert(property(X,Y)), X1 is X + 1, addBed(A,X,X1).
addBed(A,B,X) :- write('Brauche eine weitere Eigenschaft.\n'), read(Y),
                 assert(property(X,Y)), addAnimal(A,B,X).
addAnimal(A,B,C) :- write('Ein Tier mit folgenden Eigenschaften:\n'),
                    property(A,A1), property(B,B1), property(C,C1),
                    write(A1), write(', '), write(B1),
                    write(', '), write(C1), write('\n'),
                    read(Y), assert(tier(A1,B1,C1,Y)),
                    write('Neues Tier in Datenbank aufgenommen.').
% Data-Entries
```

Nach der Implementierung der metalogischen Prädikate können wir nun auch neue Eigenschaften und neue Tiere hinzufügen. Dafür haben wir die Prädikate *addBed*/1, *addBed*/2, *addBed*/3 und *addAnimal*/3 hinzugefügt. Ist keine passende Eigenschaft für das ausgesuchte Tier in der Datenbank, wird *addBed*/1 aufgerufen, welches den Benutzer auffordert eine Eigenschaft anzugeben. Diese Eigenschaft wird dann mit der nächsthöheren Zahl in die dynamische Datenbank aufgenommen, daraufhin wird *addBed*/2 aufgerufen. Dieses Prädikat wird auch aufgerufen, wenn nur eine passende Eigenschaft für das ausgesuchte Tier bereits in der Datenbank ist. Ebenso führt es die Zahl der bereits bekannten Eigenschaft mit und fragt den Benutzer nach einer weiteren Eigenschaft des Tieres. Genauso wie bei *addBed*/1 wird diese Eigenschaft in die dynamische Datenbank aufgenommen. Daraufhin wird *addBed*/3 aufgerufen. Dieses Prädikat wird auch aufgerufen, wenn nur zwei passende Eigenschaften für das ausgesuchte Tier in der Datenbank sind. Dieses Prädikat führt die beiden Zahlen der bereits bekannten Eigenschaften des Tiers mit sich und fragt nach einer dritten Eigenschaft. Genauso wie bei *addBed*/1 wird diese Eigenschaft in die dynamische Datenbank aufgenommen. Daraufhin wird *addAnimal*/3 aufgerufen. *addAnimal*/3 wird aufgerufen, wenn alle drei Eigenschaften in der Datenbank sind aber das Tier noch nicht. Der Benutzer wird dann nach dem Namen des Tieres gefragt. Dieser wird dann mit seinen bereits bekannten Eigenschaften der dynamischen Datenbank hinzugefügt. Als Bestätigung wird dem Benutzer mitgeteilt, dass das Tier in die dynamische Datenbank aufgenommen wurde.

Wenn man das Spiel nun mit demselben Tier spielt, welches man neu hinzugefügt hat, wird dieses auch vom Programm gefunden und wir haben unsere Datenbank erfolgreich um weitere Eigenschaften und ein neues Tier erweitert.

4 Zusammenfassung

In dieser Ausarbeitung haben wir die Grenzen des prädikatenlogischen Modells von Prolog kennengelernt. Wir haben über die Vor- und Nachteile von Seiteneffekten gesprochen und die extralogischen Prädikate eingeführt, die außerhalb des prädikatenlogischen Modells liegen. Zusätzlich sind wir genauer auf metalogische Prädikate eingegangen.
Wir sehen also, dass Seiteneffekte nicht nur Nachteile haben. Sie mögen zwar den Code undurchsichtiger machen, trotzdem sind sie für manche Funktion unentbehrlich. So können wir z.B. die Ein- und Ausgabe im prädikatenlogischen Modell von Prolog nicht realisieren. Stattdessen benötigen wir die extralogischen Prädikate, welche außerhalb des prädikatenlogischen Modells liegen. Diesen gelingt es nur mithilfe von Seiteneffekten, die gewünschte Funktion zu realisieren. Wir haben *read*/1 und *write*/1 für die Ein- und Ausgabe kennengelernt, *var*/1 und *nonvar*/1 um zu überprüfen, ob der übergebene Term eine Variable ist oder nicht und *atomic*/1 und *compound*/1 um zu überprüfen, ob der übergebene Term weiteres Backtracking benötigt.
Des Weiteren gibt es extralogische Prädikate, welche man unter einer neuen Kategorie zusammenfassen kann, die metalogischen Prädikate. Diese ermöglichen es Prolog die Metaprogrammierung zu realisieren. Für diese gibt es sehr weite Anwendungsgebiete von künstlicher Intelligenz bis hin zur einfachen Code-Erzeugung durch die IDE. Da Prolog eine homoikonische Sprache ist d.h. das Programm ist Datenstruktur der eigenen Sprache, ist die Metaprogrammierung auch sehr leicht in dieser umzusetzen. Dafür gibt es die metalogischen Prädikate *assert*/1 und *asserta*/1, welche die ihnen übergebenen Terme ans Ende oder an den Anfang des Programms hinzufügen, *retract*/1, *retractall*/1 und *retractz*/1, welche die ihnen übergebenen Terme aus dem Programm entfernen, *freeze*/1 und *melt*/1, welche es ermöglichen, bestimmte Zweige in einem Baum einzufrieren oder wieder aufzutauen.

5 Literatur

[1] Jürgen Giesl. *Logikprogrammierung.* Skript zur Vorlesung, RWTH Aachen, 2015.

[2] Michael Hanus. *Problemlösen mit PROLOG.* MikroComputer-Praxis. Vieweg+Teubner Verlag, 2013.

[3] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, MA, USA, 1986.

[4] SWI-Prolog. https://www.swi-prolog.org/, 2019.