Cut und Negation

Thomas Kronabeter
Danilo Lourenco Trotta

June 2017

1 Einführung

Diese Ausarbeitung behandelt das Konzept von Cut und Negation in der Programmiersprache Prolog. Beides sind System-Prädikate, die einen besonderen Stellenwert in Prolog genießen. Die Turingmächtigkeit ist nämlich auch ohne diese gewährleistet. Dabei bringt es allerdings neue Möglichkeiten der Performanceanpassung und der Codestrukturierung mit sich. Das ist bei einer langsam auswertenden Programmiersprache wie Prolog besonders wichtig, vor allem bei Projekten größerer Dimension. Im Folgenden werden wir zuerst auf die Syntax des Cuts eingehen und uns dessen Arbeitsverhalten im Detail anschauen. Hierbei ist der Schwerpunkt besonders auf die Evaluation von Anfragen gesetzt, die auf Programme mit Cut zurückgreifen. Danach behandeln wir die drei häufigsten Anwendungsfälle des Cut-Operators. Außerdem gehen wir auf die Tücken ein, die bei der Nutzung entstehen können, und klassifizieren anhand dieser die verschiedenen Cuts. Zum Schluss gehen wir auch noch auf weitere Konzepte des Cuts ein. Dabei wird die Negation noch einmal eine gesonderte Rolle spielen.

2 Cut

2.1 Syntax

Der Cut ist ein nullstelliges System-Prädikat. Er wird mit einem Ausrufezeichen dargestellt. Für eine Stelligkeit n eines Prädikats *name* werden wir im weiteren Verlauf die Notation $name/n$ verwenden. Als Beispiel hierfür verwenden wir das Cut-Prädikat, das dann als !/0 geschrieben wird. Für den Cut-Operator gelten dieselben syntaktischen Regeln wie für andere Prädikate in Prolog auch. Damit kann dieser sowohl als Bedingung in Klauseln als auch in Anfragen vorkommen. Eine Klausel K/m mit Parametern $P_1, ..., P_m$ und Bedingungen $A_1, ..., A_i, A_{i+1}, ..., A_n$, wobei der Cut hinter der i-ten Bedingung angewendet wird, wird wie folgt deklariert:

$K(P_1, ..., P_m) :- A_1, ..., A_i, !, A_{i+1}, ..., A_n.$

Ein Prädikat *example*/1 mit

```
example(_):- !.
```

ist dann ein korrekter Ausdruck. Wird das Ausrufezeichen als Parameter einer Klausel verwendet, so nimmt es das Verhalten einer Konstanten an. Bei einem Programm

```
example_2(!).
```

wird die Anfrage `?-example_2(X).` als Antwort `X = !` ausgeben. Besonders folgt daraus auch, dass der später benannte Effekt bei der Nutzung als Prädikat nicht als Parameter wirkt.

2.2 Arbeitsweise

Das Cut-Prädikat hebt sich mit seiner Arbeitsweise grundlegend von anderen Prädikaten ab, denn das Suchverhalten einer Anfrage wird modifiziert. Anhand eines Anwendungsbeispiels wird diese Änderung am deutlichsten, weshalb wir example/1

$example(X) : -A_1, ..., A_l$
$example(X) : -B_1, ..., B_i, !, B_{i+1}, B_m$
$example(X) : -C_1, ..., C_n$

definieren. Hierbei sind A_{i_1},B_{i_2} und C_{i_3}, mit $i_1 \in [1, l]$,$i_2 \in [1, m]$, $i_3 \in [1, n]$, Bedingungen der jeweiligen Klauseln. Wird nun eine beliebige Anfrage betreffend example/1 gestellt, so wird die Suche bis zum Erreichen des Cut-Operators, wie gewöhnlich, abgearbeitet.
Das bedeutet, dass die Klauseln im Programm von oben nach unten durchsucht werden. Ist eine Klausel mit passenden Parametern zur Anfrage gefunden, so werden die Bedingungen der Klausel von links nach rechts überprüft. Dabei können Variablen gesetzt werden, um die Bedingungen zu erfüllen. Sind alle Bedingungen der Klausel erfüllt, so ist eine Antwort gefunden. Wird allerdings eine Bedingung falsifiziert, so wird auf eine vorherige Bedingung zurückgesprungen und eine andere Variablenkombination ausprobiert. Dies ist als Backtracking bekannt. Ein entsprechender Beweisbaum ist in Abb. 2.2.1 zu sehen.

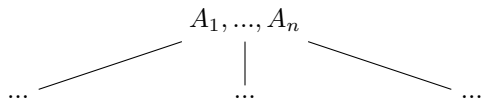

Abb. 2.2.1

Gibt es für die Klausel keine Variablenbelegung, so wird nach einer weiteren passenden Klausel in den Folgezeilen gesucht. Kommen wir nun aber wieder zu unserem Programm example/1 . example/1 hat in diesem Fall drei Klauseln. Nun sei unsere Anfrage `?-example(X).`. Wie bereits erwähnt, ist bis zum Erreichen des Cut-Operators das Suchverhalten gewöhnlich verlaufen. Gehen wir also davon aus, dass die erste Klausel keine Lösung besitzt und die Suche in der zweiten Klausel fortgesetzt wird. Jetzt seien Bedingungen $B_1, ..., B_i$ bereits verifiziert. Für die Verifizierung wurden derweilen die Variablen $V_1, ..., V_k$ gesetzt. Im nächsten Schritt wird der Cut-Operator überprüft. Der Cut ist immer erfüllt, also `true`. Ist der Cut-Operator erst einmal verifiziert, so werden noch folgende Klauseln nicht mehr untersucht. Befindet sich also in dieser Klausel keine Lösung, wird `false` zurück gegeben. Des weiteren ist der Suchraum erheblich eingeschränkt, denn alle bis zu diesem Zeitpunkt gesetzten Variablen, $V_1, ..., V_k$ sind nicht mehr änderbar. Dies hat zur Folge, dass das Backtracking nur bis zum Cut-Operator läuft und `false` ausgibt. Alle Variablen, die noch nicht gesetzt sind, können mit Hilfe des Backtrackings jedoch geändert werden. Dies ist in Abb. 2.2.2 noch einmal verdeutlicht. Die gestrichelten Pfade werden dabei nicht durchlaufen.

Sollte hinter dem Cut-Operator eine Lösung gefunden werden, so ist dies die letzte ausgegebene Lösung. Danach können keine weiteren Lösungen mit `;` angefragt werden. Das kann dann zur Folge haben, dass eine Anfrage möglicherweise nur eine Lösung ausgibt, obwohl es eventuell mehrere geben könnte.

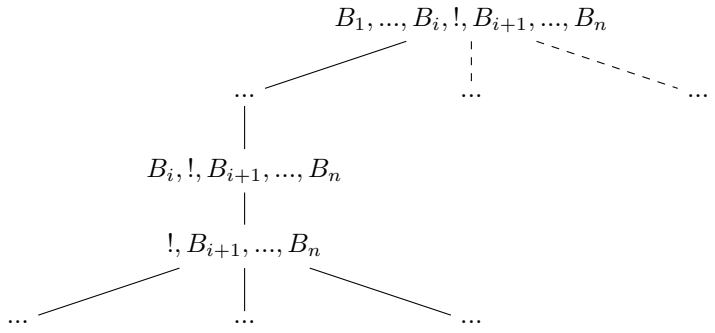

Abb. 2.2.2

3 Verwendung

3.1 Beispiel

In den nachfolgenden Kapiteln werden wir uns immer wieder Abwandlungen eines bestimmten Beispiels bedienen. Deswegen werden wir hier kurz eine Einführung in das Ursprungsprogramm geben.
Das Programm enthält nur ein Prädikat $maxList/3$. Dabei werden nur folgende Parameterkombinationen akzeptiert. Die drei Parameter sind jeweils Listen der gleichen Länge. Die Elemente der Listen sind ausschließlich natürliche Zahlen. Zudem ist das i-te Element der letzten Liste das Maximum der i-ten Elemente der ersten und zweiten Liste. Mathematisch betrachtet ist das i-te Element der ersten und zweiten Liste jeweils das erste bzw. zweite Element eines i-ten zweidimensionalen Vektors einer Vektorenliste. Die letzte Liste entspricht der Liste der Maximums-Norm der zuvor genannten Vektorliste.

```
maxList([],[],[]).
maxList([H1|L1],[H2|L2],[H|L]):- H1 < H2, H2 = H, maxList(L1, L2, L).
maxList([H1|L1],[H2|L2],[H|L]):- H1 >= H2, H1 = H, maxList(L1, L2, L).
```

Um Übersichtlichkeit gewährleisten zu können, haben wir die Überprüfung der Elemente auf nicht natürliche Zahlen ausgelassen. Behalten Sie aber im Hinterkopf, dass das obige Programm in dieser Art und Weise nicht die beschriebene Funktion erfüllt sondern nur, wenn eine entsprechende Prüfung hinzugenommen wird. Gehen wir nun genauer auf das Programm ein. Die erste Zeile ist nur aufgrund der Terminierung vorhanden. Zeile zwei und drei überprüfen, ob entweder das erste Element der ersten oder zweiten Liste das Maximum ist und dem ersten Element der dritten Liste entspricht. In beiden Fällen werden dann rekursiv die Restlisten geprüft.

3.2 Einsatz und Nutzung

Das Cut-Prädikat kann in allerlei Programmen Anwendung finden. Bei der Betrachtung mehrerer Szenarien fallen hierbei aber drei Verwendungen besonders in Augenschein. Diese werden wir uns nun genauer anschauen. Das erste Anwendungsgebiet des Cut-Operators ist das Auslassen von Klauseln beim Beweisen. Hier haben zwei oder mehr Klauseln, Bedingungen die sich gegenseitig Ausschliessen(Mutual Exclusion). Das ist in unserem Beispiel maxList/3 auch der Fall. Die zweite und dritte Klausel

```
maxList([H1|L1],[H2|L2],[H|L]):- H1 < H2, H2 = H, maxList(L1, L2, L).
maxList([H1|L1],[H2|L2],[H|L]):- H1 >= H2, H1 = H, maxList(L1, L2, L).
```

enthalten die Bedingungen $H1 < H2$ bzw. $H1 >= H2$. Die Aussagen sind widersprüchlich zueinander. Daraus folgt, dass beim Beweisen mit der bereits ausgewählten zweiten Klausel die dritte Klausel keine Lösung besitzt. Hier lässt sich der Cut leicht hinter der Bedingungen $H1 < H2$ einfügen, wie es im folgenden Programm zu sehen ist:

```
maxList([],[],[]).
maxList([H1|L1],[H2|L2],[H|L]):- H1 < H2, !, H2 = H, maxList(L1, L2, L).
maxList([H1|L1],[H2|L2],[H|L]):- H1 >= H2, H1 = H, maxList(L1, L2, L).
```

Damit lässt sich unnötiger Aufwand einsparen.
Eine andere Verwendung findet der Cut-Operator in Verbindung mit dem fail-Operator (Anmerkung: fail/0 ist ein Operator, der immer `false` evaluiert). Diese Verbindung wird in der Literatur auch häufiger als Cut-Fail-Kombination bezeichnet. Die Cut-Fail-Kombination wird in Szenarien benutzt, bei denen es darum geht, ein Fehlschlagen so früh wie möglich zu bemerken. Meist wird in einer solchen Situation eine Klausel voran gestellt, die explizit Elemente definiert, die keine Lösung darstellen und dann durch Cut-Fail verworfen werden. Spielt es also eine größere Rolle eine Anfrage früher zu falsifizieren, als sie zu verifizieren, kann der Cut-Fail verwendet werden. Zur Veranschaulichung fügen wir unserer ursprünglichen maxList/3 eine Klausel hinzu. Diese stellen wir allen anderen voran. Unser Programm sieht nun wie folgt aus:

```
maxList([H1|_],[H2|_],[H|_]):- H < H1, H < H2, !, fail.
maxList([],[],[]).
maxList([H1|L1],[H2|L2],[H|L]):- H1 < H2, H2 = H, maxList(L1, L2, L).
maxList([H1|L1],[H2|L2],[H|L]):- H1 >= H2, H1 = H, maxList(L1, L2, L).
```

Kommen wir nun zu einem weiteren Szenario. Ist es erwünscht, dass nur eine Lösung ausgegeben wird, so kann der Cut dafür verwendet werden. Für ein passendes Beispiel ändern wir maxList/3 etwas ab. Neben der arabischen Schreibweise (1,2,3,...) werden wir nun auch a^i als Zahlen für unsere maxList/3 zulassen. Hierbei ist $a^i = i$ und i ist eine natürliche Zahl. Auch in diesem Fall lassen wir der Übersicht wegen die Überprüfung der Elemente auf korrekte Notation aus. Allerdings verwenden wir anstelle von $<$, $=$ und $>=$ hier lt/2, geq/2 und eq/2, um die verschiedenen Darstellungen vergleichen zu können. Wird unter dieser Voraussetzung eine Anfrage gestellt, so wird eine Lösung unter verschiedenen Darstellungen mehrmals ausgegeben. In diesem Fall hat es für uns wenig Sinn, dieselbe Lösung erneut auszugeben. Daher nehmen wir dem Nutzer die Möglichkeit, eine zweite Lösung zu erfragen, indem wir den Cut als Schlussbedingung jeder Klausel hinzufügen.

```
maxList([],[],[]):- !.
maxList([H1|L1],[H2|L2],[H|L]):- lt(H1, H2), eq(H2, H), maxList(L1, L2, L), !.
maxList([H1|L1],[H2|L2],[H|L]):- geq(H1, H2), eq(H1, H), maxList(L1, L2, L),!.
```

Allgemein lässt sich Ableiten, dass der Cut durch Stoppen des Backtrackings und weiterer Beweispfade die Performance erheblich steigern kann.

3.3 Green und Red Cuts

3.3.1 Verschiedene Arten des Cuts

Nachdem wir nun den Einsatz und die Nutzung des Cuts kennengelernt haben, kommen wir zu den unterschiedlichen Arten des Cuts. Im Allgemeinen lassen sich diese in zwei verschiedene Arten unterteilen, nämlich in Green und Red Cuts. Dabei können wir uns eine "Fußgänger-Ampel" vorstellen und uns vereinfachend merken, dass Green dabei für: "Alles in Ordnung!" und Red dagegen für: "Achtung, aufpassen!" steht.

3.3.2 Green Cuts

Als Green Cut bezeichnen wir die Art von Cut, die nie etwas an der Logik eines Programmes verändert. Demnach ändert sich auch an dem zu erwartenden Ergebnis nichts. Deshalb auch die

Namensgebung "Green Cut", wie in 3.3.1 mit unserer "Fußgänger Ampel" versucht zu visualisieren. Schauen wir uns dazu unser bis hierhin verwendetes Beispiel noch einmal an.

```
maxList([],[],[]), ! .
maxList([H1|L1],[H2|L2],[H|L]):- H1 < H2, H2 = H, maxList(L1, L2, L), ! .
maxList([H1|L1],[H2|L2],[H|L]):- H1 >= H2, H1 = H,         maxList(L1, L2, L), ! .
```

Dieser Code ist bis auf die Cut-Operatoren identisch zu unserem ersten in 3.1 eingeführten Beispiel. In diesem Beispiel handelt es sich nun um einen Green Cut, demnach muss die Auswertung aller Anfragen an dieses Programm identisch zu unserem Beispiel Code ohne Cut-Operatoren aus 3.1 sein. Der alleinige Unterschied besteht darin, dass mit diesem Programm nun bei einer Anfrage nach einem bestimmten `X` nur ein einziges Ergebnis ausgegeben wird und das Programm danach terminiert. Das steht natürlich im Gegensatz dazu, dass wir normalerweise nach einer solchen Anfrage durch die Eingabe eines `;` nach weiteren potentiellen Ergebnissen von `X` suchen könnten. Bei der Einstufung als Green Cut hat das allerdings keine Auswirkung, solange natürlich das Ergebnis das selbe ist. Eine solche frühe Terminierung bewirkt dann wie in 3.2 bereits angemerkt eine Performancesteigerung. Meist wird dies in Fällen eines gegenseitigen Ausschlusses ausgenutzt, um mit dem Green Cut dann einen effizienteren Code zu haben. Können wir in der ersten Klausel eine Anfrage schon zu `true` evaluieren, macht es selten Sinn trotzdem noch die zweite Klausel zu überprüfen. Indem wir diese zweite Überprüfung nicht mehr ausführen, kann das natürlich positive Auswirkungen auf die Performance eines Programmes haben.

Um das ganze visuell besser zu verdeutlichen, zeigen wir nun anhand eines Beweisbaums den Unterschied.

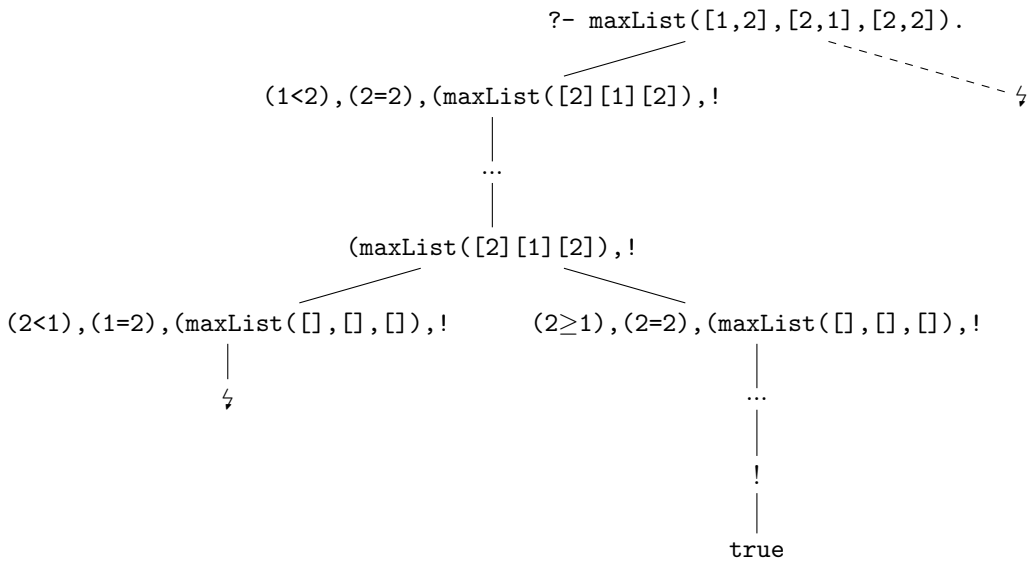

Abb. 3.3.2.1

Wie wir sehen, fällt der komplette rechte Teilbaum des Beweisbaums weg, da wir mit dem Cut-Operator das Backtracking gestoppt haben. Da unser Programm nun nicht mehr alle möglichen Pfade durchsuchen muss, spart es logischerweise Zeit und Ressourcen. Dies führt zu der genannten Performanceverbesserung. Am Ergebnis selber hat sich jedoch nichts verändert, da es immer noch die richtige Auswertung zurückgibt wie beim Programm ohne Cut-Operator. Deshalb handelt es

sich hier um einen Green Cut. Es lässt sich noch anmerken das Cuts die als letzte Bedingung an alle Klauseln angefügt werden, häufig Green Cuts sind.

3.3.3 Red Cuts

Als Red Cuts bezeichnen wir alle Cuts, die keine Green Cuts sind. Das heißt wiederum, dass Cuts, die Auswirkungen auf die Funktionalität und die Logik eines Programmes haben, Red Cuts sind. Dies hat zur Folge, dass ein Programm, in dem man nachträglich diese Cuts hinzufügt, danach einen anderen Output generieren kann, als davor ohne diese Cuts.
Stellen wir uns folgendes Szenario vor. Wir schreiben ein Programm, in welchem wir den Cut-Operator verwenden. Unser Programm funktioniert und wertet wie von uns erwartet auch richtig aus. Entfernen wir jedoch unseren Cut-Operator nun aus dem Programm, wertet es unsere Anfragen nicht mehr wie gewollt aus. Wir wissen also, unser Cut ist ein Red Cut, da dieser Auswirkungen auf die Funktionalität unseres Programms hat. Anders als bei Green Cuts, die hauptsächlich für die Performanceverbesserung verwendet werden, haben Red Cuts andere wichtige Vorteile. Viele erweiterte Strukturen in Prolog, die auf Cut aufbauen, wie z.B Negation 4.1 und If-then-else 4.3, sind nur durch Cuts möglich. Da diese Strukturen ohne den Cut-Operator nicht funktionieren würden, sind es Red Cuts.
Es ist sehr schwer, Red Cuts von Green Cuts in einem gegebenen Code bei bloßem Anschauen zu unterscheiden. Meist hilft tatsächlich nur das Ausführen des Programm Codes mit unterschiedlichen Anfragen und der Vergleich zum selben Code ohne Cut-Operatoren. Alternativ ist das Erstellen eines Beweisbaums auch eine gute Methode um zu überprüfen, um welche Art von Cut es sich im gegebenen Programm handelt.

Im Nachfolgenden schauen wir uns erneut unser Beispiel aus 3.1 an.

```
maxList([ ] ,[ ] ,[ ]).
maxList([H1|L1], [H2|L2], [H|L]) :- H1 < H2,!, H2 = H , maxList(L1,L2, L).
maxList([H1|L1], [H2|L2], [H1|L]) :- maxList(L1, L2, L).
```

Wir haben hier die zweite Klausel etwas abgeändert. Unser Programm funktioniert in dieser Konstellation genau wie es soll und wertet unsere Anfragen richtig aus. Entfernen wir aber aus diesem Code den Cut-Operator in der ersten Klausel, dann erhalten wir falsche Auswertungen, es handelt sich offensichtlich um einen Red Cut. Wir zeigen dies am besten wieder visuell anhand eines Beweisbaums und stellen folgende Anfrage an unser Programm:

```
?- maxList( [2,1] , [1,2] , [2,1] ).
```

Unser Programm gibt hier `false` zurück, was offensichtlich richtig ist. Die selbe Anfrage beim selben Code ohne den Cut-Operator gibt `true` zurück, dies stimmt offensichtlich nicht. Schauen wir uns an wie Prolog diese beiden Anfragen auswertet.

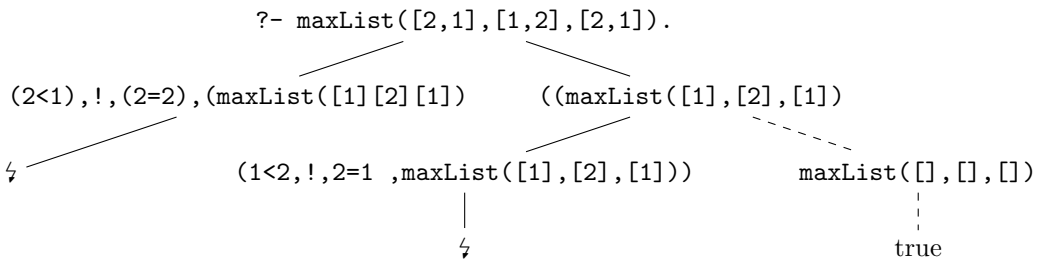

Abb 3.3.3.1

Es wird also `false` ausgegeben. Unsere Anfrage unifiziert sich mit der zweiten Klausel im Code. Prolog überprüft nun die Bedingungen dieser Klausel. Die erste Bedingung wird zu `false` ausgewertet, Prolog führt Backtracking aus und schaut, ob es unsere Anfrage noch auf eine weitere Klausel unifizieren kann, dies ist mit der dritten Klausel möglich. Nun wird versucht, die Bedingung der dritten Klausel zu beweisen. In diesem Fall ist die Bedingung eine erneute Anfrage von `maxList/3`. Diese zweite Anfrage unifiziert sich wieder mit der zweiten Klausel unseres Codes, hier wird die erste Bedingung nun zu `true` ausgewertet, die zweite Bedingung ist unser Cut-Operator, welcher immer zu `true` ausgewertet wird, unsere dritte Bedingung wird jedoch zu `false` ausgewertet, da wir aber schon über den Cut-Operator gelaufen sind, stoppt das Backtracking und Prolog wertet die komplette Anfrage zu `false` aus.

Anhand des gestrichelten Pfades können wir sehen, was zusätzlich passiert, wenn wir den Cut-Operator entfernen. Prolog würde Backtracking ausführen können und die Anfrage wieder auf die dritte Klausel unifizieren können. Wir würden erneut die Bedingung prüfen und diese Bedingung würde zu `true` ausgewertet werden. Unsere komplette Anfrage wäre also `true`.

Wie wir sehen, ist die Benutzung von Cuts beziehungsweise das Weglassen dieser in manchen Fällen nicht ganz unproblematisch.

4 Erweiterte Strukturen und Konzepte

Wir kennen nun einige wichtige Ursachen für die Verwendung von Cuts. In erster Linie stehen Performanceverbesserung sowie Strukturen, die ohne Cut nicht möglich wären. Mit diesen Konzepten befassen wir uns nun in diesem Kapitel. Die folgenden Strukturen bauen in Prolog auf dem Cut-Operator auf und wären ohne diesen, so wie wir ihn kennen, nicht möglich.

4.1 Negation

4.1.1 Konzept

Prolog besitzt keine normalerweise übliche Negation wie wir sie aus anderen Programmiersprachen wie Java, Haskell und C kennen. Das Prinzip der Negation in Prolog basiert darauf zu prüfen, ob eine Klausel nicht verifiziert werden kann. Ist dies der Fall, dann evaluiert Prolog zu `true`.

4.1.2 Struktur und Syntax

Es gibt mehrere Arten die Negation in Prolog zu verwenden. Üblicherweise benutzt man das Prädikat `not/1`. Der moderne Programmierstil präferiert die `\+ /1` Schreibweise. Dabei sind beide Prädikate äquivalent zueinander. Zur jetzigen Zeit wird die `\+ /1` Schreibweise lediglich der Verständlichkeit halber bevorzugt. Die Schreibweise `not/1` führt gerne zu Verständnisproblemen, da man intuitiv als Programmierer davon ausgeht, es bedeute: "X ist nicht wahr". Prolog arbeitet jedoch nicht so. Vielmehr wäre "X kann nicht bewiesen werden" eine genauere Beschreibung. Das `not` Prädikat baut auf dem Cut bzw. dem Red Cut auf, da dieser dafür essentiell notwendig ist, damit die Negation korrekt funktioniert. Wir können anhand des Cut-Operators im Zusammenspiel mit dem `Fail`-Operator eine eigene Negation deklarieren.
Diese könnte so aussehen:

```
neg(X) :- X, ! , fail.
```

```
neg(X).
```

4.1.3 Beispiele

Für die Negation schauen wir uns ein neues Beispiel an. Wir haben folgenden Prolog Code:

```
equal(X,Y) :- X = Y.
```

Dieses simple Programm sagt uns einfach, ob zwei in unserer Anfrage gegebenen Zahlen gleich sind oder nicht. Die Anfrage `?- equal(3,3).` würde also `true` auswerten. Stellen wir nun dieselbe Anfrage mit Negation: `?- \+ equal(3,3).` Dies würde zu `false` ausgewertet werden. Man kann die Negation auch direkt im Code einbauen. Wir verändern unseren Code nun zu:

```
equal(X,Y) :- \+ X = Y.
```

Nun würde dieselbe Anfrage, die wir vorhin benutzt haben, `?- equal(3,3).` auch zu `false` evaluieren. Unser Programm würde also theoretisch nun überprüfen, ob zwei in der Anfrage gegebene Zahlen ungleich zueinander sind. Es ist aber wichtig, dass wir das Prinzip der Negation in Prolog verstehen. Die Anfrage `?- \+ equal(X,3).` erscheint auf den ersten Blick so, als würde die Anfrage gleichbedeutend zu "Gibt es ein `X`, für das diese Anfrage `false` ausgibt?" sein. Das ist aber nicht die Negation in Prolog, sondern die Anfrage schlägt fehl, falls ein `X` existiert, sodass `equal(X,3)` bewiesen werden kann, Somit ist sie gleichbedeutend mit: "Es existiert kein `X`, für das diese Anfrage `true` ergibt.

4.2 Cut-Fail Kombination

Wie bereits vorhin erwähnt, besteht die Negation in Prolog aus einer geschickten Kombination des Cut- und des Fail-Operators. Wir zeigen anhand unseres Beispiels und eines Beweisbaums, wie genau Prolog bei der Verwendung der Negation vorgeht. Wir erinnern uns:

```
not(X) :- X, !, fail.
not(X).
```

ist die Negation. Wir stellen eine Anfrage an das `equal` Programm aus 4.1.3. Der Beweisbaum zur Anfrage `?- not(equal(3,3)).` würde dementsprechend so aussehen.

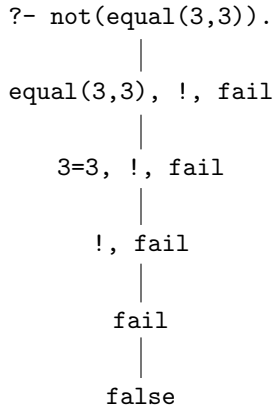

Abb 4.2.1

Nun schauen wir uns eine weitere Anfrage an.

```
?- not(equal(2,3))
```

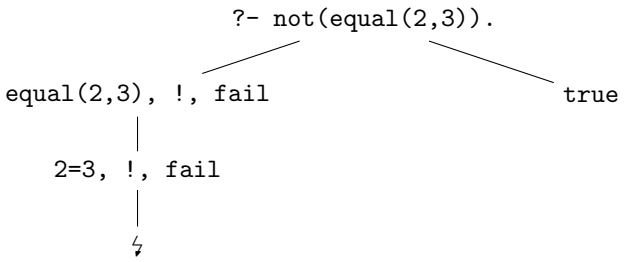

Abb 4.3.1

Hier liegt der einzige Unterschied darin, dass die erste Bedingung in der Klausel des `not` Codes nicht bewiesen werden kann. Dadurch überprüfen wir die zweite Bedingung (den Cut-Operator) also gar nicht. Stattdessen tritt Backtracking ein. Prolog unifiziert unsere Anfrage mit der zweiten Klausel und wertet damit zu `true` aus. Wir beobachten also, dass die Negation ohne den Cut auf diese Weise nicht möglich wäre.

4.3 If-then-else

Auch in Prolog ist es möglich, if-then-else-Abfragen in den Code einzubauen. Dies geschieht aber nicht wie in anderen Programmiersprachen üblich über einen spezifischen Operator, sondern durch eine clevere Verwendung des Cuts oder auch des Negation-Prädikates. Dazu schauen wir uns diese allgemeine Form des Codes an:

```
A :- B, !, C.
A :- D.
```

Wenn wir also eine Anfrage an `A` stellen, versuchen wir `B` zu beweisen. Schaffen wir dies, laufen wir über den Cut-Operator und beweisen danach `C`. Je nachdem, ob `C` nun zu `true` oder `false` ausgewertet wird, wird auch die ganze Anfrage ausgewertet. Würde `B` zu `false` ausgewertet werden, würden wir unsere Anfrage auf die zweite Klausel unifizieren und die Bedingung `D` überprüfen und deren Ergebnis übernehmen. In einfachem Pseudo-Code könnte man also sagen, dass es die folgende Struktur hat:

```
A :- if( B = True) then C
     else D
```

Den Cut-Operator kann man hier auch durch Negation ersetzen, um dieser Pseudo-Code ähnlichen Struktur näher zu kommen.

```
A :- B,C.
A :- not(B), D.
```

Hier evaluieren wir erst `B`. Ist dieses `true`, können wir `C` evaluieren. Ist `B` `false`, geht Prolog in die zweite Klausel. Die erste Bedingung dort, `not(B)`, evaluiert dann zu `true` und damit können wir als nächstes `D` evaluieren.

5 Fazit

Zusammenfassend kann man sagen, dass der Cut ein überaus wichtiger Bestandteil der effizienten Programmierung in Prolog ist. Dieser ist jedoch mit Vorsicht zu genießen, da er bei falscher Anwendung zu veränderten Funktionalitäten in einem Programm führen kann. Die veränderten Funktionalitäten sind in vielen Fällen vom Programmierer aber auch gewollt. Wir haben darüber hinaus gelernt, dass die aus dem Cut hervorgehende Performanceverbesserung durch das Stoppen des Backtrackings erreicht wird. Abschließend haben wir gesehen, wie auf dem Cut aufbauend einige weitere Strukturen in Prolog implementierbar sind. Unter diesen ist auch die Negation, welche hier, verglichen mit dem intuitiven Verständnis der Negation, anders aufzufassen ist.

Referenzen

[1] Shapiro, Ehud / Sterling, Leon: *The art of Prolog - Advanced programming techniques*, 2. Aufl, MIT Press, 1999.

[2] Clocksin, William F. / Mellish, Christopher S.: *Programming in Prolog*, 4. Aufl., Springer, 1994