Cut und Negation

Ivaylo Doychev Johannes Röder

1 Einleitung

In der Praxis unterscheidet man zwischen deklarativen und imperativen Programmiersprachen. Die Mehrheit der heutigen Programmiersprachen sind imperative Programmiersprachen, was bedeutet, dass das Programm aus nacheinander ausgeführten Anweisungen besteht. Die deklarative Programmierung ist hingegen problemorientiert statt maschinenorientiert. Deshalb steht im Vordergrund die Beschreibung des Problems (welches genau berücksichtigt werden soll). Deklarative Programmiersprachen unterteilen sich in funktionale Sprachen, bei denen das Programm generell eine Funktion realisiert, und logische Sprachen.

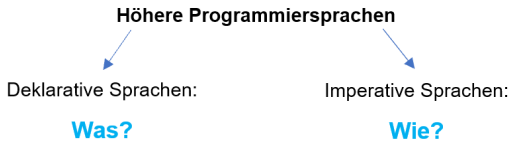

Abbildung 1: Unterteilung der Programmiersprachen

In dieser Ausarbeitung stellen wir das Cut-Prädikat vor, das dem Programmierer die Möglichkeit anbietet, den Backtracking-Mechanismus von Prolog zu kontrollieren und somit Meta-Prädikate wie Negation zu implementieren. Diese Möglichkeit erlaubt uns die Effizienz des Programms zu verbessern, was in der Praxis stark gefordert wird.

2 Grundlagen der Programmiersprache Prolog

2.1 Logische Programmiersprachen

Das Konzept:

- Das Programm besteht aus einer Ansammlung von Fakten, die Aussagen über Objekte und ihren Zusammenhang enthalten, sowie Regeln, die aus bekannten Fakten neue erzeugen.

- Der Programmierer definiert nur die logischen Relationen eines zu lösenden Problems (Wissensdatenbank). Die Wissensbasis dient dazu, Anfragen zu beweisen und Probleme zu lösen → Prinzip des logischen Programmierens.
- Kein Wissen über maschinennahe Details des Computers notwendig.
- Bei dem Beweis einer Anfrage wird die Wissensbasis von oben nach unten bearbeitet. Die Aussagen einer Anfrage werden von links nach rechts bearbeitet und hiermit wird das Prinzip des Backtracking als Grundprinzip der Problemlösung eingeführt.

2.2 Backtracking in Prolog

Bei einer Anfrage an ein Prolog-Programm gibt es zwei Möglichkeiten:

- Ein passender Fakt oder eine passende Regel wird gefunden. Wenn es sich dabei um eine Regel handelt, wird zuerst versucht die jeweiligen Bedingungen dieser Regel zu beweisen.
- Wenn hingegen weder ein passender Fakt noch eine passende Regel gefunden wird, scheitert die Anfrage. Dann wird versucht eine andere Lösung für die vorherige Anfrage zu finden. Die Datenbank wird also ab dem Punkt durchsucht, an dem die vorherige Regel die Suche unterbrochen hat. Dieser Schritt zurück wird Backtracking genannt und lässt sich gut in einem Beweisbaum darstellen.

Um Backtracking nachzuvollziehen betrachten wir das folgende Beispiel:

```
a(X) :- c(X).
a(test).
a(X) :- b(X).
b(X) :- b(b(X)).
c(hallo).
```

Bei der Anfrage: ***?- a(X).*** wird zuerst die erste Regel angewendet und beim Beweis von ***c(X)*** findet Prolog den Fakt: ***c(hallo).*** in der Datenbank. Um weitere Ergebnisse zu finden, wird nun Backtracking betrieben. Zuerst wird für ***c(X)*** eine weitere Lösung gesucht, und nachdem das fehlschlägt, für ***a(X).*** Dies lässt sich in dem Beweisbaum in Abbildung 2 darstellen:
Wie in Abb. 2 zu sehen ist der Beweisbaum nicht endlich. Dieses Problem kann man unter anderem mit einem Cut lösen. Diese Funktion von Prolog wird in den nächsten Abschnitten weiter beleuchtet.

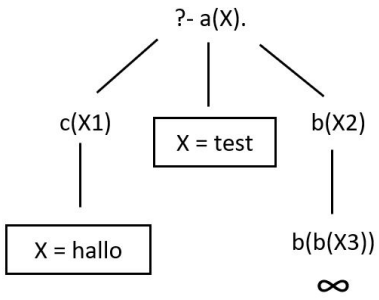

Abbildung 2: Backtrackingbeispiel

3 Cut und Negation

3.1 Der Cut

In diesem Abschnitt werden wir das Cut-Prädikat betrachten, welches dazu dient, das automatische Backtracking in Prolog zu vermeiden. Darüber hinaus ermöglicht der Cut die Definition einer Art von Negation – die sogenannte *Negation as Failure*.

3.1.1 Bezug zum Backtracking

Automatisches Backtracking ist eine der charakteristischsten Eigenschaften von Prolog. Obwohl das in vielen Fälle von Vorteil für den Programmierer und die Problemlösung ist, gibt es manche Fälle, in denen das Backtracking nicht empfehlenswert ist oder sogar unbedingt vermieden werden soll. Hierfür gibt es viele Gründe, wie z.B. unnötige Nicht-Terminierung, Zeit- und Speicheraufwand. Diese werden wir später betrachten.
Betrachten wir nun folgendes Beispielprogramm:

Regel 1: Falls $X < 3$, *dann* $Y = 0$
Regel 2: Falls $X \leq 3$, *dann* $Y = 2$
Regel 3: Falls $X \leq 6$, *dann* $Y = 4$

In Prolog wird das folgendermaßen realisiert:

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

Die Anfrage: “?− f(1,Y), 2 < Y.“ führt zu folgendem Beweisbaum:

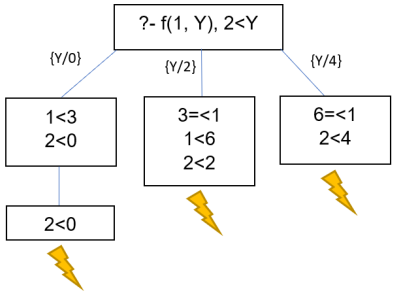

Abbildung 3: Beweisbaum ohne Cut

Das Ergebnis liefert also „false“. Als Grundbeweisprinzip benutzt Prolog die Methode ”von oben nach unten”, deshalb wird erst der linkeste Pfad durchlaufen. Da X mit 1 belegt ist, wird die erste Regel betrachtet und so wird Y mit 0 instanziiert und als Folgerung scheitert das zweite Beweisziel $2 < 0$. Dann passiert unnötiges Backtracking, obwohl die Regeln sich gegenseitig ausschließen, denn Prolog weiß nicht, dass die anderen Belegungen zu keinem Ergebnis führen würden. Um dieses unnötige Backtracking zu verhindern, kommt das Cut-Prädikat ins Spiel, das durch „!“ gekennzeichnet wurde. So erhalten wir das folgende modifizierte Programm:

```
f(X,0) :- X < 3, !.
f(X,0) :- X =< 3, X < 6, !.
f(X,0) :- 6 =< X.
```

Der Effekt des Cuts in der ersten ***f***–Klausel ist, dass man bei einer Anfrage ***f(...)*** nach der erfolgreichen Substitution von ***X*** in der ersten ***f***–Klausel und dem Gelingen des Beweisziels $\boldsymbol{X < 3}$ nicht mehr zurücksetzen kann, um alternative Beweise zu versuchen. So erhalten wir:

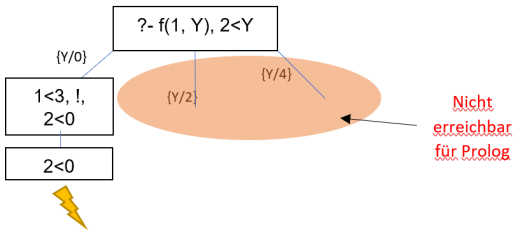

Abbildung 4: Beweisbaum mit Cut

3.1.2 Hauptanwendungen von Cuts

Oftmals können wir mit dem Cut die Effizienz des Programms verbessern. Die Hauptidee ist, dass wir Prolog explizit mitteilen, dass die alternativen Beweisziele zu „fail“ führen würden. Bei sich gegenseitig ausschließenden Regeln wäre z.B. das Backtracking völlig unnötig. Dies können wir durch die Verwendung von Cuts vermeiden. Darüber hinaus kann das Backtracking zu ungewollter Nicht-Terminierung, Zeit- oder Speicheraufwand führen, welche der Programmausführung schadet. Mit Hilfe von Cuts lässt sich das Problem bei vernünftigerer Implementierung lösen.
Außerdem ermöglicht es uns Meta–Prädikate wie die *Negation* zu programmieren.

3.1.3 Green Cuts

In der Praxis bezeichnet man diejenigen Cuts als grün, die nur die Effizienz aber nicht das Ergebnis oder das Terminierungsverhalten des Programms beeinflussen. Also bedeutet das, dass beim Entfernen des Cuts aus dem Programmcode das Programm immer noch dieselben Ergebnisse liefert. Um das Konzept nachzuvollziehen, betrachten wir folgendes spezifisches Beispiel zur Bestimmung des Minimums zweier gegebenen Elemente in Prolog:

```
minimum(X,Y,X) :- X=<Y. <----------------> minimum(X,Y,X) :- X=<Y,! .
minimum(X,Y,Y) :- X>Y.  <----------------> minimum(X,Y,X) :- X>Y, !.
```

Die Klauseln auf der linken Seite schließen sich gegenseitig aus, so wäre es empfehlenswert einen Cut hinzuzufügen, um das unnötige Backtracking zu verhindern. Somit steigt die Effizienz des Programms, ohne das Endergebnis zu verändern.

3.1.4 Red Cuts

Während die so genannten Green Cuts dafür genutzt werden, um unnötige Zweige des Beweisbaumes abzuschneiden und somit das Programm effektiver machen, verändern Red Cuts die Bedeutung eines Programms und somit die Lösungen, die bewiesen werden.
Am Beispiel einer Einkaufsliste lässt sich ein red cut erklären:

```
einkauf(X,[X,XS]):- !.
einkauf(X,[Y,XS]):- einkauf(X,XS).
```

Wenn man nun einen Artikel in der Einkaufsliste suchen möchte, verhindert der Cut, dass nach dem Auffinden des Artikels weiter gesucht wird. Der orange markierte Teil des Beweisbaumes in Abbildung 5 wird somit abgeschnitten. Diese Implementierung bringt jedoch einen Mangel mit sich. Beim Ausgeben der kompletten Liste (***?- einkauf(X,[wurst, kaese, salat].***) verhindert der Cut, dass mehr als ein Ergebnis ausgegeben wird. Das einzige Ergebnis des Beweisbaumes in Abbildung 6 ist also $\boldsymbol{X = wurst}$. Alle anderen Ergebnisse können nicht erreicht werden. Es handelt sich somit um einen Red Cut.

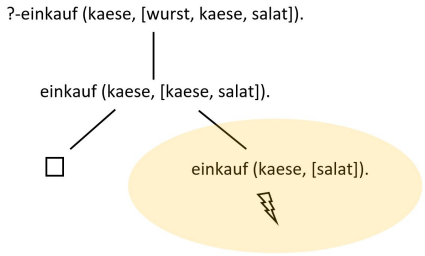

Abbildung 5: Beweisbaum 1

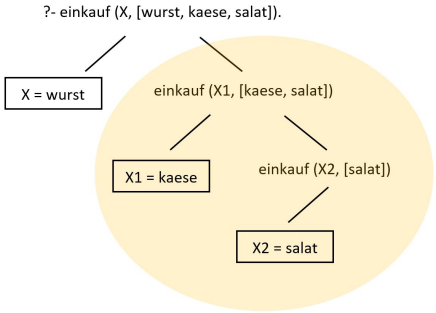

Abbildung 6: Beweisbaum 2

3.1.5 Gründe für Cuts

Das Hauptziel bei der Verwendung von Cuts sollte keinesfalls die Bedeutungsänderung des Programms, sondern vielmehr die Effizienzoptimierung des selbigen sein. Hiermit ist gemeint, das Programm hinsichtlich der Ressourcen des Computers, Rechenzeit und Speicherplatz, zu verbessern.

Speicherplatz

Normalerweise nutzen Prologimplementierungen zwei Speicher: den Stack und den Heap. Der Stack (local stack) kontrolliert den generellen Kontrollfluss, während der Heap (global stack) die Datenstrukturen konstruiert, die während der Programmausführung benötigt werden. Wird nun eine Anfrage an Prolog gestellt, wird dem Stack ein so genannten Stack Frame hinzugefügt. Je mehr Anfragen gestellt werden, desto mehr Speicher benötigt der Stack. Dies ist vor allem bei rekursiven Anfragen zu berücksichtigen.
Um nun die Speichernutzung eines Prolog Programmes effizient zu gestalten, sollte vor allem das Wachstum des Stacks reguliert werden, denn zu viele Anfragen können auch dazu

führen, dass das Programm abbricht (Stack Overflow). Dies passiert vor allem bei rekursiven, nicht-terminierenden Programmen. Unter anderem kann durch Cuts eine ökonomische Speichernutzung herbeigeführt werden.

Zeit

Die Zeit, die ein Prolog-Programm benötigt, ist direkt abhängig von der Zahl der Substitutionen, die es vollziehen muss. Um diese zu optimieren, gibt es mehrere Möglichkeiten. Je nach Situation ist zum Beispiel die Implementierung eines effizienteren Algorithmus sinnvoll.
Neben der Verwendung anderer Algorithmen kann natürlich auch die Implementierung eine große Rolle spielen. Um diese zu verbessern, sollten folgende Punkte beachtet werden:

- Sinnvolle Anordnung der Regeln und Fakten
- Vermeidung von Endlosschleifen, u.a. durch Cuts

Generell sind Cuts nur nötig, wenn Prolog durch Indexierung nicht selbst bestimmen kann, ob ein Beweis terminiert oder nicht. Beim Anwenden der Cuts sind dann die folgenden Prinzipien zu beachten:

- Cuts sollten so lokal wie nur möglich sein
- Ein Cut sollte platziert werden, sobald die korrekte Antwort ausgewählt wurde
- Es sollten erst dann Cuts zu einem Programm hinzugefügt werden, wenn das Programm korrekt funktioniert. Die Bedeutung eines Programms sollte sich durch den Cut nicht ändern.

3.1.6 Probleme von Cuts

Bisher wurde schon teilweise gezeigt, wie ein Cut sehr schnell und ungewollt die Bedeutung eines Programmes verändern kann. Einerseits kann ein Cut natürlich harmlos und sogar vorteilhaft sein, gleichzeitig kann derselbe Cut ein ungewolltes Verhalten des Programms hervorrufen, wenn die Regel des Programms anders verwendet wird.
Betrachten wir das folgende Beispiel:

```
preis(wurst, 4) :- !.
preis(kaese, 5) :- !.
preis(salat, 1) :- !.
preis(X, 3).
```

Wir ordnen nun jedem Artikel unserer Einkaufsliste einen Preis zu. Alle Artikel, die sich nicht in der Liste befinden, sollen den Preis 3 haben. Diese Nutzung von Cuts hat gleich mehrere Probleme:

- Ohne die Cuts würde Anfragen wie ***?- preis(wurst,X).*** zwei Ergebnisse haben: $X = 4$ und $X = 3$. Um dies zu verhindern wurden die Cuts eingeführt. Dies verändert jedoch die Bedeutung des Programms.
- Wenn nun die Anfragestrategie geändert wird und zum Beispiel gefragt wird, ob der Preis von Wurst 3 ist ***?- preis(wurst,3).*** antwortet das Programm mit true, obwohl ***wurst*** eigentlich den Preis 4 hat.

Somit sollten Cuts nur genutzt werden, wenn man sich als Programmierer im Klaren über die Anfragen an das Programm ist. Wenn die Anfragestrategie geändert wird, können Cuts ungewollte Ergebnisse herbeiführen.

3.2 Negation

Eine Negation kann in Prolog mit Hilfe des mitgelieferten Prädikats ***not*** umgesetzt werden. Somit kann das Prädikat vom Programmierer sofort genutzt werden, ohne es vorher implementieren zu müssen. ***not X*** kann genau dann bewiesen werden, wenn ***X*** fehlschlägt. Für mathematische Operatoren gibt es Kurzschreibweisen: ***not(N = 1)*** kann durch ***n\=1*** und ***not(N ≤ 1)*** durch ***N > 1*** ersetzt werden. Die Definition von ***not*** ist die folgende:

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

Wenn ***X*** ausgewertet werden kann, schlägt das Prädikat ***not X*** nach dem Cut fehl. Durch den Cut kommt es auch nicht zur Auswertung der zweiten Regel. Wenn ***X*** jedoch nicht ausgewertet werden kann, wird die zweite Regel angewendet und ***not X*** ist bewiesen. Hierbei ist das Prädikat ***fail*** ein von Prolog Vorgegebenes, das eine Anfrage immer fehlschlagen lässt.

3.2.1 Terminierung einer Negation

Dies führt jedoch zu einem potentiellen Problem: Bisher änderte die Reihenfolge der Argumente nur die Reihenfolge der möglich Lösungen im Beweisbaum. Nun kann sich jedoch auch die Bedeutung des Programms mit den Argumenten ändern. Ob ein Ziel ***G*** terminiert hängt von ***G*** ab. In unserem Beispiel terminiert ***not G*** immer dann, wenn ***G*** auch terminiert. Wenn ***G*** jedoch nicht terminiert, dann könnte ***not G*** terminieren. Dies ist aber nicht sicher.
Ein Beispiel hierfür ist das folgende Programm:

```
p(s(X)) :- p(X).
q(a).
```

Die Anfrage ***?- not((p(X),q(X)))*.** terminiert nicht, da ***?-p(X).*** nicht terminiert. Angenommen wir ändern die Reihenfolge der Argumente zu ***?- not(q(X),p(X))*.**, terminiert die

Anfrage und liefert insgesamt das Ergebnis ***true*** zurück.
Zusammenfassend ist bei der Verwendung von Negationen in dieser Form zu sagen, dass Anfragen, die Variablen enthalten, nicht zwingend korrekt funktionieren. Somit sollte man als Programmierer darauf achten negierte Anfragen möglichst ohne Variablen aufzurufen.

3.2.2 Negation oder Cut: Vergleich

Generell ist ein guter Programmierstil, Cuts durch Negationen zu ersetzen, da Cuts ein Programm unübersichtlicher machen, während Negationen die Lesbarkeit erhöhen. Dies kann man an den folgenden Programmen sehen:

A:- B,C. **Statt** *A:- B, !, C.*
A:- not(B),D. **Statt** *A:-D.*

Eigentlich ist der Code mit Negation besser verständlich, aber in dem obigen Beispiel mit der Negation würde Prolog versuchen, die Regel ***B*** zweimal zu beweisen. Bei komplizierteren Programmen wäre das zu ineffizient, weshalb in diesem Fall der rote Cut bevorzugt wird.

Die vernünftigere Wahl?

Im Allgemeinen gibt es keine universelle Antwort, da die Verwendung von Cut oder Negation sehr vom jeweiligen Fall abhängig ist. In der Praxis verwendet man in einem Projekt oft die Kombination von Cut und Negation, welche in manchen Fällen Probleme mit sich bringt.

4 Hintergrund und Zusammenfassung

In der Literatur wird der Cut teilweise als eine der einflussreichsten Neuerungen in Prolog überhaupt bezeichnet. Dementsprechend soll hier auch die Geschichte des Cut kurz Erwähnung finden.
Eingeführt wurde der Cut 1973 in Marseilles Prolog. 9 Jahre später wurde 1982 die Terminologie der Green Cuts und Red Cuts eingeführt. Dabei wurde versucht, zwischen legitimer und illegitimer Nutzung von Cuts zu unterscheiden. Mit der Zeit kamen immer neue Erweiterungen und Anwendungen des Cuts hinzu, die aber diesen Rahmen sprengen würden.

Zusammenfassend ist zu sagen, dass der Cut ein mächtiges und wichtiges Werkzeug ist, um Prolog Programme vor allem effizienter zu machen. Hierbei muss man jedoch Vorsicht walten lassen, da ein Cut schnell ungewünschte Probleme herbeiführen kann. Deswegen gibt es ein paar Regeln, die wir genannt haben, an die man sich halten sollte. Cuts können auch durch Negationen ersetzt werden, was oftmals guter Programmierstil ist, aber auch wiederum die Effizienz des Programmes verschlechtern kann.

Literatur

[1] William F Clocksin and Christopher S Mellish. *Programming in Prolog*. Springer Science & Business Media, fourth edition edition, 2003.

[2] Jürgen Giesl. Logikprogrammierung. Technical report, Lehr- und Forschungsgebiet Informatik 2 RWTH Aachen, 2015.

[3] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.