Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Anordnung der Literale und Effizienz

Bsp.: Berechnung für `son(X,lot)`

- 1. Möglichkeit: `son(X,Y) :- male(X),parent(Y,X).`
  $\longrightarrow$ man betrachtet die Männer nacheinander und prüft, ob sie Kinder von `lot` sind.

- 2. Möglichkeit: `son(X,Y) :- parent(Y,X),male(X).`
  $\longrightarrow$ man betrachtet die Kinder von `lot` nacheinander und prüft, ob sie Männer sind.

Anordnung der Literale und Effizienz

Bsp.: Berechnung für `son(X,lot)`

- 1. Möglichkeit: `son(X,Y) :- male(X),parent(Y,X).`
  $\longrightarrow$ man betrachtet die Männer nacheinander und prüft, ob sie Kinder von `lot` sind.

- 2. Möglichkeit: `son(X,Y) :- parent(Y,X),male(X).`
  $\longrightarrow$ man betrachtet die Kinder von `lot` nacheinander und prüft, ob sie Männer sind.

$\Longrightarrow$ Die zweite Anordnung ist günstiger für die Anfrage `son(X,lot)` aber die erste ist besser für `son(sarah,X)`

$\Longrightarrow$ Die optimale Anordnung hängt von der beabsichtigten Benutzung ab.

Anordnung der Literale und Effizienz

$\Longrightarrow$ Heuristik: Literale deren Ableitung effizient ist (z.B. arithmetische Teste), sollten möglichst links, vor anderen Literalen (insbesondere vor rekursiven) Atomen stehen.

Bsp.: Eine Prozedur `partition(Liste,Pivot,Kleinere,Groessere)` kann benutzt werden, um eine Liste in zwei Listen der Elemente, die kleiner bzw. größer als ein Pivot sind. (☞ Quicksort).

- Eine Klausel, die die Prozedur definiert könnte so aussehen:

```
partition([X|Xs],Y,[X|Ks],Gs) :- X<=Y,partition(Xs,Y,Ks,Gs)
```

- Diese führt i.A. zu effizienteren Berechnungen als:

```
partition([X|Xs],Y,[X|Ks],Gs) :- partition(Xs,Y,Ks,Gs),X<=Y
```

Anordnung der Literale und Terminierung

Die Anordnung der Literale kann über Terminierung entscheidend sein.

```
quicksort([X|Xs],Ys) :-
        partition(Xs,X,Kleinere,Groessere),
        quicksort(Kleinere,Ls),
        quicksort(Groessere,Bs),
        append(Ls,[X|Bs],Ys).
```

$\Longrightarrow$ terminiert, weil die rekursive Sortierung auf die kleineren Listen `Kleinere` bzw. `Groessere` angewendet wird.

```
quicksort([X|Xs],Ys) :-
        quicksort(Kleinere,Ls),
        quicksort(Groessere,Bs),
        partition(Xs,X,Kleinere,Groessere),
        append(Ls,[X|Bs],Ys).
```

$\Longrightarrow$ terminiert nicht.

Redundante Lösungen

Prolog gibt für jede erfolgreiche Ableitung eine Antwort aus. U.U. (wenn mehrere Klauseln für den selben Fall zuständig sind) kann eine Antwort sich wiederholen, z.B.:

```
append([],Ys,Ys).
append([X],Ys,[X|Ys]).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

```
?- append([1],[2,3],X).
X = [1, 2, 3] ;
X = [1, 2, 3] ;
No
```

Systemprädikate

- **Systemprädikate** (*builtin Prädikate, bips*) sind Prädikate, die vom implementierenden System direkt unterstützt werden, statt mit Hilfe von Klauseln definiert zu sein.

$\Longrightarrow$ Effizienz, dafür Eingeschränkungen bezüglich ihrer Benutzung.

- **Arithmetische Systemprädikate** liefern Zugang zur effizienten, maschinenunterstützten arithmetischen Funktionalität.

Auswertung arithmetischer Ausdrücke: das Prädikat `is`

Zur Evaluierung eines arithmetischen Ausdrucks benutzt man das infixierte Prädikat `is(Wert, Ausdruck)` d.h.: `Wert is Ausdruck`

- Prolog-Interpretierung des Zieles:
  1. Wenn `Ausdruck` unter der aktuellen Variablensubstitution zu einem Wert v ausgewertet werden kann, liefert die Unifikation von v und `Wert` das Ergebnis der Ableitung des Zieles.
  2. Sonst gibt es einen Laufzeitfehler.

- Beispiele:

`X is 1+2`	Antwort: `X=3`.
`3 is 1+2`	Antwort: `yes`.
`1+2 is 1+2`	Antwort: `no`. (Grund: `1+2` und 3 unifizieren nicht.)

Auswertung arithmetischer Ausdrücke: das Prädikat `is`

Gründe warum ein `Ausdruck` in `Wert is Ausdruck` nicht auswertbar sein könnte:

- `Ausdruck` ist kein arithmetischer Ausdruck, z.B. `1+x`
  $\Longrightarrow$ **Das Ziel scheitert.** (*failure*)

- `Ausdruck` benutzt Variablen, die bei der Auswertung (noch) nicht belegt sind, z.B. `1+Y`, wenn noch keine Substitution von `Y` im Laufe der aktuellen Ableitung vorliegt.
  $\Longrightarrow$ **Laufzeitfehler** (*error condition*)

Das Prädikat `is`

- Vorsicht: `is` dient nicht der Zuweisung eines Wertes an eine Variable.
- X is X+1 schlägt fehl oder führt zu einem Laufzeitfehler – immer.

Arithmetische Vergleiche

- `1+2 <= 6-3`
  - Linke Seite wird ausgewertet $\longrightarrow$ 3;
  - Rechte Seite wird ausgewertet $\longrightarrow$ 3;
  - 3 und 3 unifizieren $\longrightarrow$ Antwort `yes`.
- Andere Vergleichsoperatoren: `>=`,`<`, `>`, `=:=` (Gleichheit), `=/=` (Ungleichheit).

Arithmetik in Prolog: Beispiel

```
factorial(N,F) :-
            N>0, N1 is N-1, factorial(N1,F1), F is N*F1.
factorial(0,1).
```

```
?- factorial(3,X).
X = 6 ;
No

?- factorial(X,6).
ERROR: Arguments are not sufficiently instantiated
```

Arithmetik in Prolog: Beispiel

- Alternative Implementierung der Fakultätsfunktion:

```
factorial(N,F) :- factorial(0,N,1,F).
factorial(I,N,T,F) :-
        I < N, I1 is I+1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

- Welche Variante is vorzuziehen? (☞ Übung)

Explizite Kontrolle der Zielauswertung

- Prolog stellt ein Systemprädikat, die das Backtracking bei der Suche von Prolog steuern kann: *cut*, geschrieben !.

- Der Sinn eines **cut** ist, den Suchaufwand für eine Berechnung zu reduzieren, indem man einen Zweig des Suchbaumes abschneidet.
  - green cuts: schneiden Zweige ab, die keine Lösungen enthalten $\Longrightarrow$ erhöhen die Effizienz;
  - red cuts: schneiden Zweige ab, die Lösungen enthalten.

Cuts in Prologprogrammen

- Da cuts die Bedeutung eines Programms von der prozeduralen Interpretierung zusätzlich abhängig machen, verletzten sie die strebenswerte Deklarativität.
  - green cuts: nützlich als Kompromiss zwischen Effizienz und Deklarativität.
  - red cuts: eher unerwünscht.

Cuts: Bedeutung

Cut ist ein nullstelliges Prädikat, das **immer erfüllt** ist. Wird das Cut im Laufe der Ableitung eines aktuellen Zieles A' mit Hilfe einer Klausel

$$A \leftarrow A_1, \ldots, A_k, !, A_{k+1}, \ldots, A_n$$

erfüllt (wobei A und A' unifizieren), so werden Alternative zur Erfüllung von A, A_1, $\ldots$, A_k im Laufe der aktuellen Ableitung von A' ausgeschlossen. D.h.:

- alternative Klauseln, deren Kopf mit A' unifizieren werden ignoriert;
- Wenn die Ableitung von A_i mit $i \geq k+1$ im Laufe der weiteren Ableitung von A' fehlschlägt, werden im Laufe des Backtracking alternative Ableitungen nur soweit zurückverfolgt bis zum !.

Green Cuts

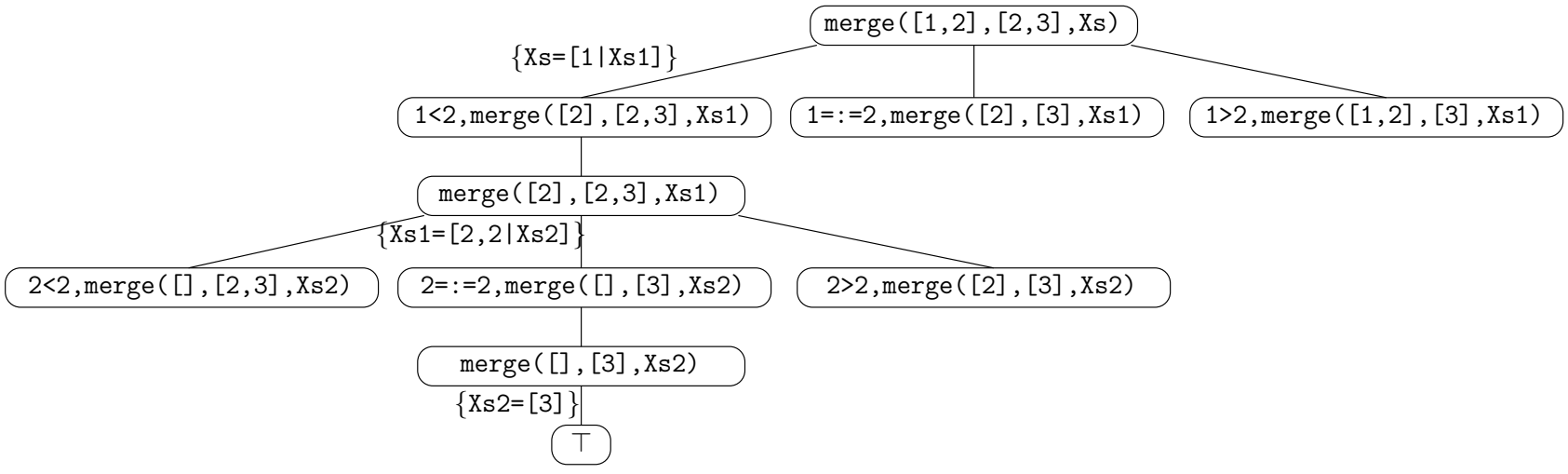
Klauseln zum Mischen zweier geordneten Listen:

```
merge([X|Xs],[Y|Ys],[X|Zs]) :-
                          X < Y, merge(Xs,[Y|Ys],Zs).      (1)

merge([X|Xs],[Y|Ys],[X,Y|Zs]) :-
                          X =:= Y, merge(Xs,Ys,Zs).        (2)

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
                          X > Y, merge([X|Xs],Ys,Zs).      (3)

merge([],[Y|Ys],[Y|Ys]).                                   (4)
merge(Xs,[],Xs).                                           (5)
```

- Das Programm ist deterministisch: es gibt für jedes Ziel höchstens eine Klausel, die zur erfolgreichen Ableitung des Zieles führt;

- Ob die Auswahl einer der Klauseln (1), (2), (3) zum Erfolg führt, hängt ausschließlich von den Testen `X<Y`, `X=:= Y` bzw. `X > Y`.

Green Cuts

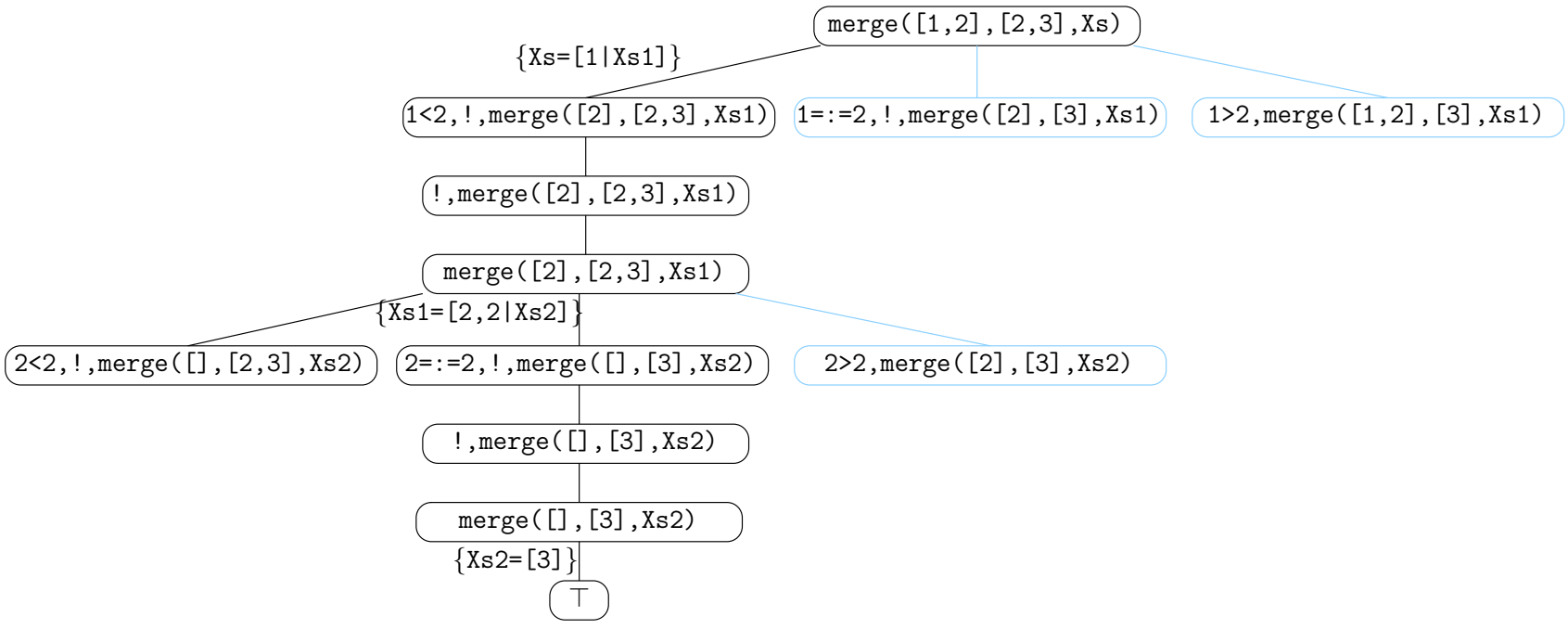

merge([1,2],[2,3],Xs)
{Xs=[1|Xs1]}
1<2,merge([2],[2,3],Xs1)
1=:=2,merge([2],[3],Xs1)
1>2,merge([1,2],[3],Xs1)
merge([2],[2,3],Xs1)
{Xs1=[2,2|Xs2]}
2<2,merge([],[2,3],Xs2)
2=:=2,merge([],[3],Xs2)
2>2,merge([2],[3],Xs2)
merge([],[3],Xs2)
{Xs2=[3]}
⊤

Green Cuts

Wenn (1) zur Erfüllung eines Zieles gewählt wird, braucht man nach dem Test `X =< Y` keine weitere Klauseln betrachten. Ähnliches gilt für den Test `X =:= Y` in (2). Zur Vermeidung der Suche unnötiger Ableitungen kann man hier Cuts einsetzen:

```
merge([X|Xs],[Y|Ys],[X|Zs]) :-
                        X < Y, !, merge(Xs,[Y|Ys],Zs).     (1)

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
                        X =:= Y, !, merge([X|Xs],Ys,Zs).   (2)

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
                        X > Y, merge([X|Xs],Ys,Zs).        (3)

merge([],[Y|Ys],[Y|Ys]).                                   (4)
merge(Xs,[],Xs).                                           (5)
```

Green Cuts


merge([1,2],[2,3],Xs)
{Xs=[1|Xs1]}
1<2,!,merge([2],[2,3],Xs1)
1=:=2,!,merge([2],[3],Xs1)
1>2,merge([1,2],[3],Xs1)
!,merge([2],[2,3],Xs1)
merge([2],[2,3],Xs1)
{Xs1=[2,2|Xs2]}
2<2,!,merge([],[2,3],Xs2)
2=:=2,!,merge([],[3],Xs2)
2>2,merge([2],[3],Xs2)
!,merge([],[3],Xs2)
merge([],[3],Xs2)
{Xs2=[3]}
⊤

Die Prolog-Auswertung

- Die Auswertung eines Zieles in Prolog benötigt einen Stack für den Tiefendurchlauf über den Suchbaum. Darin muss man i.A. beim Absteigen durch die Auswahl einer Klausel für das aktuelle Ziel die übrigen alternativen Klauseln merken, um später die Suche via Backtracking fortsetzen zu können.
  $\Longrightarrow$ Informationen über die jeweils zuletzt gewählten Klauseln (*choice points*) müssen in jedem Kellerrahmen gespeichert werden.

- Insbesondere, in der prozeduralen Auslegung:
  - Eine Klausel $A \leftarrow B_1, \ldots, B_n$ entspricht der Definition einer Prozedur A.
  - Im Unterschied zu prozeduralen Programmiersprachen hat A statt eine, so viele Definitionen, wieviele Klauseln A definieren. Ein Interpreter muss i.A. alle Definitionen der Reihe nach betrachten.

Die Last-call-Optimierung

Last-call-Optimierung: $A \leftarrow B_1, \ldots, B_{n-1}, B_n$

- Benutze für die Auswertung von B_n den Kellerrahmen für die Auswertung von A wieder.

  Notwendige Bedingung: es gibt keine Alternative Berechungen der Ziele A, B_1, $\ldots$, B_{n-1}. Manche Gelegenheiten zur Last-call-Optimierung können automatisch erkannt werden.

- Cuts können solche Optimierungen zusätzlich unterstützen z.B.:

$$A \leftarrow B_1, \ldots, B_{n-1}, !, B_n$$

Die Last-call-Optimierung

Die letzten (rekursiven) Prädikate in den untenstehenden Klauseln eignen sich für die Last-call-Optimierung (Tail-Recursion-Optimierung):

```
merge([X|Xs],[Y|Ys],[X|Zs]) :-
            X < Y, !, merge(Xs,[Y|Ys],Zs).

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
            X =:= Y, !, merge([X|Xs],Ys,Zs).

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
            X > Y, merge([X|Xs],Ys,Zs).

merge([],[Y|Ys],[Y|Ys]).
merge(Xs,[],Xs).
```

Negation in Prolog

Negation in Prolog wird mit Hilfe des vordefinierten Prädikats `not` implementiert, das wie folgt definiert ist:

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

- Ein Feature von Prolog ist, dass Terme benutzt werden können, um beides Programme und Daten zu repräsentieren. Daten können in Programme transformiert werden (und umgekehrt): `call(X)` transformiert `X` in einem Ziel und versucht dieses abzuleiten. Syntaktisch: `call(X)` $\equiv$ `X` **als Ziel**.

- `fail` ist ein Prädikat, das immer scheitert.

Negation in Prolog vs. NaF

- **Das Metavariable Feature** ist eine vereinfachte Syntax, die erlaubt, `call` wegzulassen

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

- `not` ist eine ungenaue Implementierung der Negation durch Scheitern.
  - $not(X)$ ist erfolgreich in der LP-Semantik, wenn alle Pfade in allen Suchbäumen endlich sind und zu Scheiternknoten führen.
  - `not(X)` ist erfolgreich in Prolog, wenn alle Pfade im Prolog-Suchbaum endlich sind und zu Scheiternknoten führen.