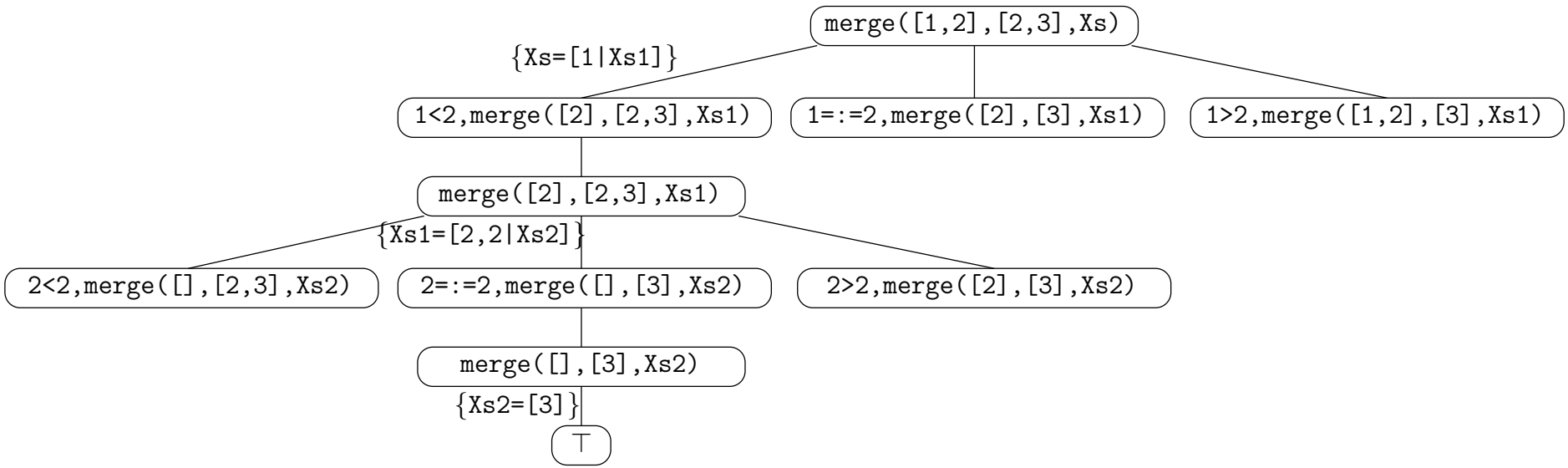
Green Cuts

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

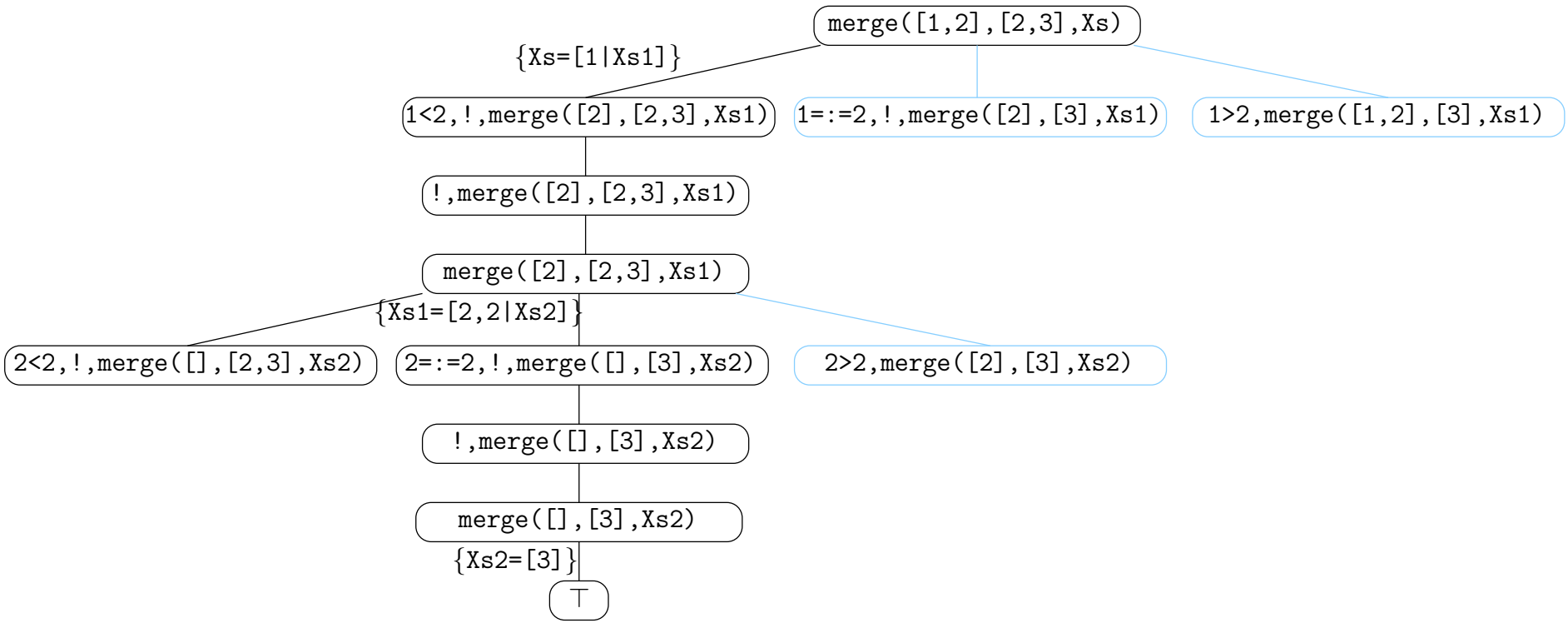

merge([1,2],[2,3],Xs)
{Xs=[1|Xs1]}
1<2,merge([2],[2,3],Xs1)
1=:=2,merge([2],[3],Xs1)
1>2,merge([1,2],[3],Xs1)
merge([2],[2,3],Xs1)
{Xs1=[2,2|Xs2]}
2<2,merge([],[2,3],Xs2)
2=:=2,merge([],[3],Xs2)
2>2,merge([2],[3],Xs2)
merge([],[3],Xs2)
{Xs2=[3]}
⊤

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 Last-call-Optimierung

- 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.
$\Longrightarrow$ Informationen über die jeweils zuletzt gewählten Klauseln (*choice points*) muss in jedem Kellerrahmen gespeichert werden.

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 (rekursive) 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

```
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.

Negation in Prolog vs. NaF

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

- ▷ $not(q(b), p(a))$ ist nicht definiert in der LP-Semantik.
- ▷ `not((q(b),p(a)))` ist erfolgreich in Prolog.

Negation und Ziele mit Variablen

```
braucht_schein(X) :- not(diplom(X)), student(X).
student(piotr).
student(tina).
diplom(tina).
```

Die Anfrage `braucht_schein(X)` scheitert, obwohl die Antwort {`X=piotr`} unter einer Interpretierung von `not` als logische Negation erwartet wird.

$\Longrightarrow$ Prolog erlaubt Negation durch Scheitern nur für zur Laufzeit unter den aktuellen Substitutionen variablenfreie Ziele.

Negation und Ziele mit Variablen

```
braucht_schein(X) :- student(X), not(diplom(X)).
student(piotr).
student(tina).
diplom(tina).
```

$\Longrightarrow$ Antwort {`X=piotr`}.

Der Programmierer muss sicherstellen, dass die Variablen in einem negierten Ziel bei seiner Auswertung belegt sind.

Red Cuts

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

Cuts deren Auftritt in einem Programm die Semantik des Programmes ändern heißen red cuts.

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

Die Anfrage `min(1,2,2)` liefert (unerwünschterweise) *yes*.

Red Cuts

Eine Prozedur zur Entfernung von Elementen aus einer Liste:

```
delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- Z \== X, delete(Xs,Z,Ys).
delete([],X,[]).
```

Red Cuts

Delete mit green Cuts:

```
delete([X|Xs],X,Ys) :- !, delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- Z \== X, !, delete(Xs,Z,Ys).
delete([],X,[]).
```

Delete mit red Cuts:

```
delete([X|Xs],X,Ys) :- !, delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- !, delete(Xs,Z,Ys).
delete([],X,[]).
```

Das Programm mit red Cuts funktioniert richtig, ist dafür bei minimal besserer Effizienz viel unleserlicher geworden.

Selbst-modifizierende Programme

- Prolog erlaubt laufende Programme bei der Laufzeit zu analysieren und zu ändern.
- Das Ziel `clause(Kopf,Rumpf)` bietet Zugang zu den Klauseln des laufenden Programms.
  - ▷ `Kopf` darf keine unbelegte Variable sein.
  - ▷ Die erste Klausel, deren Kopf mit `Kopf` unifiziert wird gefunden und `Rumpf` wird mit dessen Rumpf unifiziert.
  - ▷ Alle Klauseln deren Kopf mit `Kopf` unifizieren werden via Backtracking gefunden.
  - ▷ Fakten haben `true` als ihren Rumpf.

Selbst-modifizierende Programme

Beispiel:

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

Die Antwort zur Anfrage `clause(member(X,Ys),Rumpf)` liefert die Antworten {`Ys=[X|Xs], Rumpf=true`} und {`Ys=[Y|Ys1], Rumpf=member(X,Ys1)`}.

Selbst-modifizierende Programme

Systemprädikate zum Hinzufügen und Entfernen von Klauseln zum (laufenden) Programm:

- `assertz(Klausel)`: fügt `Klausel` als die letzte Klausel der entsprechenden Prozedur. Z.B. `assertz((mammal(X) :- whale(X)))`.
- `asserta(Klausel)`: fügt `Klausel` als die erste Klausel der entsprechenden Prozedur.
- `retract(K)`: entfernt die erste Klausel, die mit `K` unifiziert. Z.B. kann eine Klausel `a :- b,c` mit `retract((a :- X))` entfernt werden.

Selbst-modifizierende Programme

Das dynamische Hinzufugen und Entfernen von Klauseln macht einen Unterschied zwischen “dynamischen” und “statischen” benutzerdefinierten Prädikaten.

- ▷ Statische Prädikate können kompiliert werden $\Longrightarrow$ können effizienter ausgewertet werden.
- ▷ Dynamische Prädikate müssen als solche deklariert werden.
- ▷ Dynamische Prädikate machen den Code von Seiteneffekten abhängig $\Longrightarrow$ weniger deklarativ und leserlich.
- ▷ Allerdings können dynamische Prädikate u.U. die Effizienz unterstützen, z.B. für dynamische Programmierung.

Beispiel: Dynamische Programmierung

Dynamische Programmierung, Idee: partielle Ergebnisse während einer Berechnung speichern, die später wieder benutzt werden können. ($\longrightarrow$ Übung, die Berechnung der Binomialkoeffizienten).

Idee: versuche ein Ziel abzuleiten, und wenn es möglich ist, speichere dieses Ergebnis als einen Fakt und vermeide, dass es später alternative Ableitungen betrachtet werden.

```
lemma(Z) :- Z, asserta((Z :- !)).
```

Beispiel: Dynamische Programmierung

Anwendung: Die Türme aus Hanoi

```
:- op(100, xfx, [to]).
:- dynamic hanoi/5.

hanoi(1,A,B,C,[A to B]).
hanoi(N,A,B,C,Moves) :-
        N > 1, N1 is N -1,
        lemma(hanoi(N1,A,C,B,Ms1)),
        hanoi(N1,C,B,A,Ms2),
        append(Ms1,[A to B|Ms2],Moves).

lemma(P):- P, asserta((P :- !)).

test_hanoi(N,Pegs,Moves) :-
      hanoi(N,A,B,C,Moves), Pegs = [A,B,C].
```

Mehr Prolog

Prolog bietet mehr an, z.B.:

- Prädikate zum Testen und Manipulieren der Struktur der Terme;
- Mehr meta-logische Prädikate z.B. zum Testen des Zustands der Ableitung;
- Mehr extra-logische Prädikate, die Seiteneffekte bei ihrer "Ableitung" haben, z.B für Ein- und Ausgabe oder für die Schnittstelle zum Betriebssystem.
- Es gibt Prolog-Erweiterungen, z.B. für Constraint-Programmierung...