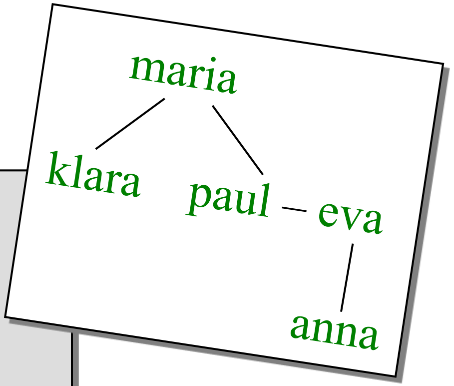



```
istMutterVon(maria, klara).
istMutterVon(maria, paul).
istMutterVon(eva, anna).

istVerheiratetMit(paul, eva).

istGrossmutterVon(G, E) :-
    istMutterVon(G, M),istMutterVon(M, E).
istGrossmutterVon(G, E) :-
    istMutterVon(G, V),istVaterVon(V, E).

istVaterVon(V, K) :-
    istVerheiratetMit(V, M),istMutterVon(M, K).
...
```

Resolution in Prolog (4)

```
?- istGrossmutterVon(maria,anna).
```

```
istGrossmutterVon(G1, E1) :-
    istMutterVon(G1, V1),istVaterVon(V1, E1).
```
$\Rightarrow U_1 = \{G1/maria,\ E1/anna\}$

```
-> istMutterVon(maria,V1),istVaterVon(V1,anna).
```

```
istMutterVon(maria, paul).
```
$\Rightarrow U_2 = \{V1/paul\}$

```
-> istVaterVon(paul,anna).
```

```
istVaterVon(V2, K2) :-
  istVerheiratetMit(V2, M2),istMutterVon(M2, K2).
```
$\Rightarrow U_3 = \{V2/paul,\ K2/anna\}$

```
-> istVerheiratetMit(paul,M2),istMutterVon(M2,anna).
```

```
istVerheiratetMit(paul, eva).
```
$\Rightarrow U_4 = \{M2/eva\}$

```
-> istMutterVon(eva,anna).
```

```
istMutterVon(eva, anna).
```
$\Rightarrow U_5 = \emptyset$

-> □

Resolution in Prolog (5)

```
?- istGrossmutterVon(maria,anna).
```

U_1 = {G1/maria, E1/anna}

U_2 = {V1/paul}

U_3 = {V2/paul,K2/anna}

U_4 = {M2/eva}

$U_5 = \emptyset$

Die Antwort auf die Anfrage ist die Substitution U aller in der Anfrage vorkommenden Variablen, die die Variablen genauso ersetzt wie die Substitution

$U_5 \circ U_4 \circ U_3 \circ U_2 \circ U_1$ = {G1/maria,E1/anna,V1/paul,V2/paul,K2/anna,M2/eva}

Hier: $U = \emptyset$.

Deskriptive Programmierung

Ableitungsbäume

Zur Erinnerung, Motivation für Betrachtung operationeller Semantik …

Wir wollten zum Beispiel verstehen, warum für

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

eine Reihe von Anfragemustern/„Aufrufmodi“ sehr gut funktioniert:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(s(0)))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

aber andere nicht:

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```

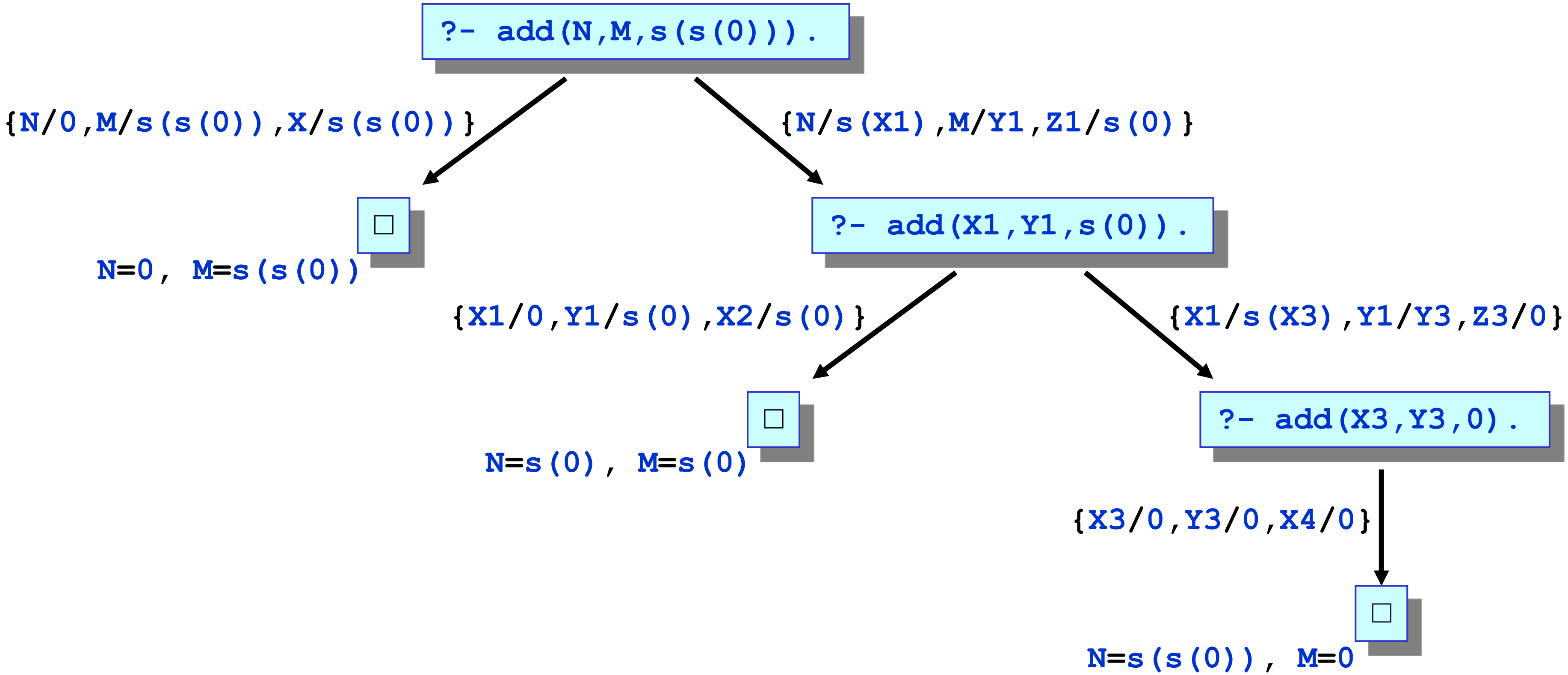
sonst Endlossuche

Explizite Aufzählung von Lösungen

Beginnen wir mit einem einfachen Beispiel für nur die Addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

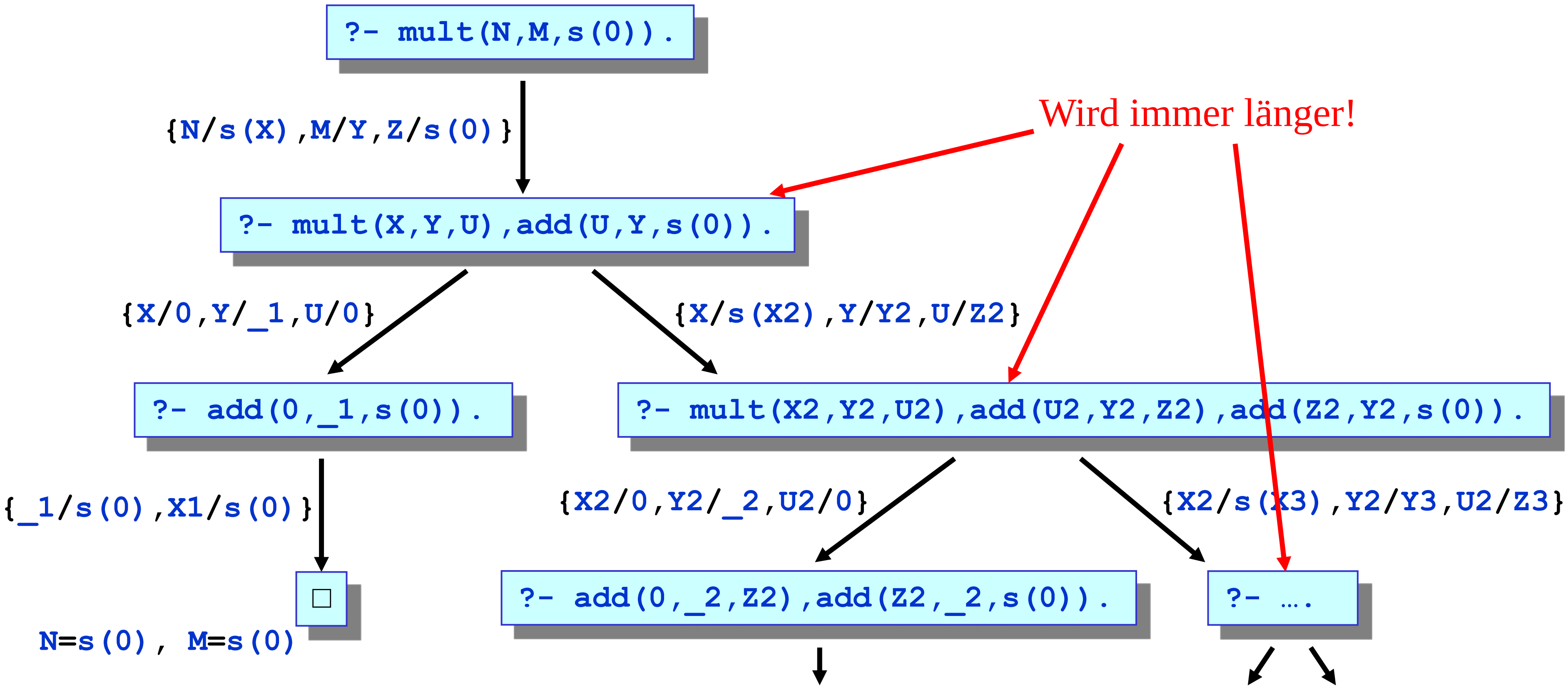
Vollständige Suche:



Ein Beispiel mit endloser Suche

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```



Probehalber Umordnung von Literalen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

⬇

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

```
?- mult(N,M,s(0)).
```

{N/s(X),M/Y,Z/s(0)}

```
?- add(U,Y,s(0)),mult(X,Y,U).
```

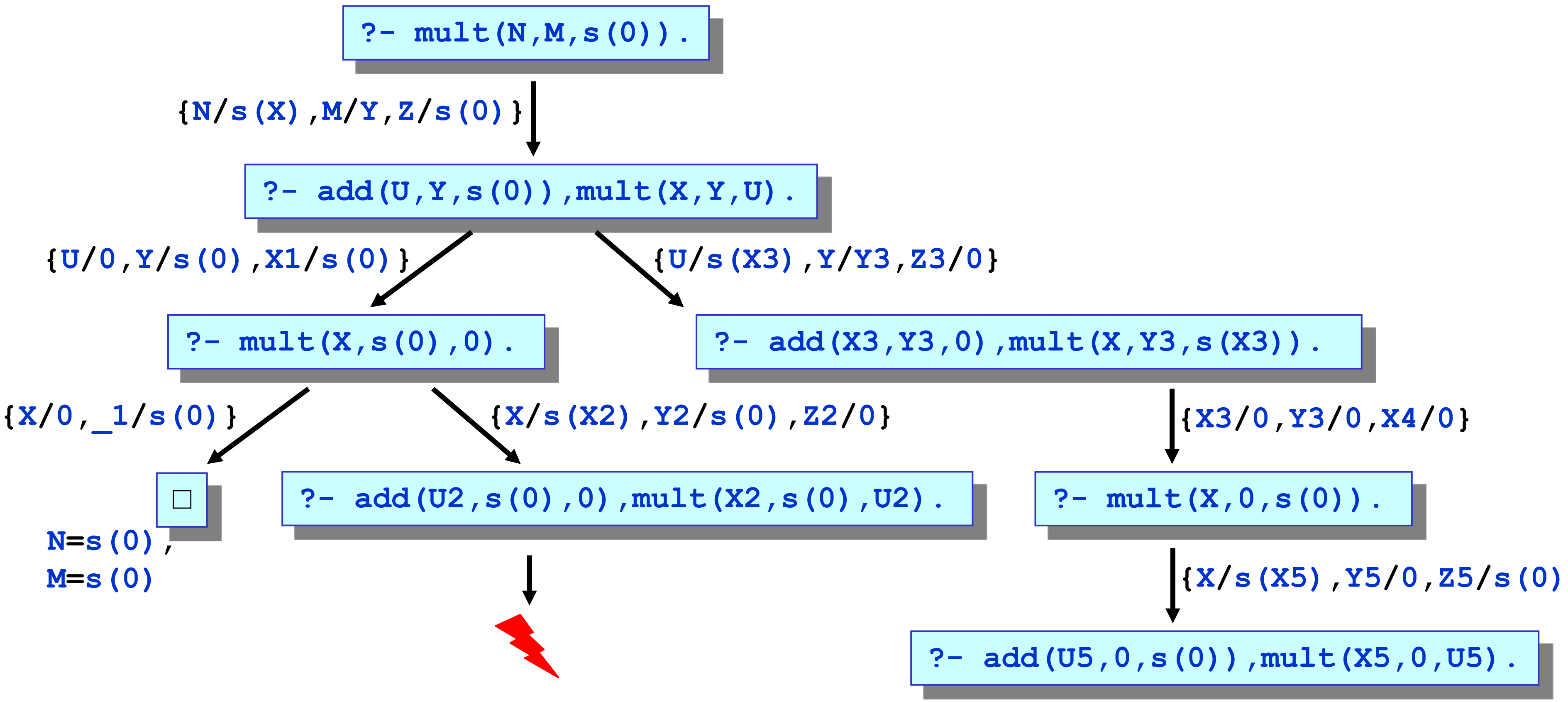
{U/0,Y/s(0),X1/s(0)}

```
?- mult(X,s(0),0).
```

Probehalber Umordnung von Literalen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```



Probehalber Umordnung von Literalen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```

{X3/0,Y3/0,X4/0}

```
?- mult(X,0,s(0)).
```

{X/s(X5),Y5/0,Z5/s(0)}

```
?- add(U5,0,s(0)),mult(X5,0,U5).
```

{U5/s(X6),Y6/0,Z6/0}

```
?- add(X6,0,0),mult(X5,0,s(X6)).
```

{X6/0,X7/0}

```
?- mult(X5,0,s(0)).
```

Sieht nicht gut aus!

Detailliertere Beschreibung der Erzeugung von Ableitungsbäumen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

Eingabe: Anfrage und Programm, zum Beispiel `mult(N,M,s(0))` und:

Ausgabe: Baum, erzeugt durch folgende Schritte:

1. Erzeuge Wurzelknoten mit Anfrage, merke als noch zu bearbeiten.
2. Solange noch zu bearbeitende Knoten vorhanden:
   - wähle linkesten solchen Knoten
   - ermittle alle Regeln, deren Kopf mit dem linkesten Literal im Knoten unifizierbar ist
   - erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
   - sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
   - vermerke jeweils verwendeten Unifikator

```
?- mult(N,M,s(0)).
```

$\downarrow$ `{N/s(X),M/Y,Z/s(0)}`

```
?- add(U,Y,s(0)),mult(X,Y,U).
```

noch zu bearbeiten

Detailliertere Beschreibung der Erzeugung von Ableitungsbäumen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

2. Solange noch zu bearbeitende Knoten vorhanden:
   - wähle linkesten solchen Knoten
   - ermittle alle Regeln, deren Kopf mit dem linkesten Literal im Knoten unifizierbar ist
   - erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
   - sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
   - vermerke jeweils verwendeten Unifikator

```
?- mult(N,M,s(0)).
```

{N/s(X),M/Y,Z/s(0)}

```
?- add(U,Y,s(0)),mult(X,Y,U).
```

{U/0,Y/s(0),X1/s(0)}

{U/s(X3),Y/Y3,Z3/0}

```
?- mult(X,s(0),0).
```

noch zu bearbeiten

```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```

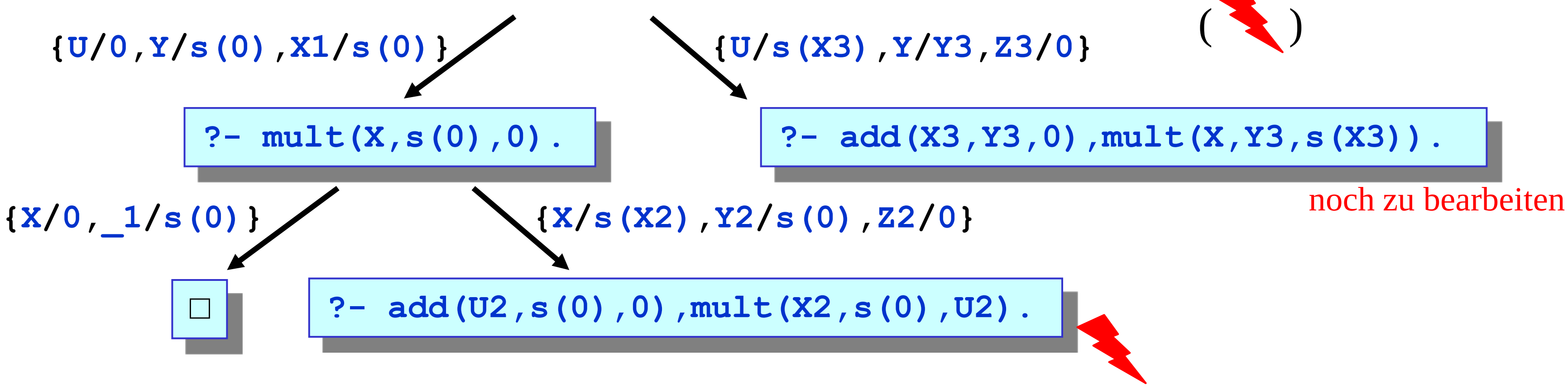
noch zu bearbeiten

Detailliertere Beschreibung der Erzeugung von Ableitungsbäumen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

2. Solange noch zu bearbeitende Knoten vorhanden:
   - wähle linkesten solchen Knoten
   - ermittle alle Regeln, deren Kopf mit dem linkesten Literal im Knoten unifizierbar ist
   - erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
   - sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
   - vermerke jeweils verwendeten Unifikator
   - markiere Knoten als nicht weiter zu bearbeiten, wenn leer (□) oder linkestes Literal mit keinem Regelkopf unifizierbar ( )



Detaillierte Beschreibung der Erzeugung von Ableitungsbäumen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

2. Solange noch zu bearbeitende Knoten vorhanden:
   - wähle linkesten solchen Knoten
   - ermittle alle Regeln, deren Kopf mit dem linkesten Literal im Knoten unifizierbar ist
   - erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
   - sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
   - vermerke jeweils verwendeten Unifikator
   - markiere Knoten als nicht weiter zu bearbeiten, wenn leer oder linkestes Literal mit keinem Regelkopf unifizierbar
   - an Erfolgsknoten, Annotation der Lösung (Komposition der Unifikatoren, angewandt auf relevante Variablen)

`?- mult(X,s(0),0).`

`{X/0,_1/s(0)}` → □ `N=s(0), M=s(0)`

`{X/s(X2),Y2/s(0),Z2/0}` → `?- add(U2,s(0),0),mult(X2,s(0),U2).`

`?- add(X3,Y3,0),mult(X,Y3,s(X3)).`

noch zu bearbeiten

Zurück zum Beispiel: Was tun?

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```

{X3/0,Y3/0,X4/0}

```
?- mult(X,0,s(0)).
```

{X/s(X5),Y5/0,Z5/s(0)}

```
?- add(U5,0,s(0)),mult(X5,0,U5).
```

{U5/s(X6),Y6/0,Z6/0}

```
?- add(X6,0,0),mult(X5,0,s(X6)).
```

{X6/0,X7/0}

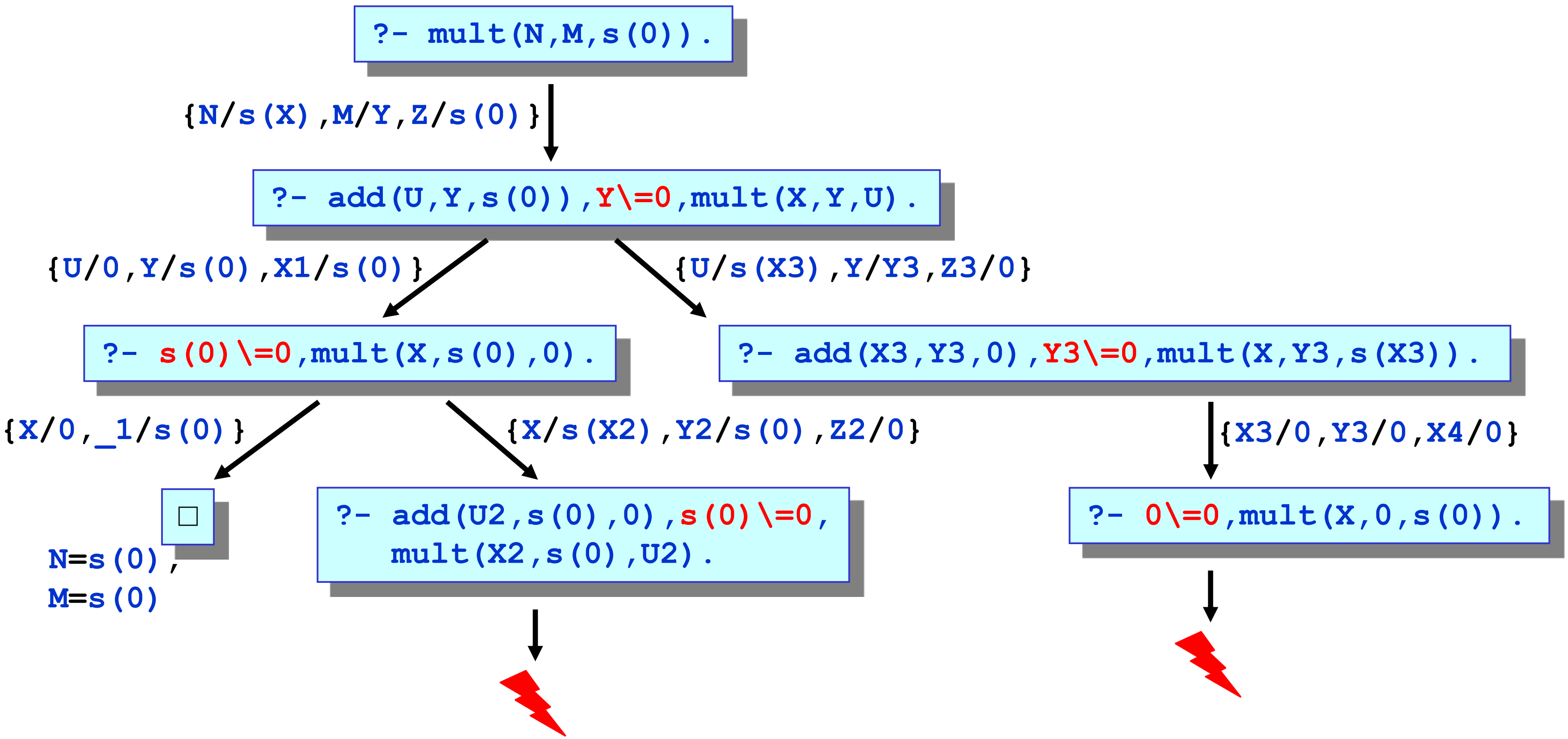
```
?- mult(X5,0,s(0)).
```

Sieht nicht gut aus!

Versuch: Einfügen eines extra Tests

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```



Nur teilweiser Erfolg

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
N = s(s(s(s(0)))),
M = s(0) ;
false.
```

```
?- mult(s(0),0,0).
false.
```

Neue Ergebnisse gefunden, alte Ergebnisse verloren!

Erneute „Reparatur“

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

Jetzt klappt zwar:

```
?- mult(s(0),0,0).
true.
```

Und es funktioniert sogar auch allgemein **mult(?X,?Y,+Z)**.

Aber leider (erst hier bemerkt):

```
?- mult(s(0),s(0),N).
N = s(0) ;
abort
```

sonst Endlossuche

Also geht nicht mehr **mult(+X,+Y,?Z)**.

Eine neue „Unendlichkeitsfalle“

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(s(0),s(0),N).
```

{X/0,Y/s(0),N/Z}

```
?- add(U,s(0),Z),s(0)\=0,mult(0,s(0),U).
```

Sieht nicht gut aus!

{U/0,X1/s(0),Z/s(0)}

```
?- s(0)\=0,mult(0,s(0),0).
```

{_1/s(0)}

□

N=s(0)

{U/s(X2),Y2/s(0),Z/s(Z2)}

```
?- add(X2,s(0),Z2),s(0)\=0,mult(0,s(0),s(X2)).
```

wichtige Beobachtung:
(siehe letzte Vorlesung)

```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

vs.

```
?- add(s(0),U,Z).
Z = s(U).
```

Ausnutzen von Kommutativität

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

wichtige Beobachtung:
(siehe letzte Vorlesung)

```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

vs.

```
?- add(s(0),U,Z).
Z = s(U).
```

Ausnutzen von Kommutativität

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(s(0),s(0),N).
```

{X/0,Y/s(0),N/Z} ↓

```
?- add(s(0),U,Z),s(0)\=0,mult(0,s(0),U).
```

{X1/0,U/Y1,Z/s(Z1)} ↓

```
?- add(0,Y1,Z1),s(0)\=0,mult(0,s(0),Y1).
```

{Y1/X2,Z1/X2} ↓

```
?- s(0)\=0,mult(0,s(0),X2).
```

{_1/s(0),X2/0} ↓

□ N=s(0)

Eine tatsächlich allgemein geeignete Definition

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
N = s(s(s(s(0)))),
M = s(0) ;
false.

?- mult(s(0),s(0),N).
N = s(0).

?- add(X,0,X),not(mult(s(s(_)),s(s(_)),X)).
...
```

Es funktionieren <u>alle</u> Aufrufmodi <u>außer</u> **`mult(?X,?Y,?Z)`**!

Die operationelle Semantik:

- bildet den tatsächlichen Prolog-Suchvorgang ab, mit Backtracking
- benutzt essentiell Unifikation (und Resolution)
- erlaubt Verstehen von Effekten wie Nichttermination
- gibt Einblick in Auswirkungen von Änderungen der Reihenfolge von und innerhalb Regeln

Deskriptive Programmierung

Negation in Prolog

Negation (1)

- Der logischen Programmierung liegt zunächst eine <u>positive Logik</u> zugrunde.

  Ein Literal ist beweisbar, wenn es (ggfs. über mehrere Schritte) auf die Beweisbarkeit unmittelbarer Tatsachen zurückgeführt werden kann.

- Prolog bietet aber auch die Möglichkeit, **Negation** zu verwenden.
  - Diese ist allerdings nur bedingt mit der erwartbaren logischen Bedeutung vereinbar.
  - `\+ Goal`, bzw. `not(Goal)`, ist beweisbar gdw. `Goal` nicht beweisbar ist.

<u>Beispiel</u>: **`\+ member(4,[2,3])`** ist beweisbar, da **`member(4,[2,3])`** nicht beweisbar ist, d.h. es existiert ein „endlicher Misserfolgsbaum“.

<u>Vorsicht</u>:

```
?- member(X,[2,3]).            ⇒ X = 2; X = 3.
?- \+ member(X,[2,3]).         ⇒ false.
?- \+ \+ member(X,[2,3]).      ⇒ true.
```

(Negation führt nicht zur Bindung von Variablen.)

Negation (2)

- Warum „endlicher Misserfolgsbaum“?
  - Wir können nicht allgemein zeigen, dass aus den Regeln eines Programms eine bestimmte negative Aussage folgt.
  - Wir können lediglich zeigen, dass wir eine bestimmte <u>positive</u> Aussage <u>nicht</u> folgern können. (Negation as Failure)
  - Dabei bedeutet „zeigen“, einen Beweis zu suchen und zu scheitern.
  - Dass wir wirklich notgedrungen scheitern, lässt sich nur mit Sicherheit sagen, wenn der Suchraum endlich ist.

- Zu Grunde liegende Annahme:

Closed World Assumption

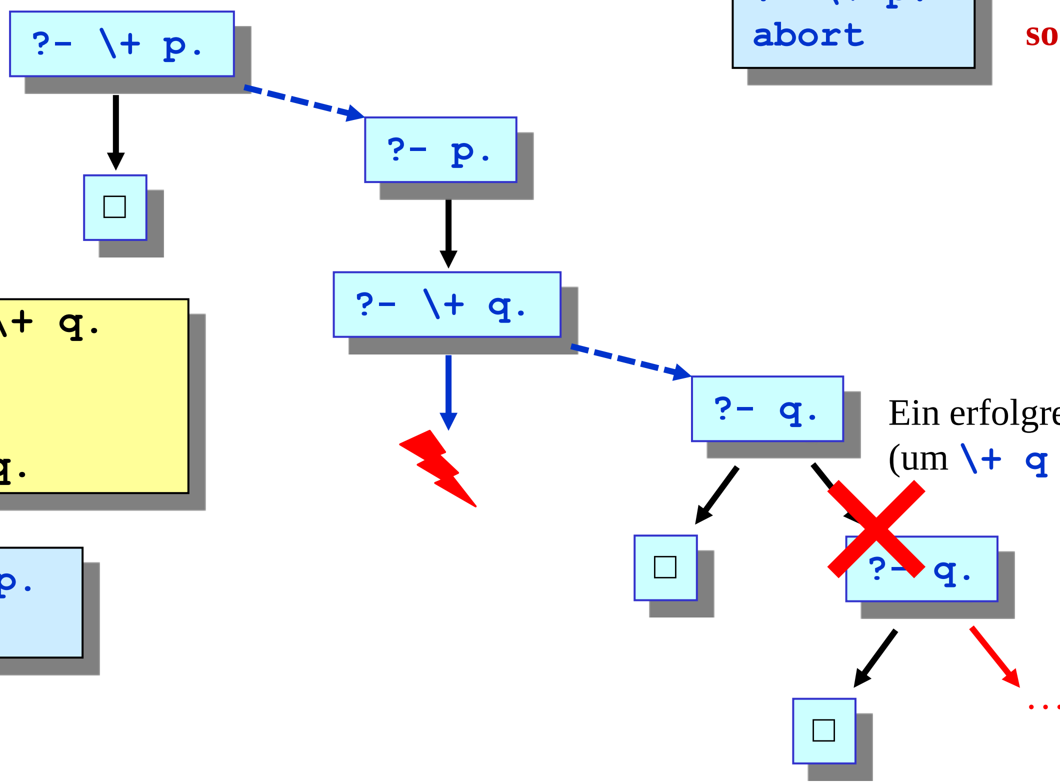
Negation (3)

<u>Beispiele:</u>

```
p :- \+ q.

q.
q :- q.
```

```
?- \+ p.
true.
```

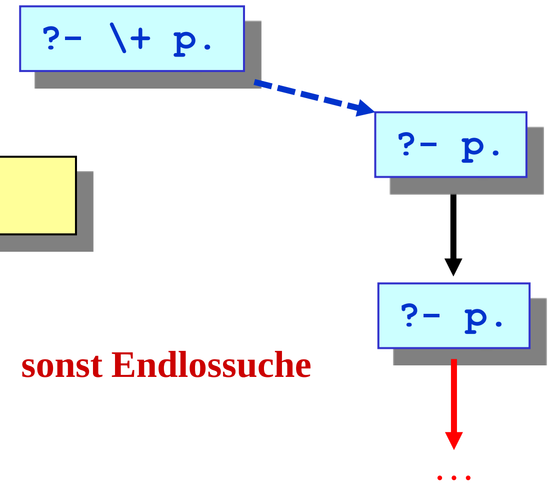


Ein erfolgreicher Zweig (für `q`) genügt (um `\+ q` scheitern zu lassen).

```
p :- p.
```

```
?- \+ p.
abort
```

sonst Endlossuche



Negation (4)

Beispiele mit Variablen:

```
human(marcellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- human(X), \+ married(X,Y).
```

```
?- single(X).
X = marcellus.

?- single(marcellus).
true.

?- single(vincent).
false.
```

```
human(marcellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- \+ married(X,Y), human(X).
```

```
?- single(X).
false.

?- single(marcellus).
true.

?- single(vincent).
false.
```

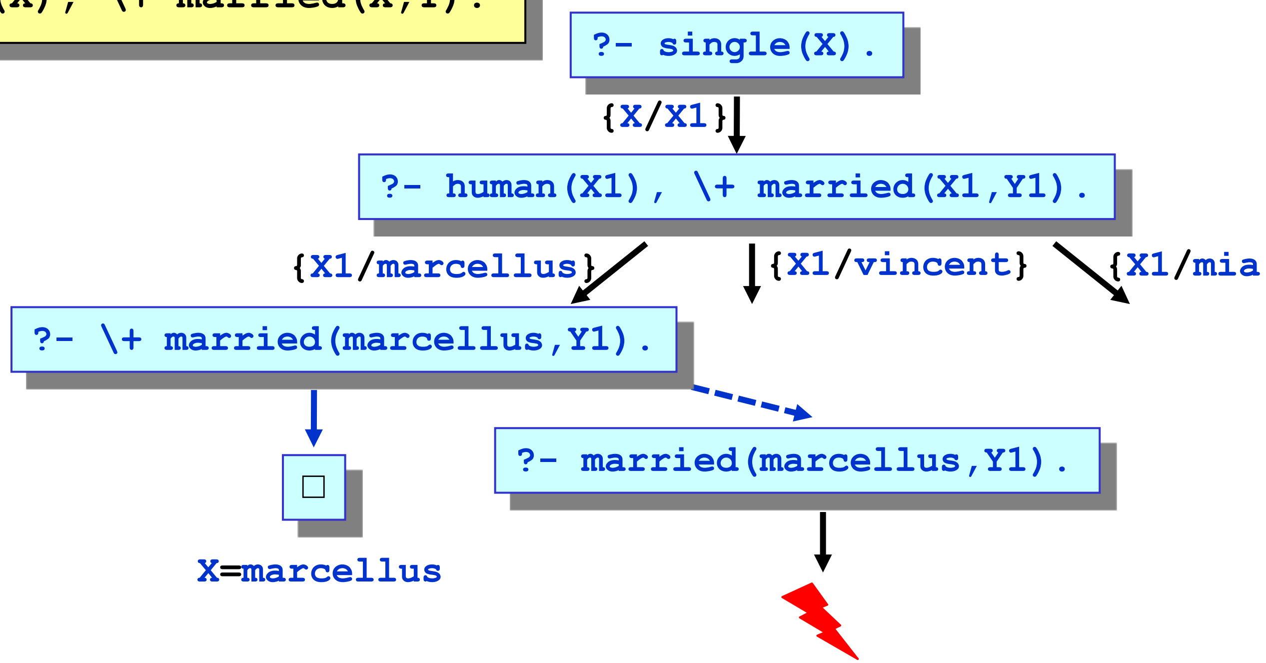
Negation (5)

<u>Beispiele mit Variablen:</u>

```
human(marcellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- human(X), \+ married(X,Y).
```



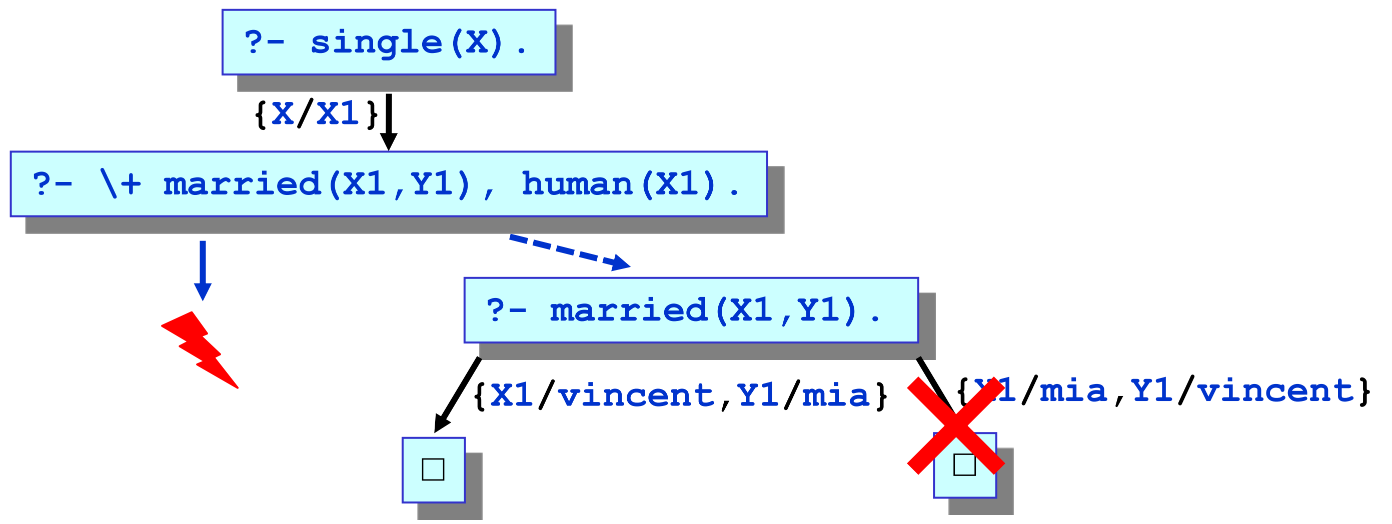
Negation (6)

<u>Beispiele mit Variablen:</u>

```
human(marcellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- \+ married(X,Y), human(X).
```

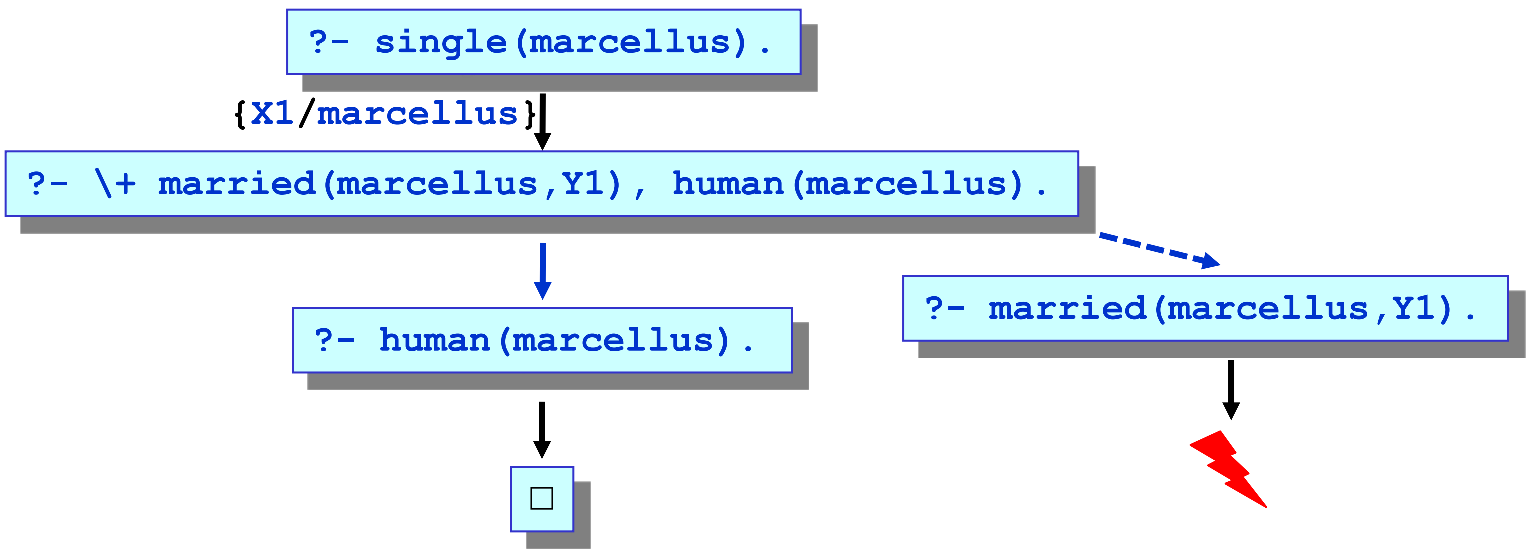


<u>Beispiele mit Variablen:</u>

```
human(marcellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- \+ married(X,Y), human(X).
```



Negation (8)

Erklärung aus „logischer Sicht“:

Unter den Annahmen, dass **`X`** ursprünglich ungebunden ist, und durch **`human(X)`** stets gebunden wird, bedeutet

```
single(X) :- human(X), \+ married(X,Y).
```

dass $\forall X : \text{human}(X) \land \neg(\exists Y : \text{married}(X,Y)) \Rightarrow \text{single}(X)$.

Unter den gleichen Annahmen bedeutet jedoch

```
single(X) :- \+ married(X,Y), human(X).
```

dass $\forall X : \neg(\exists X,Y : \text{married}(X,Y)) \land \text{human}(X) \Rightarrow \text{single}(X)$.