Programming Paradigms

Summer Term 2017

13th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Operational semantics of Prolog

As a reminder, ingredients for the operational semantics:

1. Unification
2. Resolution
3. Derivation trees

Programming Paradigms

Derivation trees

Reminder: Motivation for considering operational semantics…

We wanted to understand why, for example, for

```
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).
```

several kinds of queries/“call modes” work very well:

```
?- 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.
```

but others don’t:

```
?- 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
```

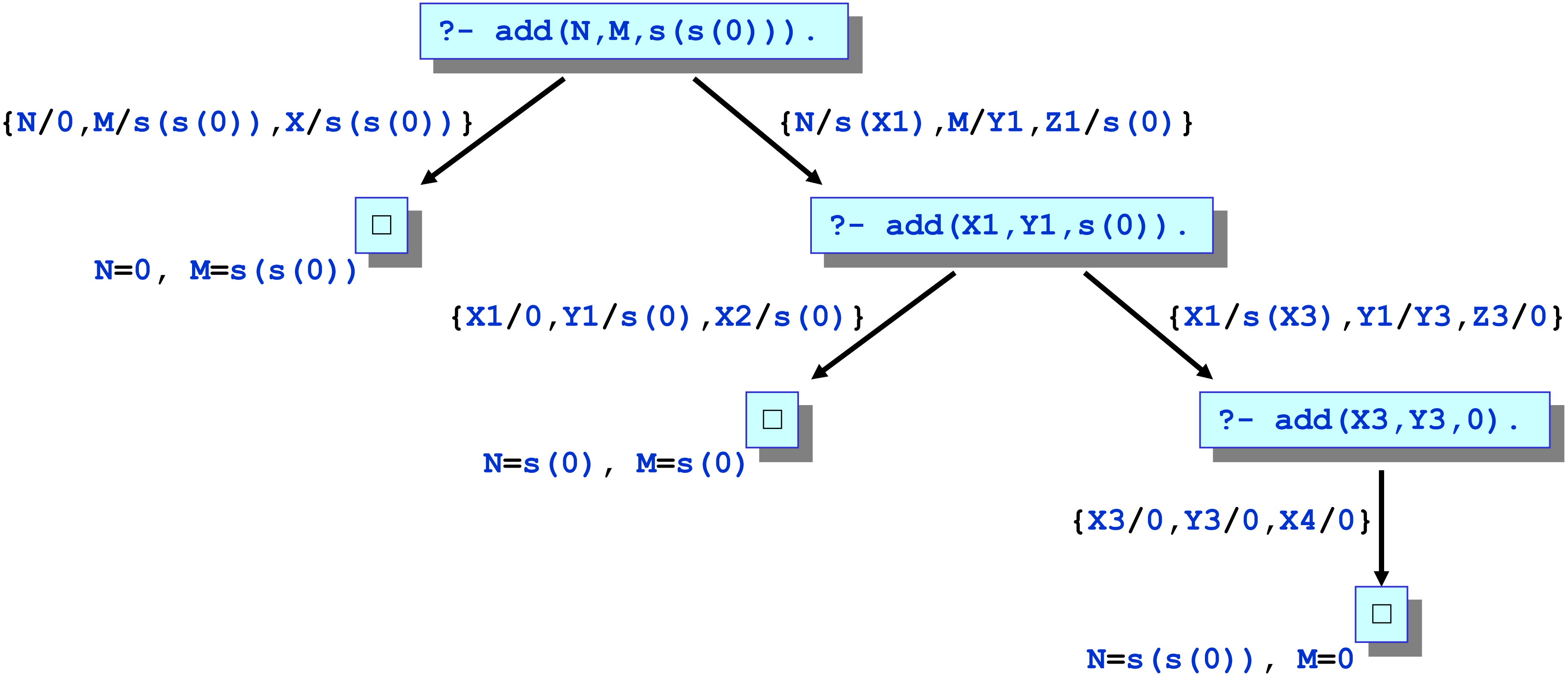
otherwise infinite search

Explicit enumeration of solutions

Let us start with a simple example just for addition:

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

Exhaustive search:



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

`{N/0,M/s(s(0)),X/s(s(0))}`

`{N/s(X1),M/Y1,Z1/s(0)}`

□

`N=0, M=s(s(0))`

`?- add(X1,Y1,s(0)).`

`{X1/0,Y1/s(0),X2/s(0)}`

`{X1/s(X3),Y1/Y3,Z3/0}`

□

`N=s(0), M=s(0)`

`?- add(X3,Y3,0).`

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

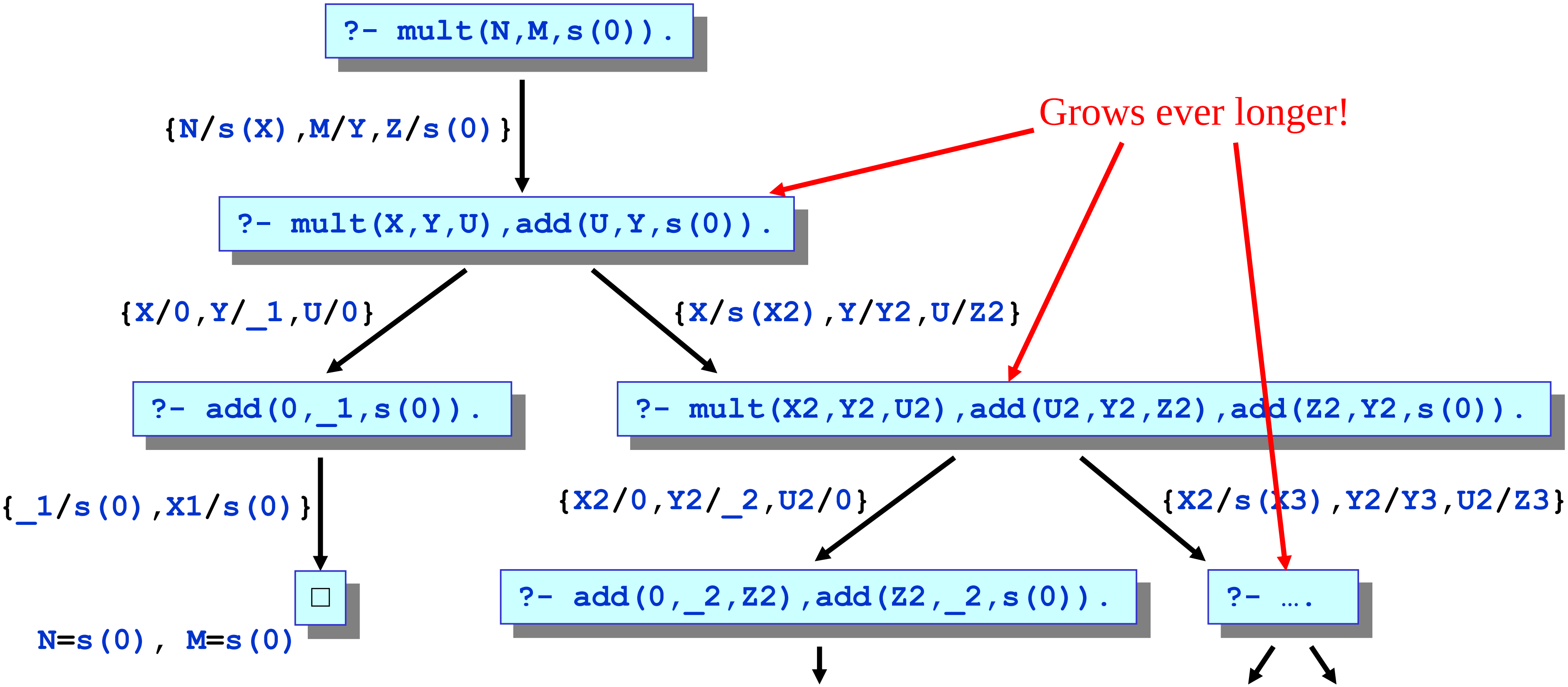
□

`N=s(s(0)), M=0`

An example with infinite search

```
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).
```



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

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

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

`{X/0,Y/_1,U/0}`

`{X/s(X2),Y/Y2,U/Z2}`

`?- add(0,_1,s(0)).`

`?- mult(X2,Y2,U2),add(U2,Y2,Z2),add(Z2,Y2,s(0)).`

`{_1/s(0),X1/s(0)}`

`{X2/0,Y2/_2,U2/0}`

`{X2/s(X3),Y2/Y3,U2/Z3}`

□

`N=s(0), M=s(0)`

`?- add(0,_2,Z2),add(Z2,_2,s(0)).`

`?- ….`

Grows ever longer!

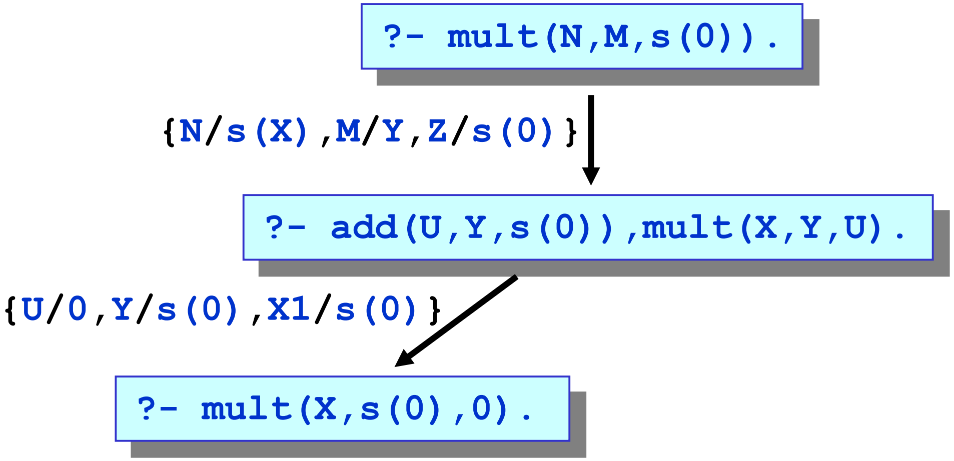
Experiment with changed order of literals

```
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).
```

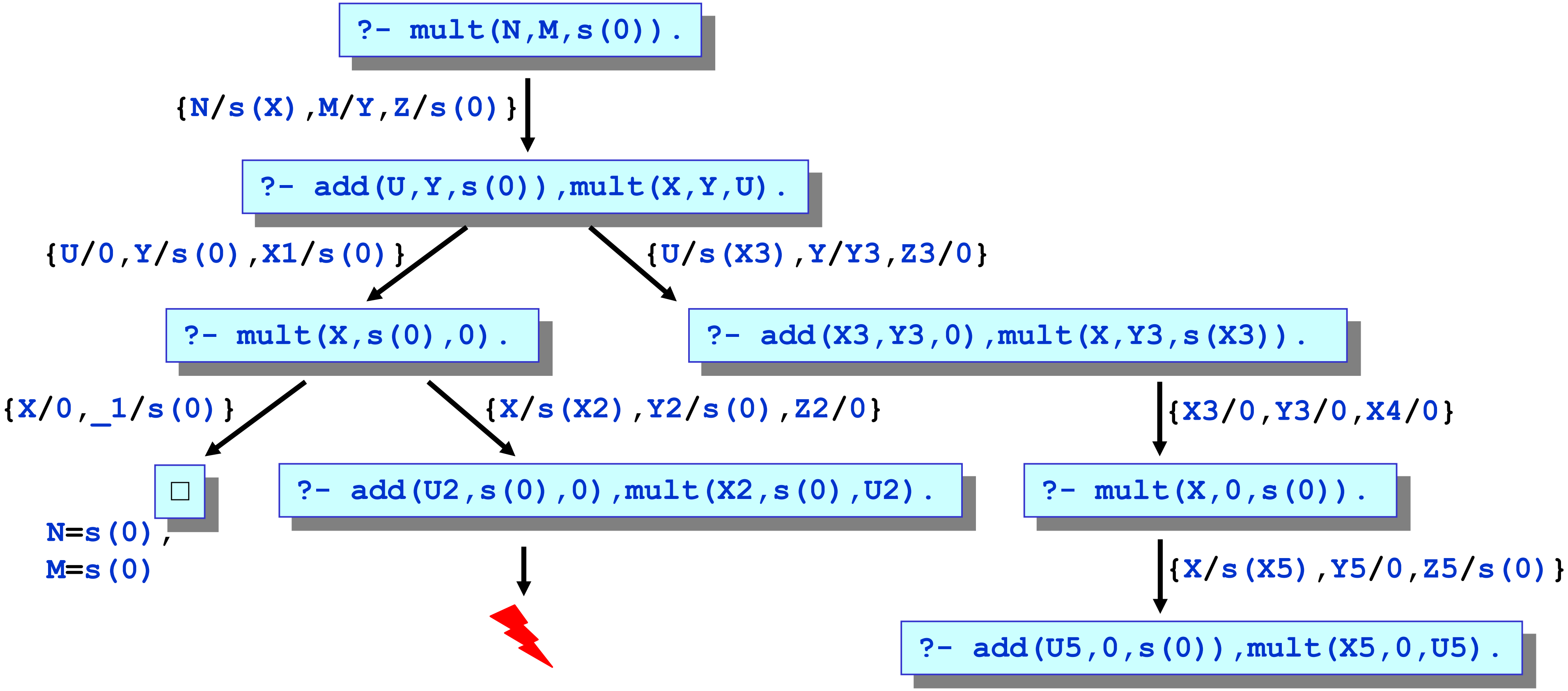
{U/0,Y/s(0),X1/s(0)}

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

Experiment with changed order of literals

```
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)}

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

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

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

{X/0,_1/s(0)}

{X/s(X2),Y2/s(0),Z2/0}

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

□

N=s(0),
M=s(0)

?- add(U2,s(0),0),mult(X2,s(0),U2).

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

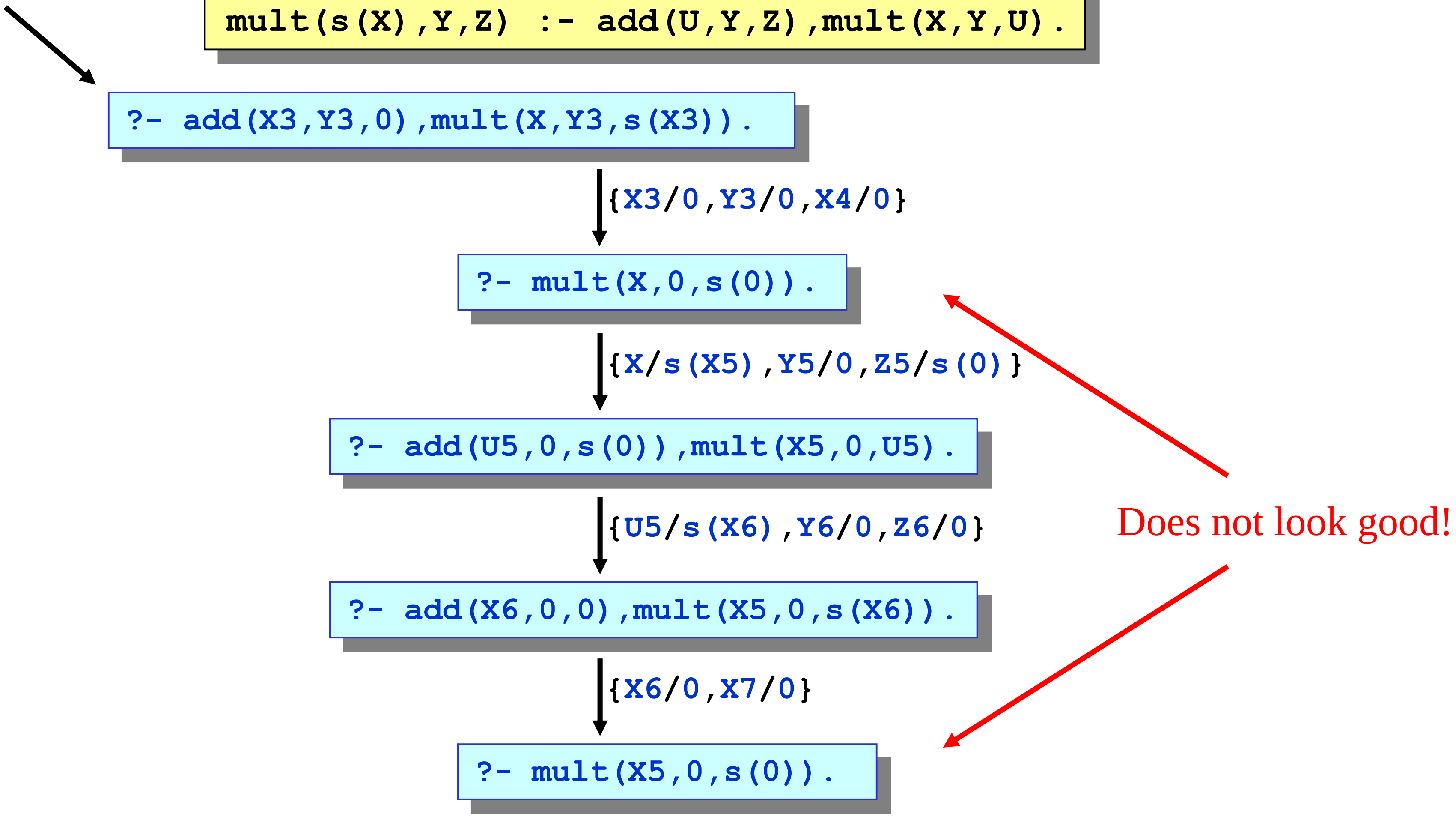
{X/s(X5),Y5/0,Z5/s(0)}

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

Experiment with changed order of literals

```
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)).
```

Does not look good!

Detailed description of the generation of derivation trees

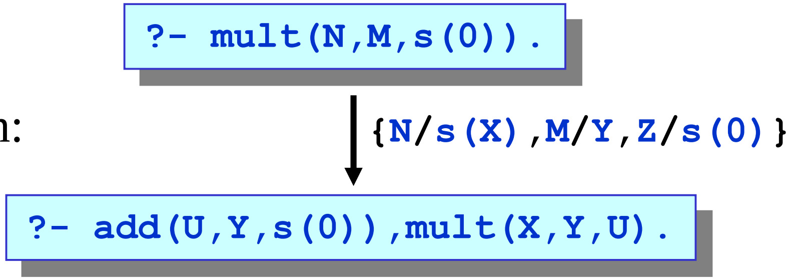
Input: query and program, for example `mult(N,M,s(0))` and:

```
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).
```

Output: tree, generated by following steps:

1. Generate root node with query, remember it as still to be worked on.
2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case



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

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

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

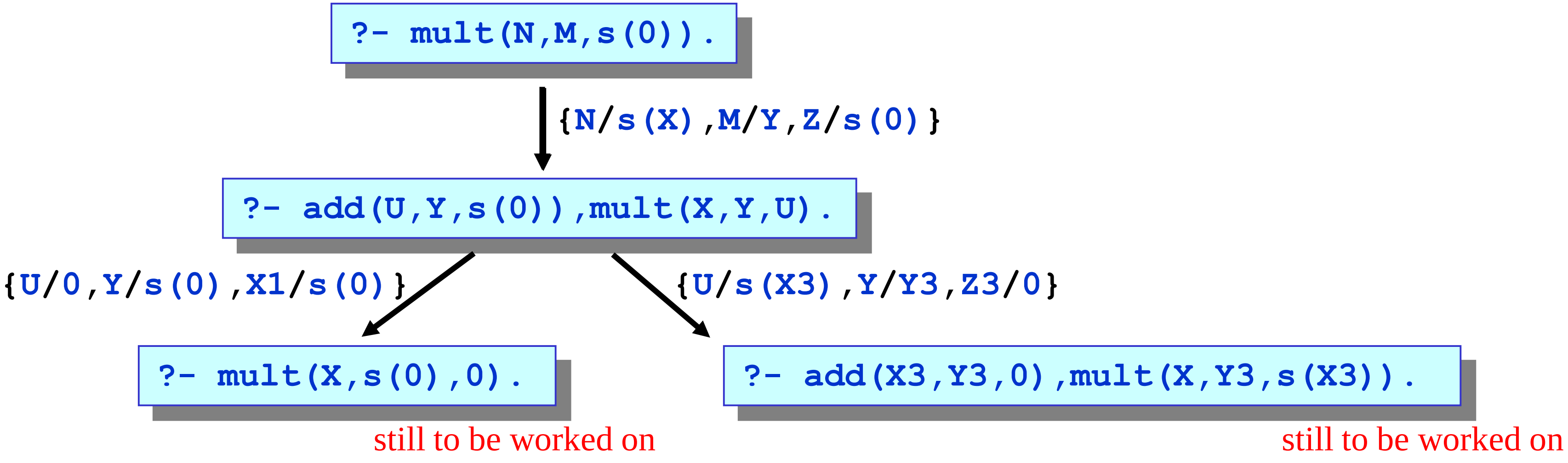
still to be worked on

Detailed description of the generation of derivation trees

```
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. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case



`?- 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).`

still to be worked on

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

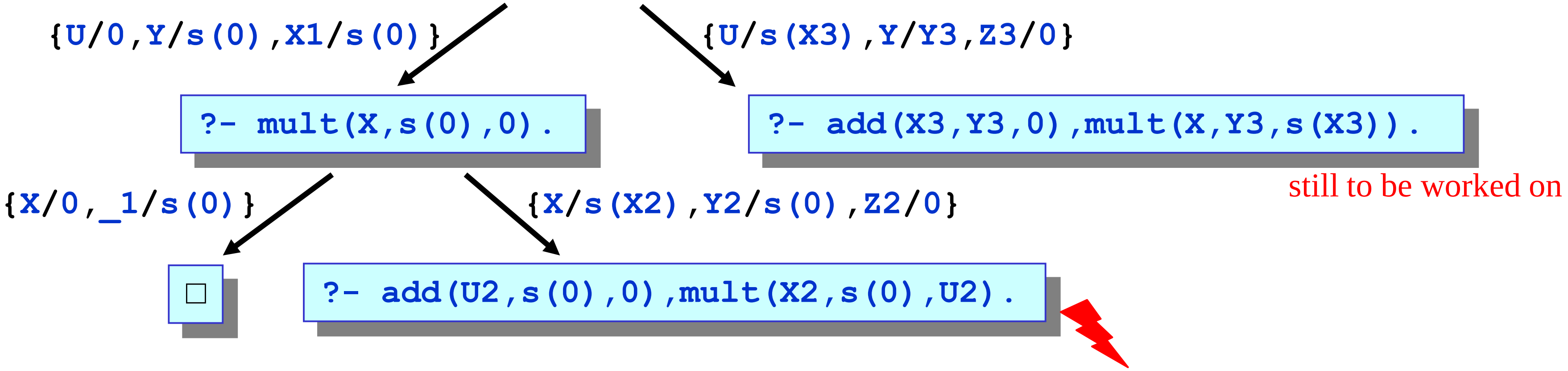
still to be worked on

Detailed description of the generation of derivation trees

```
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. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case
   - mark nodes as finished if they are empty (□) or if their left-most literal is not unifiable with any fact/rule head (⚡)



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

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

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

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

still to be worked on

`{X/0,_1/s(0)}`

`{X/s(X2),Y2/s(0),Z2/0}`

□

`?- add(U2,s(0),0),mult(X2,s(0),U2).`

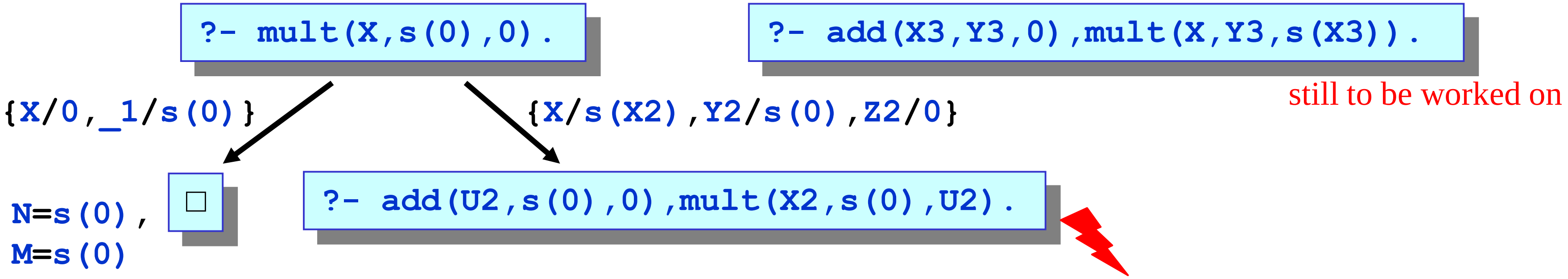
⚡

Detailed description of the generation of derivation trees

```
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. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case
   - mark nodes as finished if they are empty or if their left-most literal is not unifiable with any fact/rule head
   - at successful nodes, annotate the solution (the composition of unifiers along the path from the root, applied to all variables that occurred in the original query)



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

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

still to be worked on

`{X/0,_1/s(0)}`

`{X/s(X2),Y2/s(0),Z2/0}`

`N=s(0),`
`M=s(0)`

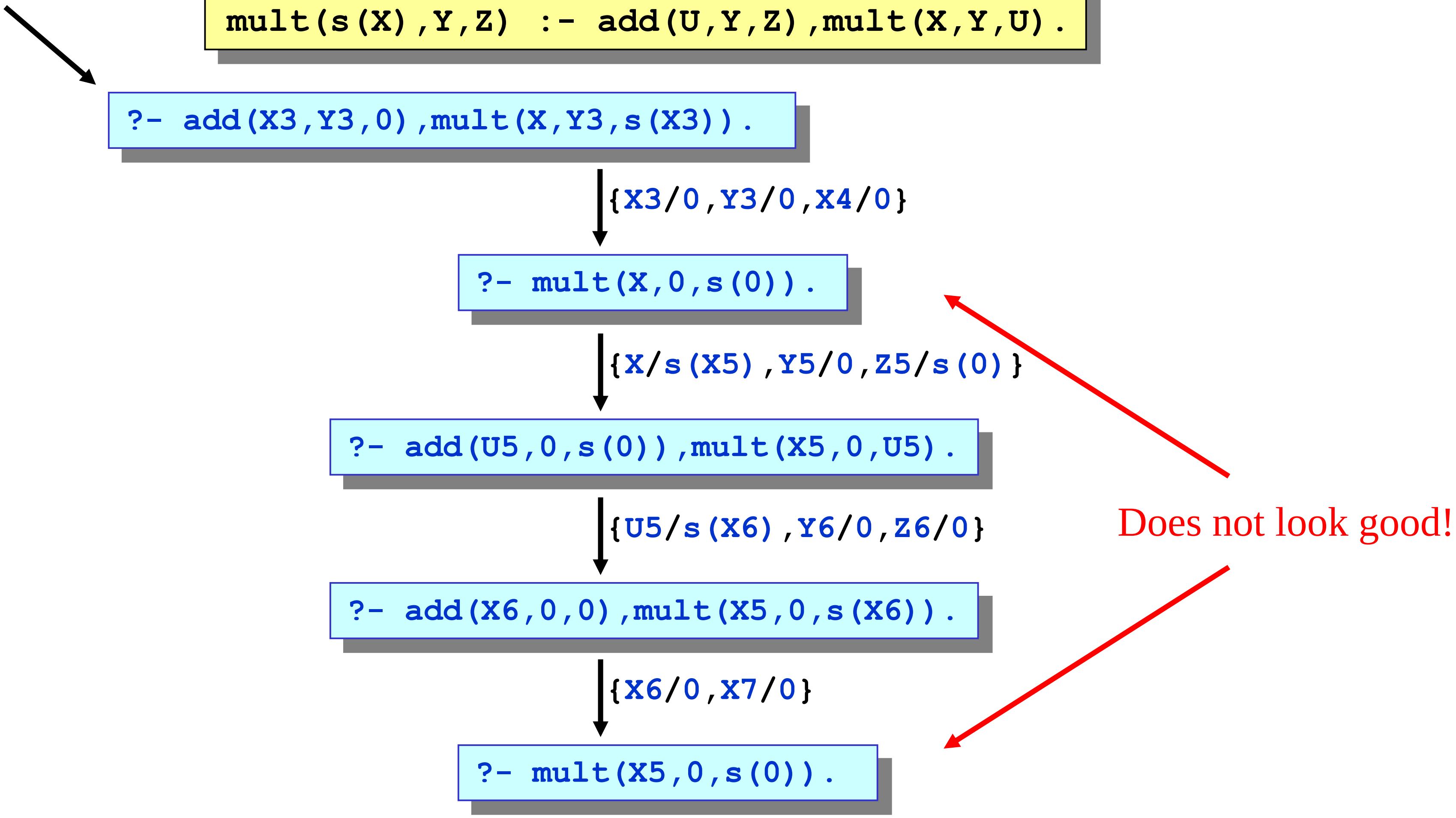
□

`?- add(U2,s(0),0),mult(X2,s(0),U2).`

Back to the example: What to do?

```
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}

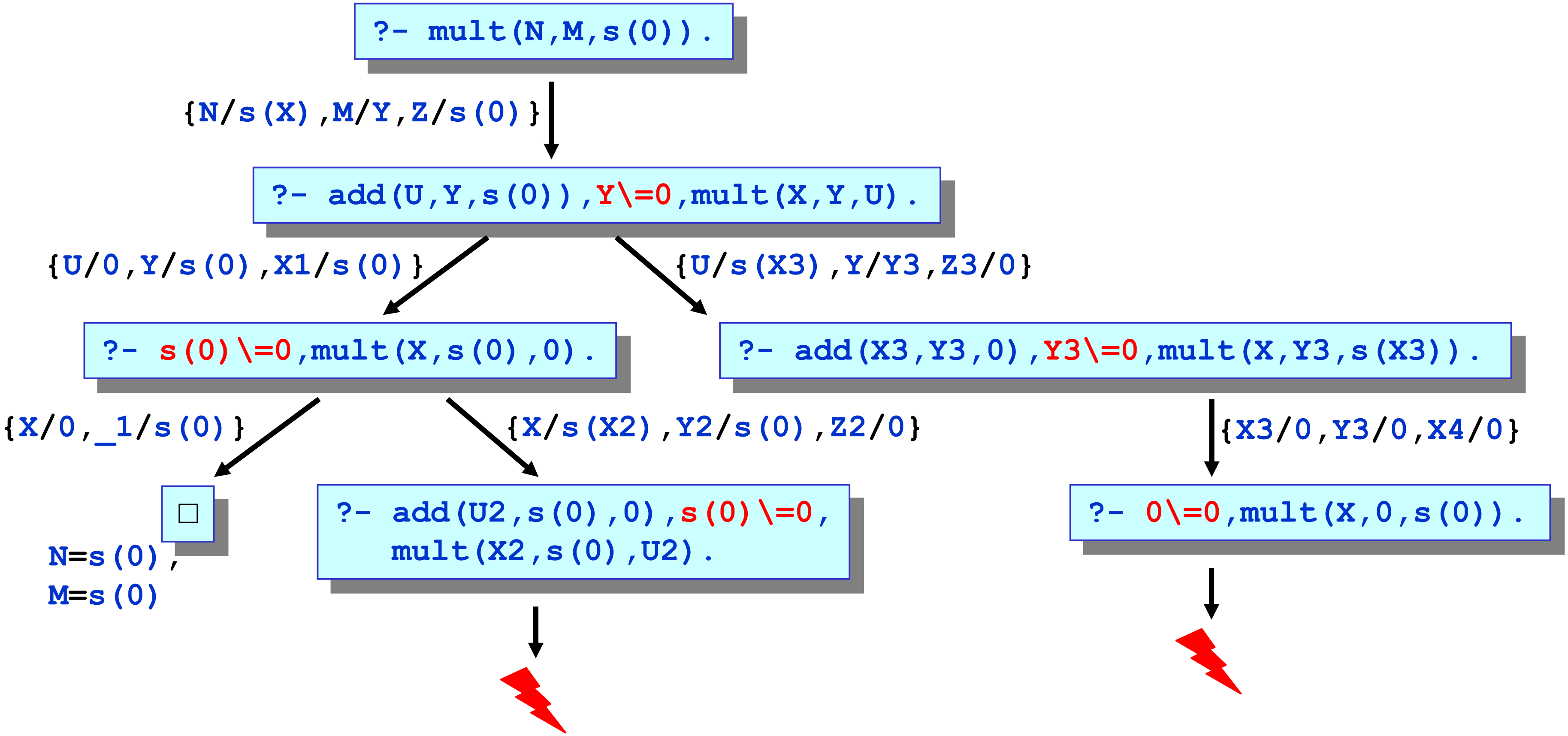
```
?- mult(X5,0,s(0)).
```

Does not look good!

Attempt: introducing an extra test

```
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(0)).`

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

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

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

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

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

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

`{X/0,_1/s(0)}`

`{X/s(X2),Y2/s(0),Z2/0}`

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

□

`N=s(0),`
`M=s(0)`

`?- add(U2,s(0),0),s(0)\=0,`
`mult(X2,s(0),U2).`

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

Only partial success

```
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.
```

New results found, old results lost!

Yet another "repair"

```
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).
```

Now this works:

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

And it even works generally `mult(?X,?Y,+Z).`

But unfortunately (only noticed now):

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

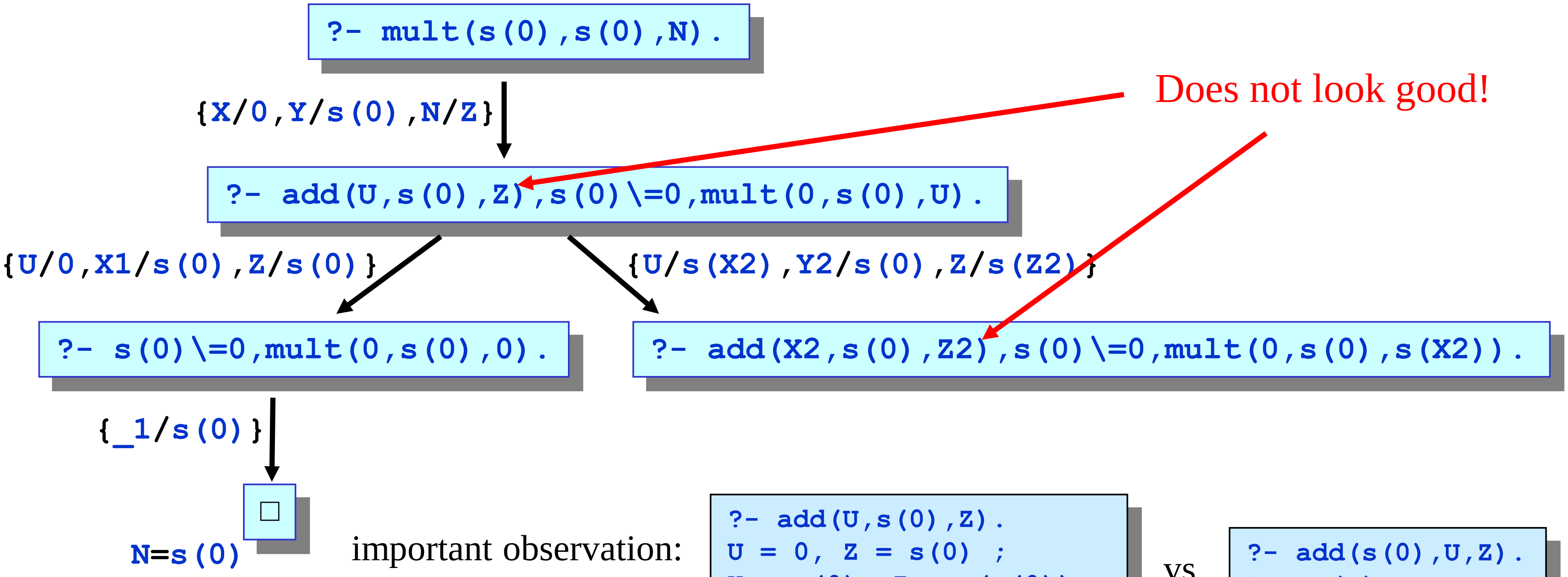
otherwise infinite search

So `mult(+X,+Y,?Z).` does not anymore work.

A new “infinity trap”

```
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).`

Does not look good!

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

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

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

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

`{_1/s(0)}`

□

`N=s(0)`

important observation:
(see last lecture)

```
?- 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).
```

Exploiting commutativity

```
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).
```

important observation:
(see last lecture)

```
?- 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).
```

Exploiting commutativity

```
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)

Indeed a generally useful 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)).
...
```

Now all call modes work well, except **`mult(?X,?Y,?Z)`**!

“Lessons learned”

The operational semantics:

- reflects the actual Prolog search process, with backtracking
- makes essential use of unification (and resolution steps)
- enables understanding of effects like non-termination
- gives insight into impact of changes to the order of, and within, facts and rules

Programming Paradigms

Negation in Prolog

Negation (1)

- Logic programming is primarily based on a positive logic.

  A literal is provable if it can be reduced (possibly via several resolution steps) to the validity of known facts.

- But Prolog also offers the possibility to use **negation**.
  - However, Prolog negation is not fully compatible with the expected logical meaning.
  - `\+ Goal`, or `not(Goal)`, is provable if and only if `Goal` is not provable.

Example: `\+ member(4,[2,3])` is provable, since `member(4,[2,3])` is not provable, i.e., it exists a “finite failure tree”.

Caution:

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

(Negation does not yield variable bindings.)

Negation (2)

- Why “finite failure tree”?
  - We cannot, in general, show that from the clauses of a program a certain negative statement follows.
  - We can only show that a certain positive statement can not be deduced. (negation as failure)
  - Here, “show” means to attempt a proof of the positive statement but to fail.
  - That any such attempt will necessarily fail (for some given positive statement) can only be said with certainty if the search space is finite.

- Underlying assumption:

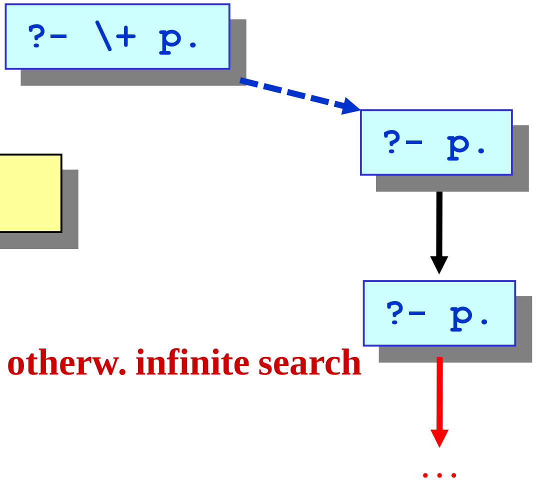
closed world assumption

Negation (3)

Examples:

```
p :- p.
```

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


?- \+ p.

?- p.

?- p.

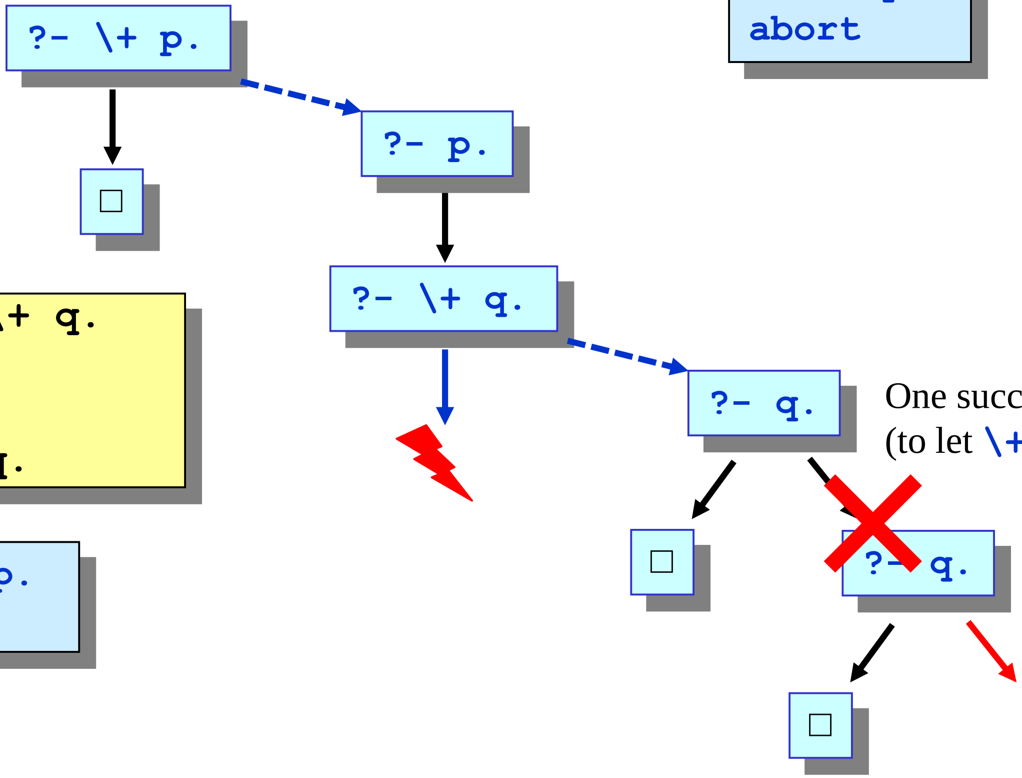
otherw. infinite search

...

```
p :- \+ q.

q.
q :- q.
```

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


?- \+ p.

□

?- p.

?- \+ q.

?- q.

□

?- q.

□

...

One successful branch (for **`q`**) suffices (to let **`\+ q`** fail).

Negation (4)

Examples with variables:

```
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.
```

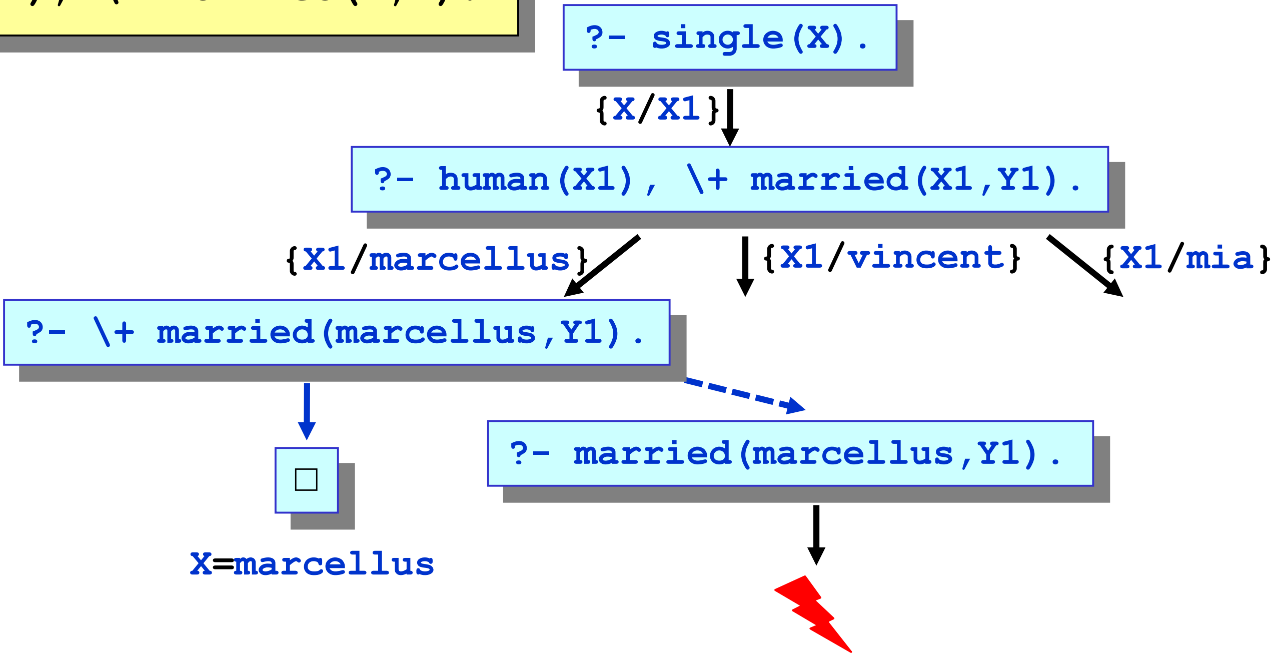
Negation (5)

Examples with variables:

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

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

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



?- single(X).

{X/X1}

?- human(X1), \+ married(X1,Y1).

{X1/marcellus}

{X1/vincent}

{X1/mia}

?- \+ married(marcellus,Y1).

□

X=marcellus

?- married(marcellus,Y1).

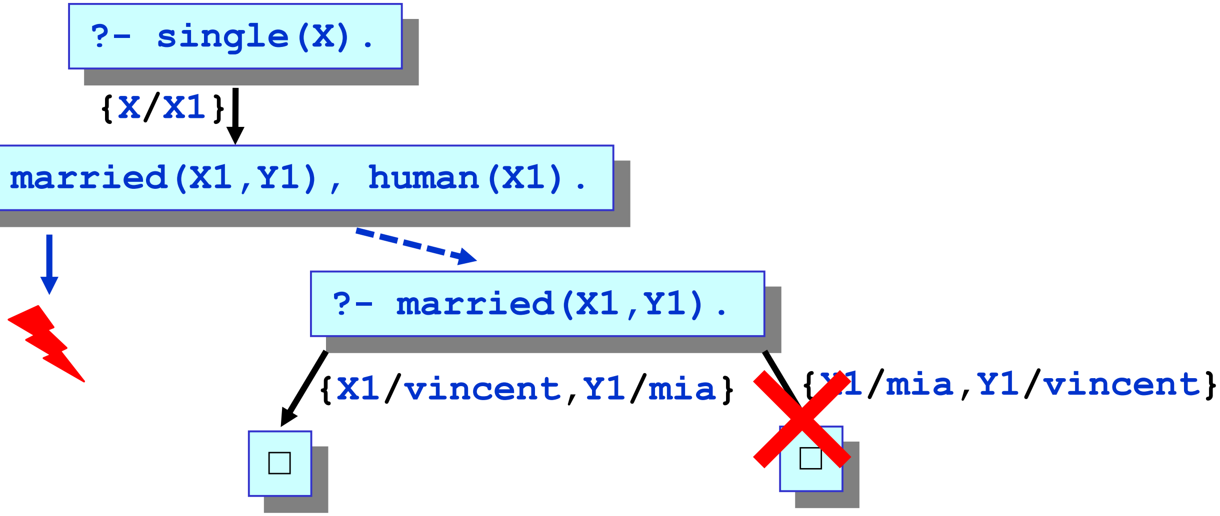
Negation (6)

Examples with variables:

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

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

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



?- single(X).

{X/X1}

?- \+ married(X1,Y1), human(X1).

?- married(X1,Y1).

{X1/vincent,Y1/mia}

{X1/mia,Y1/vincent}

□

□

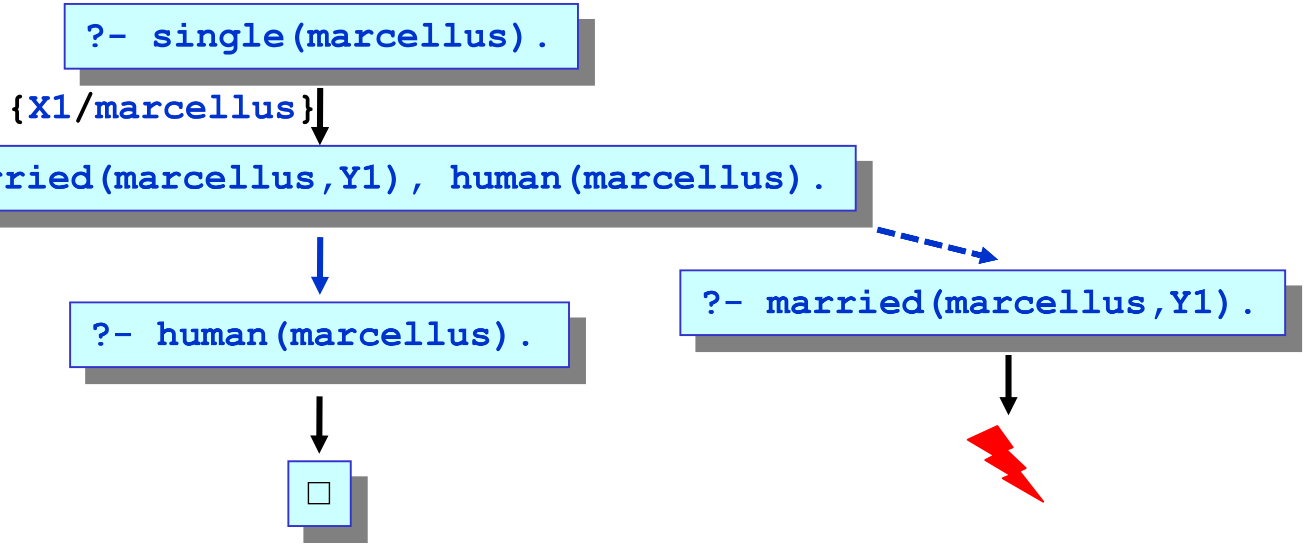
Negation (7)

Examples with variables:

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

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

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



`?- single(marcellus).`

{X1/marcellus}

`?- \+ married(marcellus,Y1), human(marcellus).`

`?- human(marcellus).`

□

`?- married(marcellus,Y1).`

Negation (8)

Explanation from “logical perspective” :

Under the assumptions that **`X`** is originally unbound and by **`human(X)`** will always be bound, this:

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

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

But under the same assumptions, this:

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

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

Summary on Negation

- no real logical negation: instead, negation as failure
- proof search in “side branch”, does not bind variables to the outside
- can only be truly understood procedurally/operationally
- problems with attempting a declarative perspective:
  - not compositional
  - sensitive against changes to the order of, and within, facts and rules
  - T_P-operator would be non-monotone