

Prolog Lecture 3

- Symbolic evaluation of arithmetic
- Controlling backtracking: cut
- Negation

Symbolic Evaluation

Let's write some Prolog rules to evaluate symbolic arithmetic expressions such as `plus(1,mult(4,5))`

```
eval(plus(A,B),C) :- eval(A,A1),  
                     eval(B,B1),  
                     C is A1 + B1.
```

```
eval(mult(A,B),C) :- eval(A,A1),  
                    eval(B,B1),  
                    C is A1 * B1.
```

```
eval(A,A).
```

Evaluation starts with the first matching clause

Q: How does Prolog evaluate:

```
eval(plus(1,mult(4,5)),Ans)
```

A: Step 1, see if the first matching clause is true

```
eval(plus(A,B),C) :- eval(A,A1),  
                     eval(B,B1),  
                     C is A1 + B1.
```

In this case the variable bindings are:

- A = 1, B = mult(4,5) and C = Ans

Next it looks at the body of the rule

The body of the clause with head `eval(plus(A, B), C)` and variable bindings

`A = 1`, `B = mult(4,5)` and `C = Ans` is:

```
eval(1, A1),  
eval(mult(4, 5), B1),  
Ans is A1 + B1.
```

This is a conjunction: all parts must be true for the clause to be true

The body is checked term by term from left to right

First part of the body: `eval(1,A1)`

Try: `eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.`

Fail because 1 does not unify with `plus(A,B)`

Try: `eval(mult(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 * B1.`

Fail because 1 does not unify with `mult(A,B)`

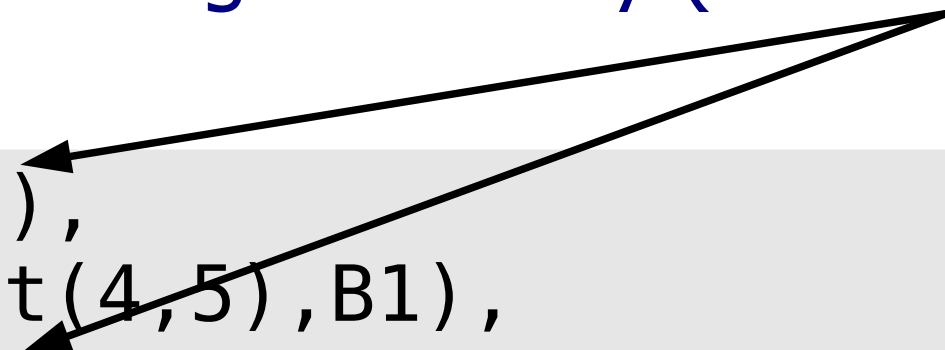
Try: `eval(A,A).`

Succeed: `eval(1,A1)` is true if `A1 = 1`

The body is checked term by term from left to right

From previous slide, `eval(1, A1)` was provable,
with the side-effect of binding: `A1=1`.

So continuing through the body (note `A1` is now
bound):



```
eval(1, 1),  
eval(mult(4, 5), B1),  
Ans is 1 + B1.
```

The body is checked term by term
from left to right

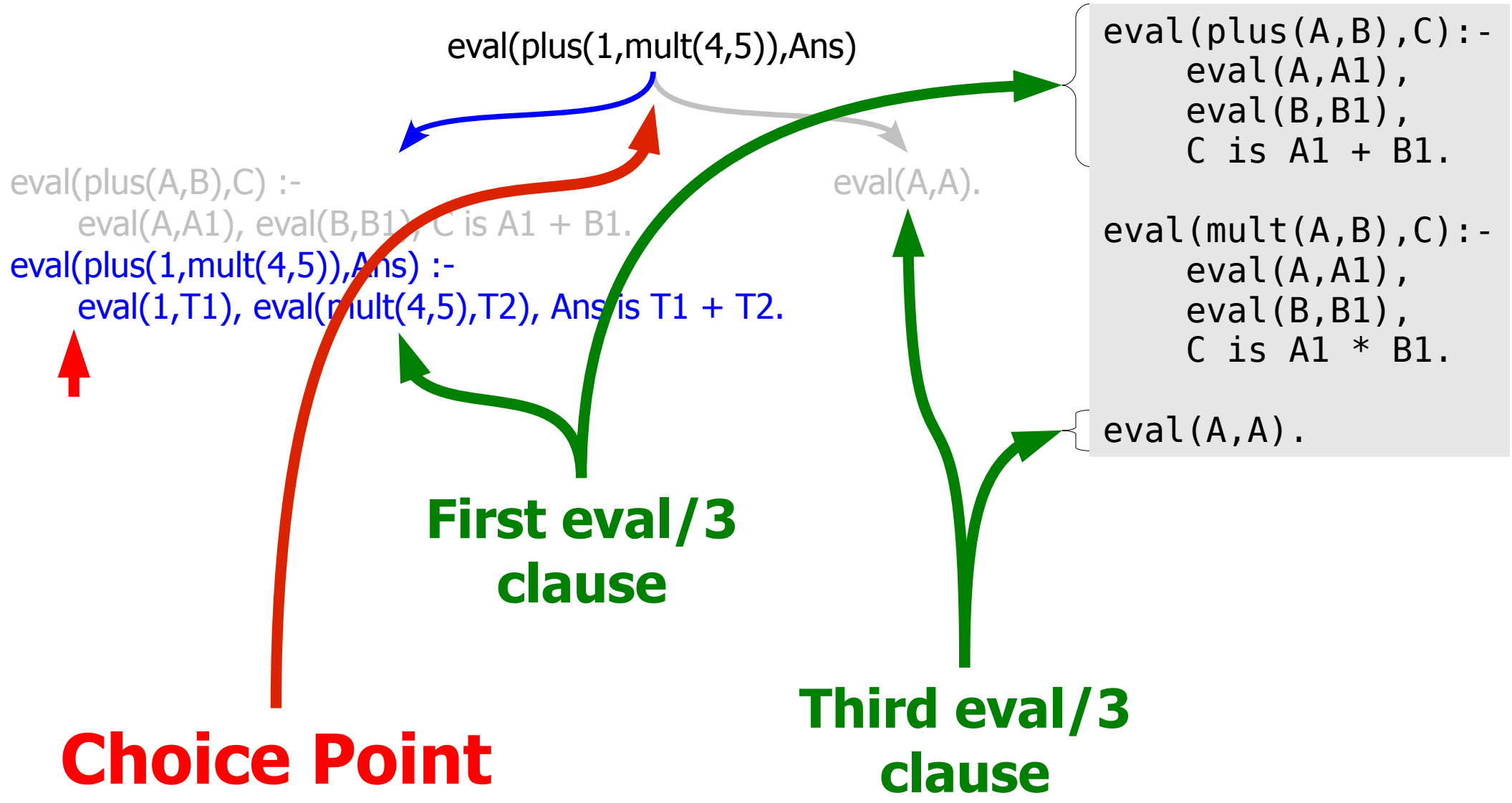
So `eval(mult(4,5), B1)` will bind `B1=20`:

```
eval(1,1),  
eval(mult(4,5),20),  
Ans is 1 + 20.
```

The body is checked term by term
from left to right

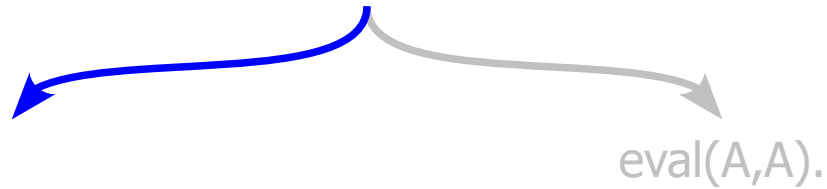
Ans will be bound to 21, after "is" does its job.

```
eval(1,1),  
eval(mult(4,5),20),  
21 is 1 + 20.
```

Be sure that you understand why the second eval/3 clause does not appear in this choice point

eval(plus(1,mult(4,5)),Ans)



eval(plus(A,B),C) :-

eval(A,A1), eval(B,B1), C is A1 + B1.

eval(plus(1,mult(4,5)),Ans) :-

eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.



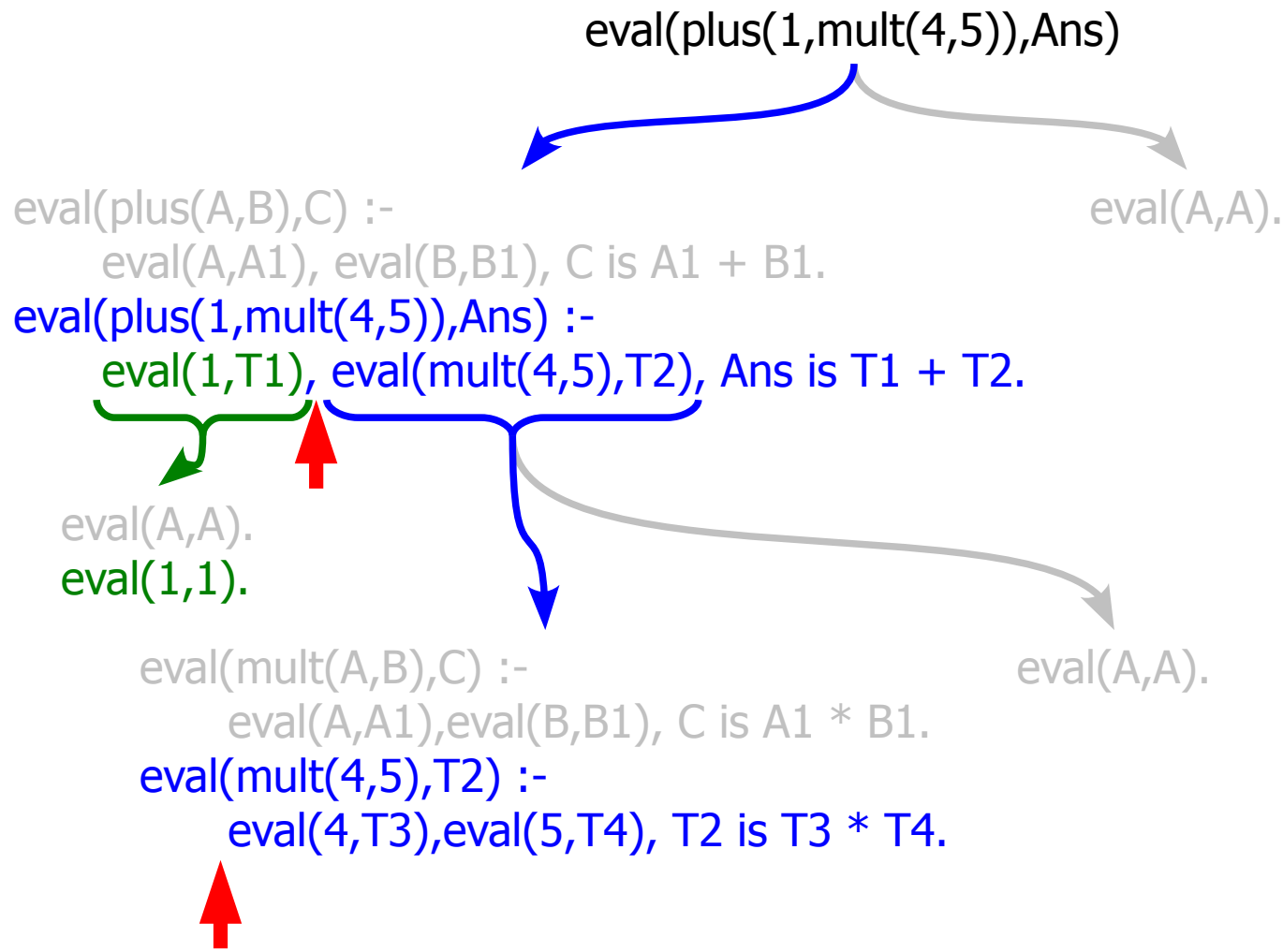
eval(A,A).

eval(1,1).

```
eval(plus(A,B),C) :-  
    eval(A,A1),  
    eval(B,B1),  
    C is A1 + B1.
```

```
eval(mult(A,B),C) :-  
    eval(A,A1),  
    eval(B,B1),  
    C is A1 * B1.
```

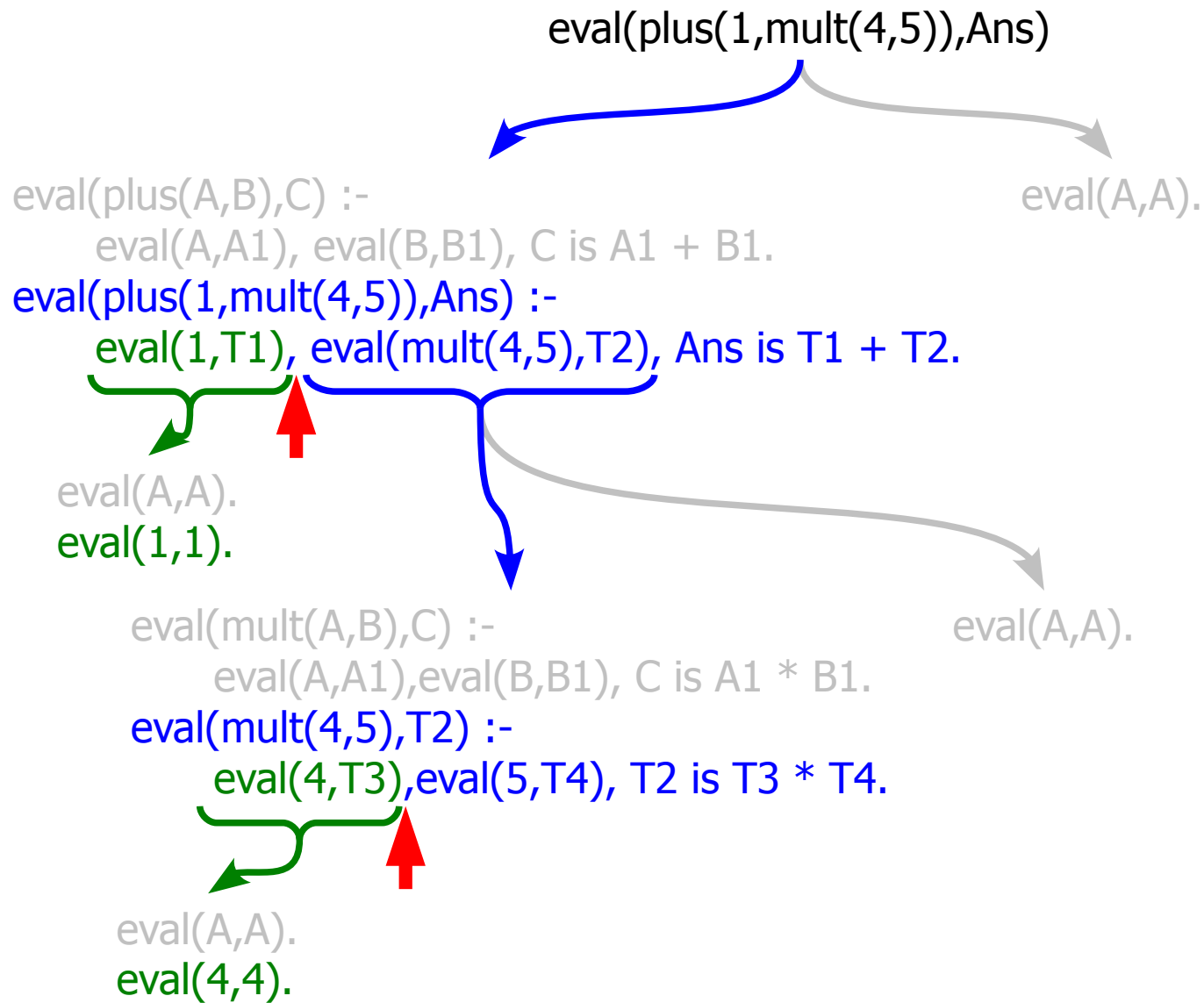
```
eval(A,A).
```



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.

eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.

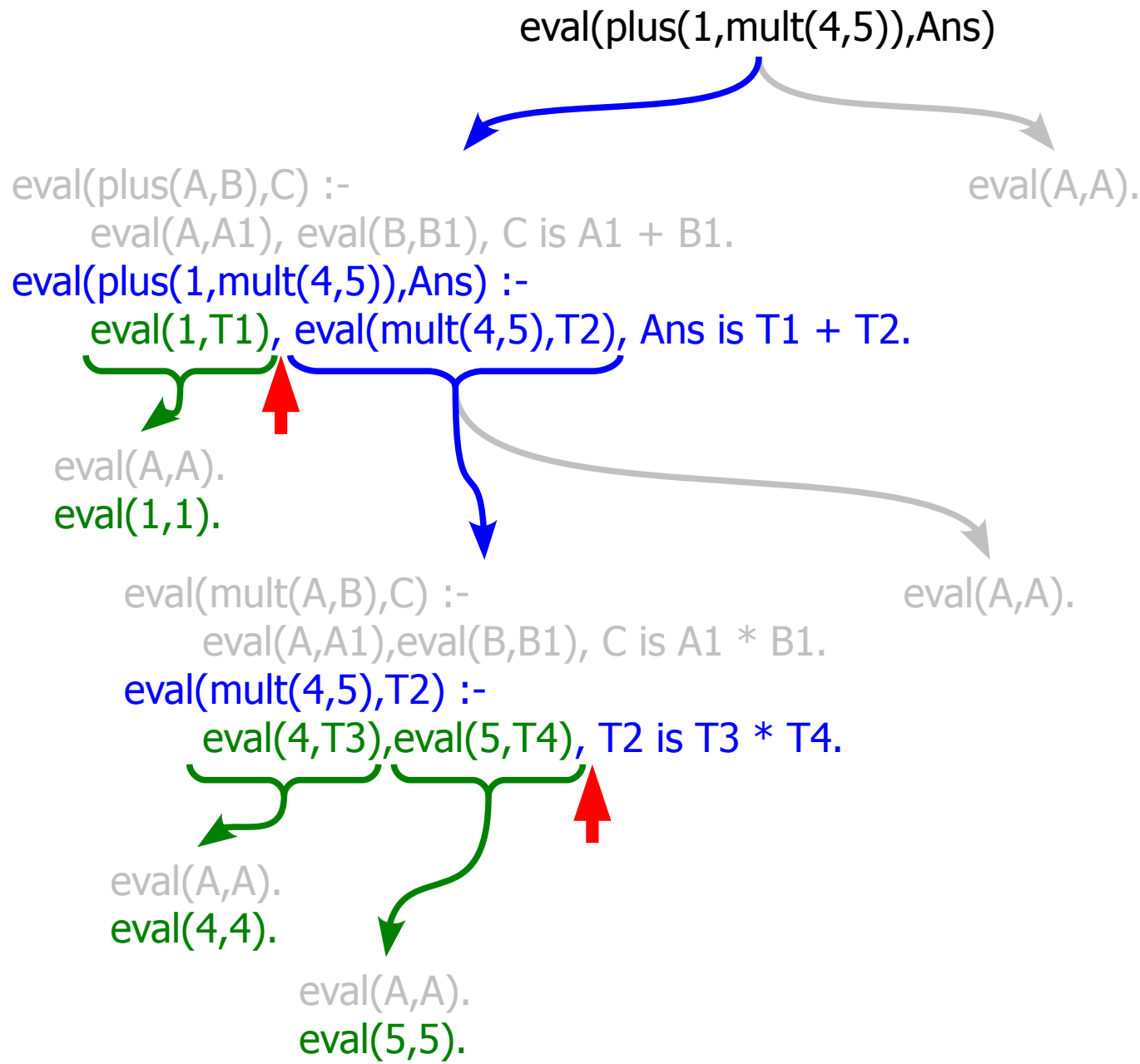
eval(A,A).
```



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.
```

```
eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.
```

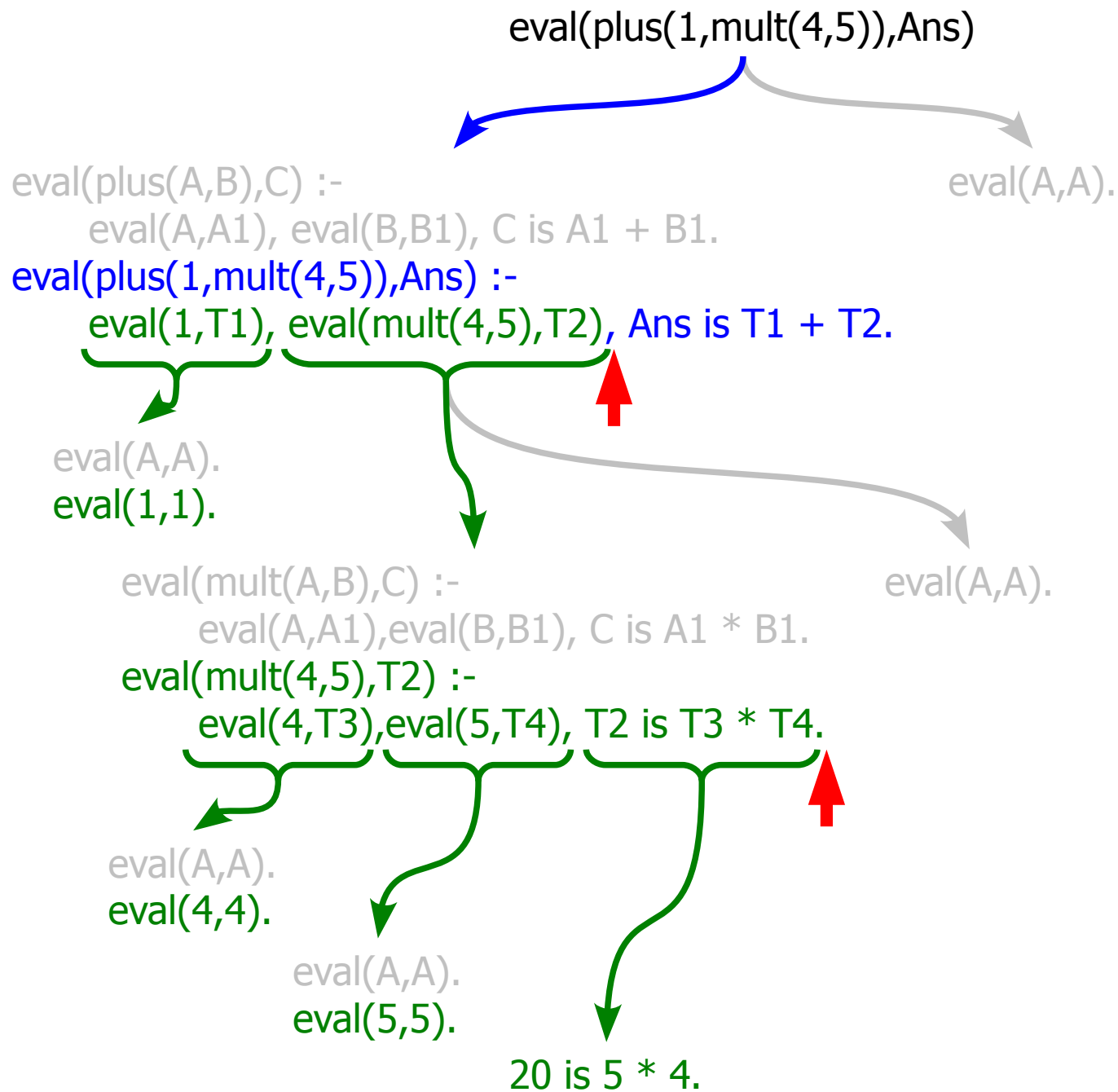
```
eval(A,A).
```



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.

eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.

eval(A,A).
```



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.

eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.

eval(A,A).
```

eval(plus(1,mult(4,5)),Ans)

eval(A,A).

eval(plus(A,B),C) :-

eval(A,A1), eval(B,B1), C is A1 + B1.

eval(plus(1,mult(4,5)),Ans) :-

eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-

eval(A,A1),eval(B,B1), C is A1 * B1.

eval(mult(4,5),T2) :-

eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(4,4).

eval(A,A).
eval(5,5).

20 is 5 * 4.

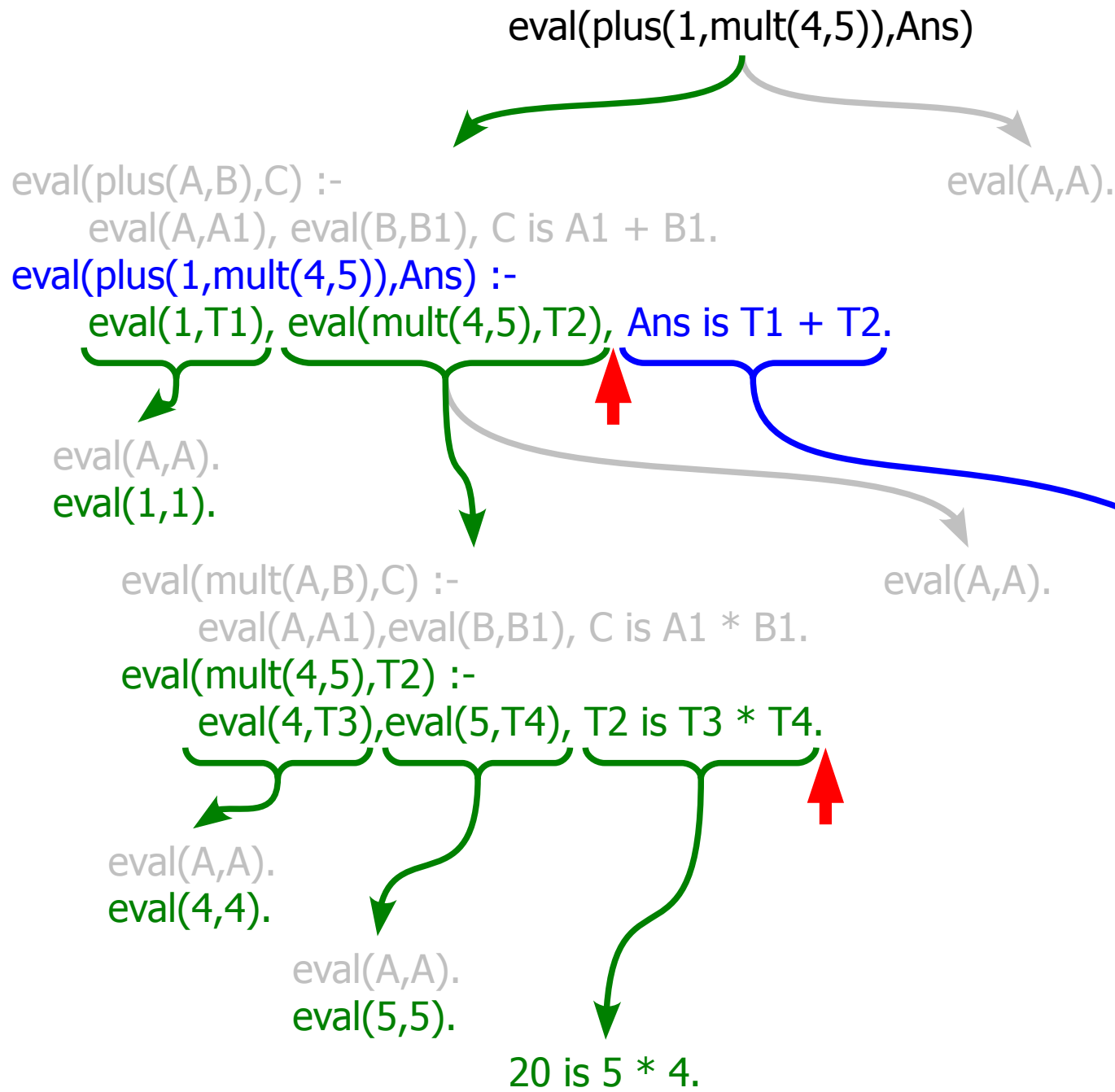
21 is 1 + 20.

```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.
```

```
eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.
```

```
eval(A,A).
```

What happens if we use backtracking and ask Prolog for the next solution?

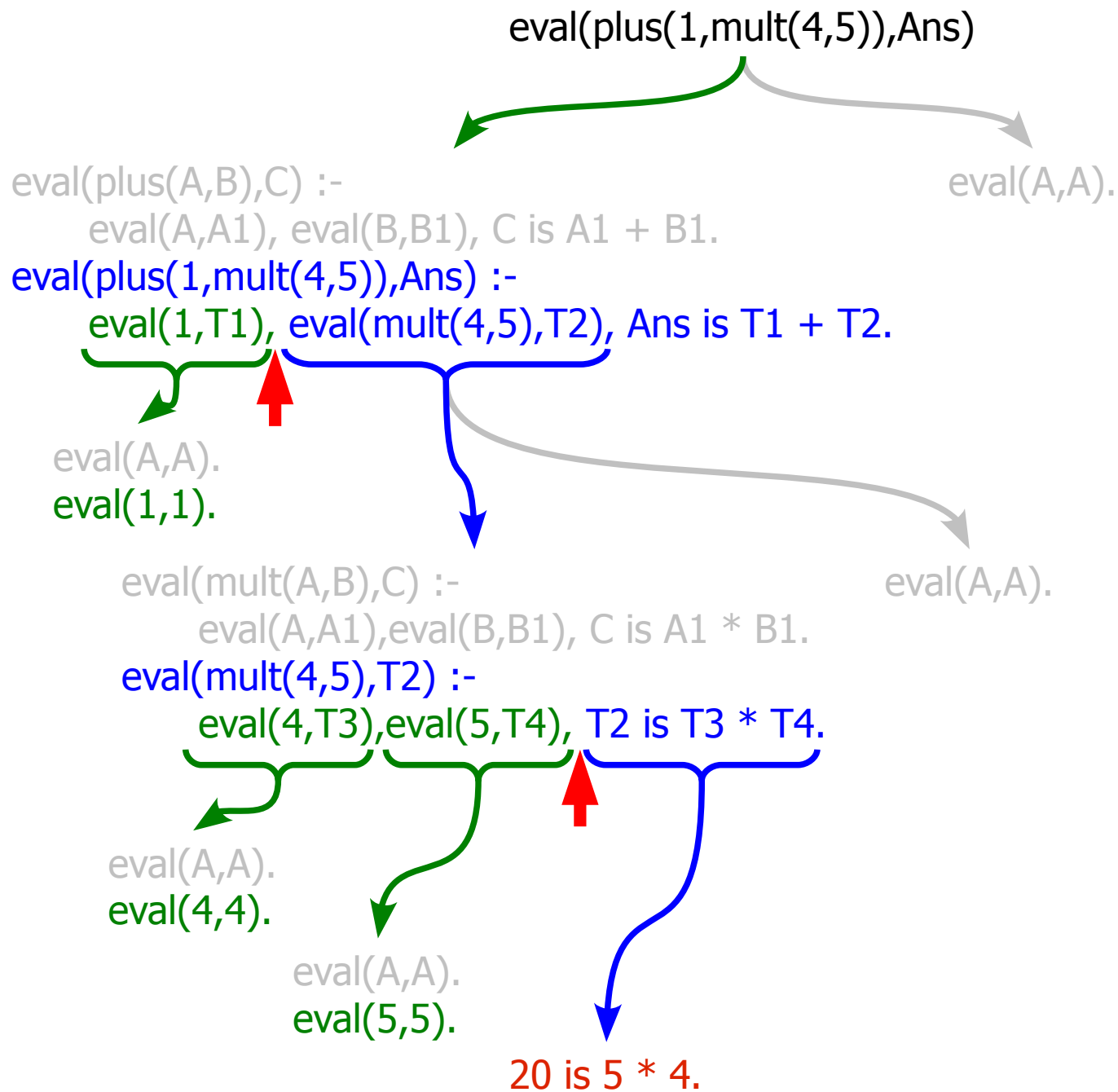


```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.

eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.

eval(A,A).
```

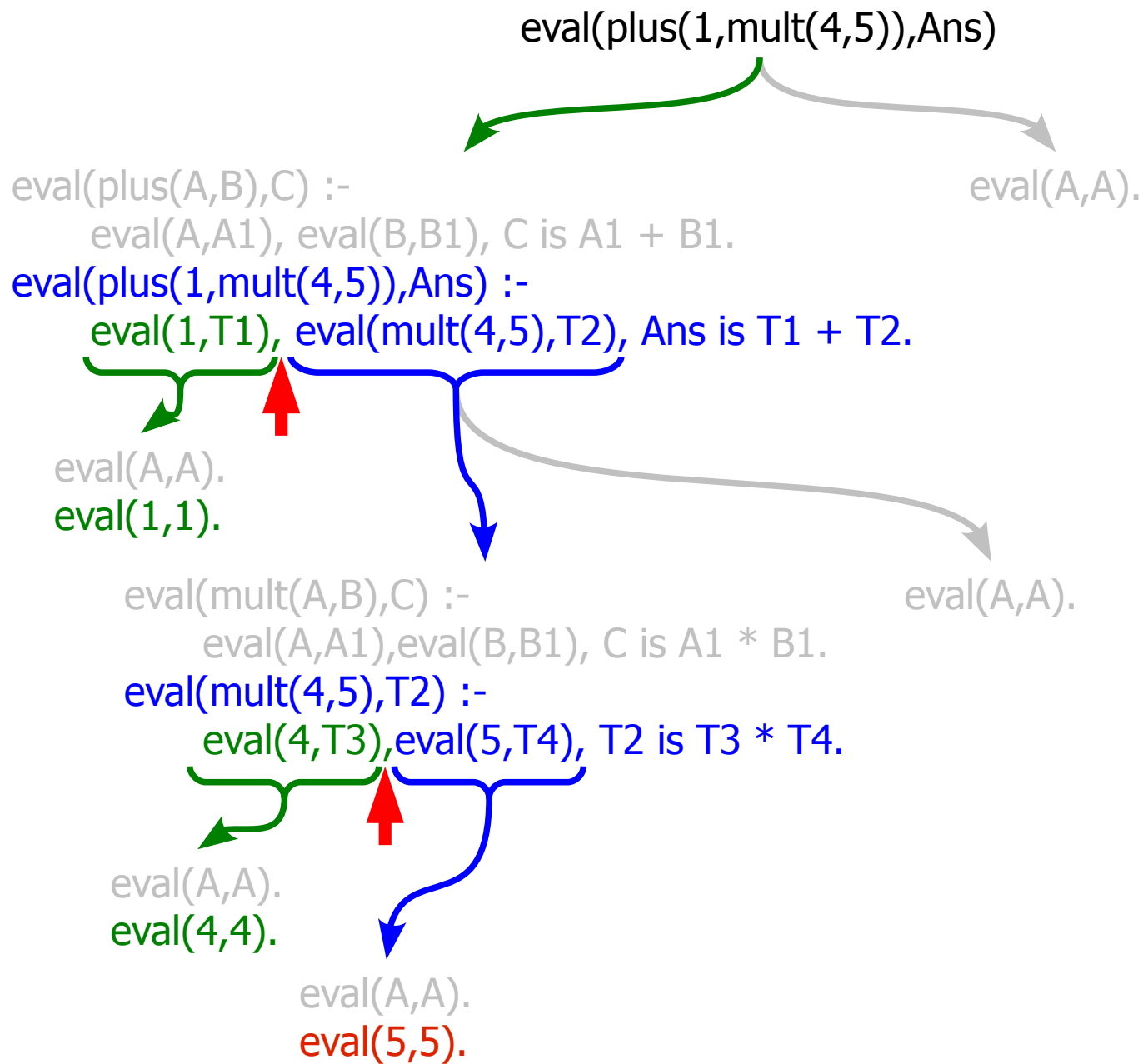
~~21 is 1 + 20.~~



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.

eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.

eval(A,A).
```



```

eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.

```

```

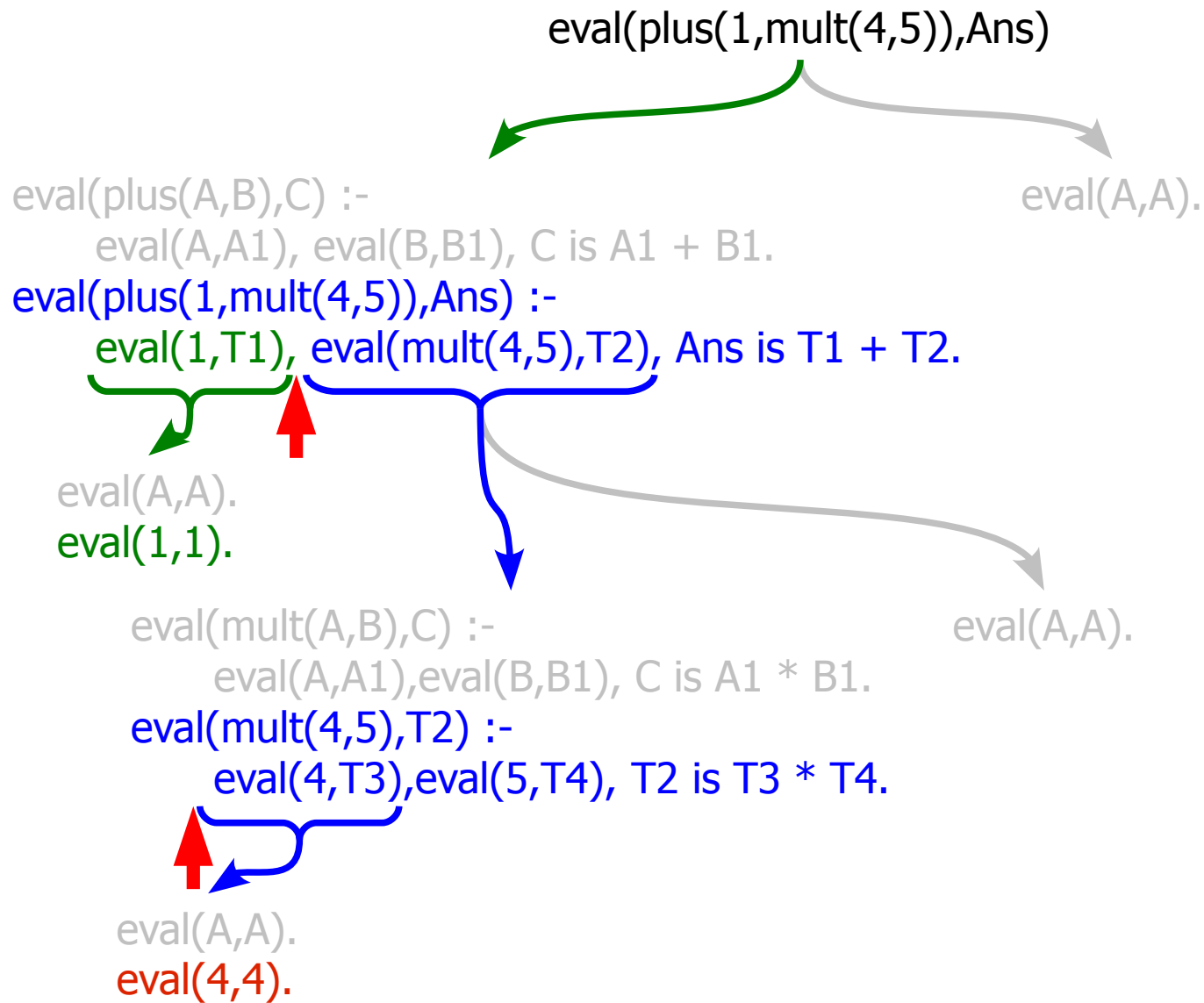
eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.

```

```

eval(A,A).

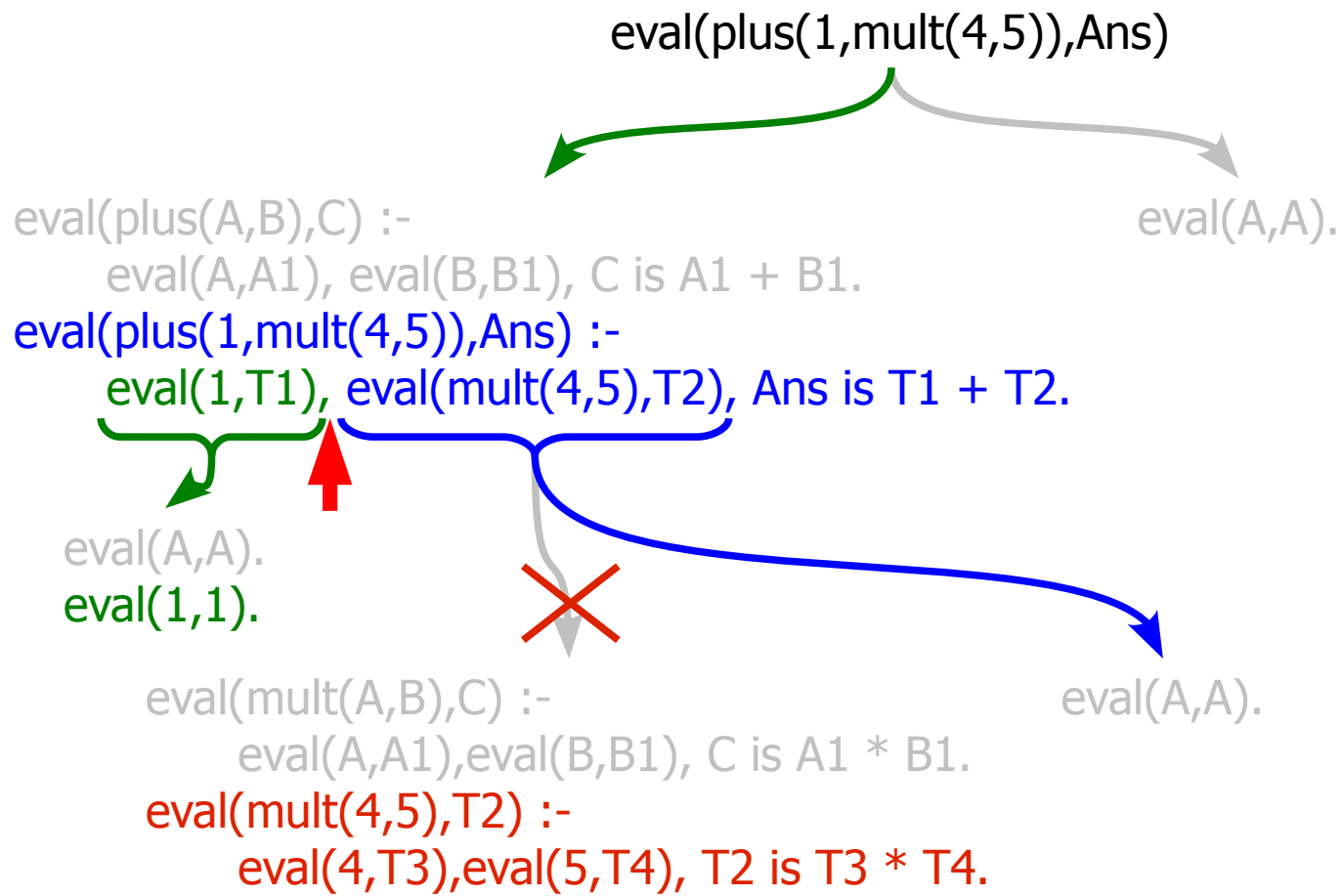
```



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.
```

```
eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.
```

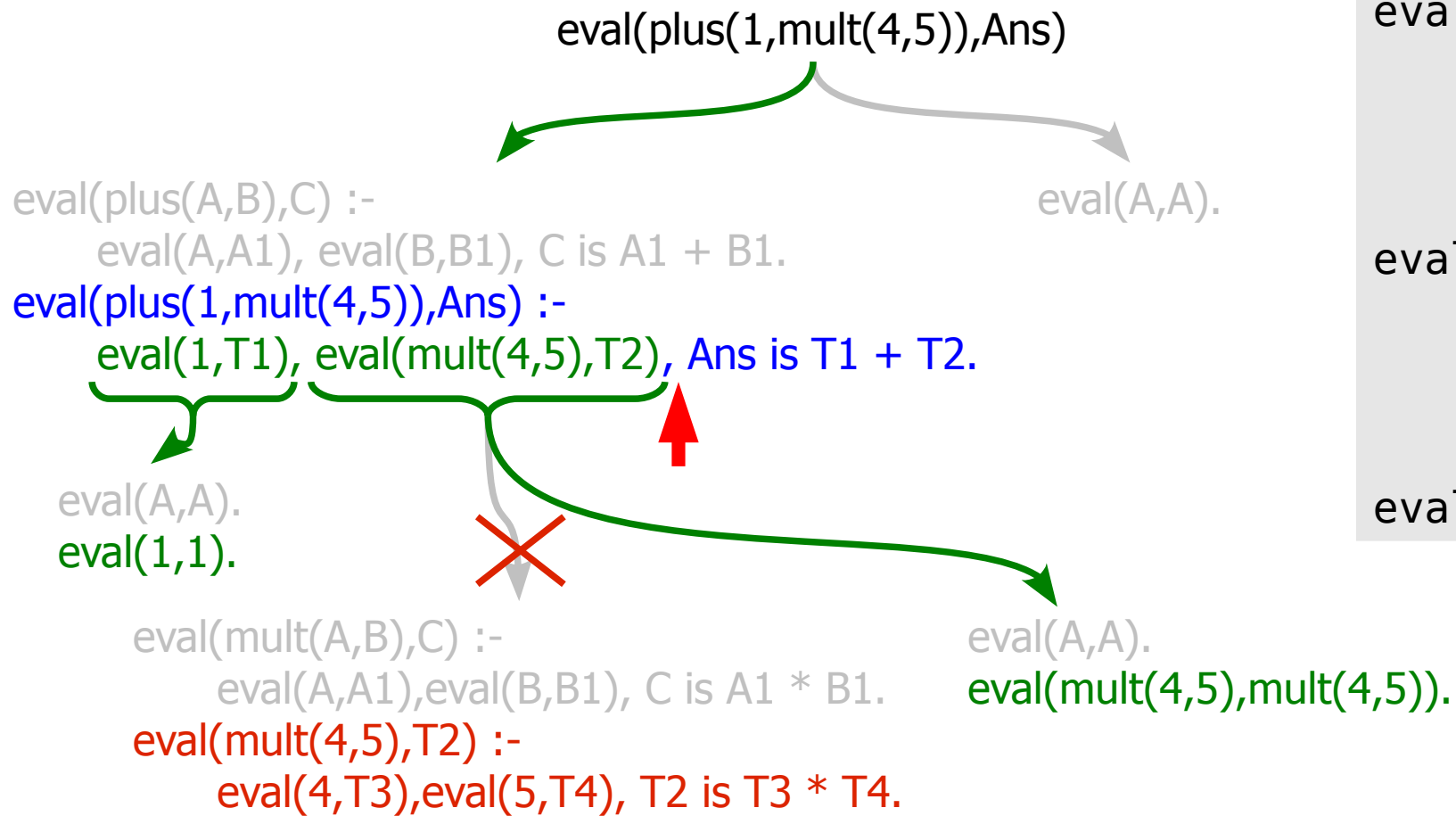
```
eval(A,A).
```



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.
```

```
eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.
```

```
eval(A,A).
```



```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.

eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.

eval(A,A).
```

eval(plus(1,mult(4,5)),Ans)
 eval(A,A).

eval(plus(A,B),C) :-
 eval(A,A1), eval(B,B1), C is A1 + B1.

eval(plus(1,mult(4,5)),Ans) :-
 eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
 eval(1,1).

eval(mult(A,B),C) :-
 eval(A,A1),eval(B,B1), C is A1 * B1.

eval(mult(4,5),T2) :-
 eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
 eval(mult(4,5),mult(4,5)).

```
eval(plus(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 + B1.
```

```
eval(mult(A,B),C) :-
  eval(A,A1),
  eval(B,B1),
  C is A1 * B1.
```

```
eval(A,A).
```

**Ouch... "is" can't handle
 the mult(4,5) term!**

Ans is 1 + mult(4,5)

(a) Eliminate spurious solutions by making your clauses orthogonal

Need to eliminate the (unwanted) choice point

A way to do this: make sure only one clause matches: **eval(A,A)** becomes **eval(gnd(A),A)**.

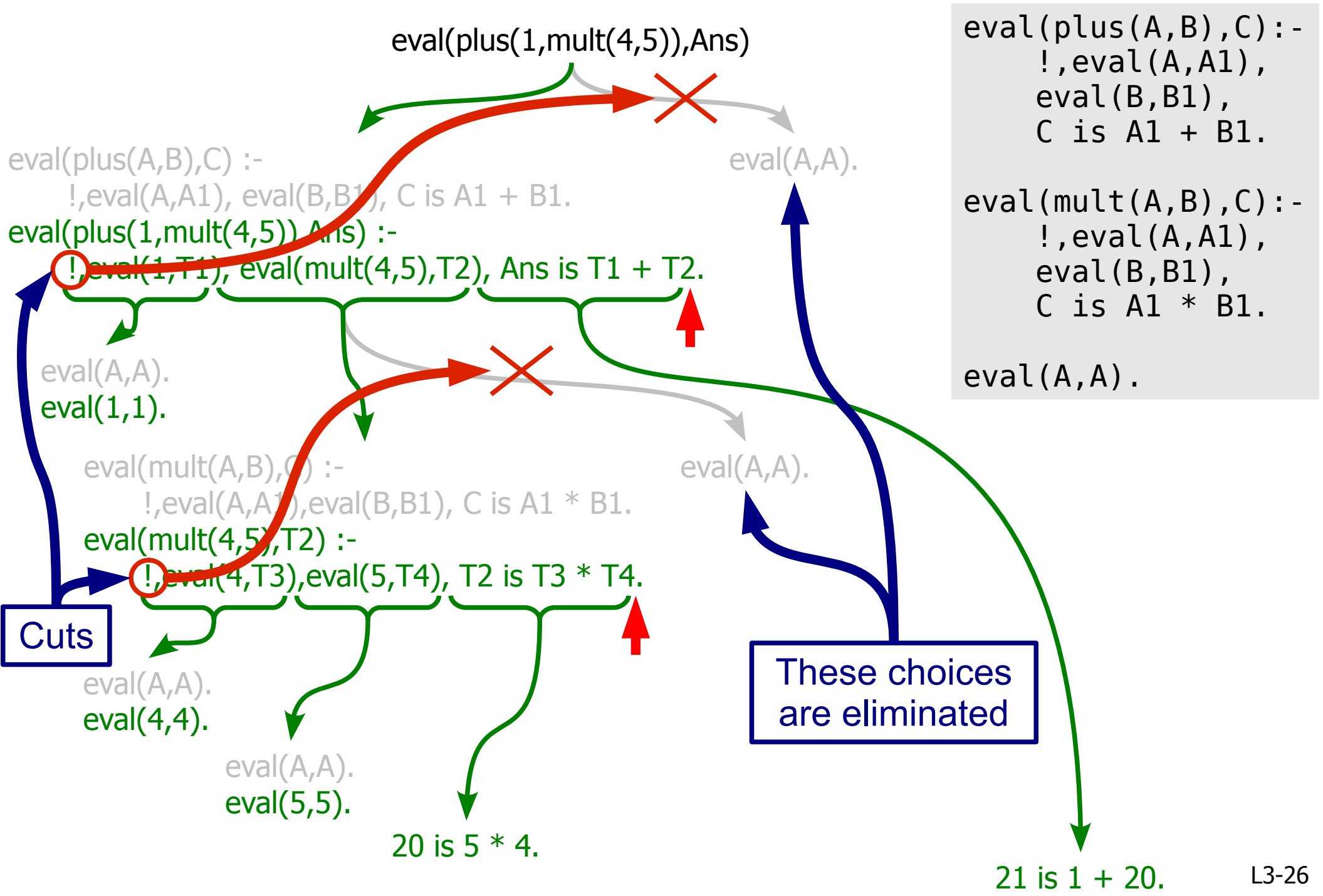
```
eval(plus(A,B),C) :- eval(A,A1),
                      eval(B,B1),
                      C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1),
                     eval(B,B1),
                     C is A1 * B1.
eval(gnd(A),A).
```


(b) Eliminate spurious solutions by explicitly discarding choice points

Alternatively we can tell Prolog to commit to its first choice and discard the choice point (p114/126)

We do this with the cut operator. Written: !

```
eval(plus(A,B),C) :- !,eval(A,A1),
                    eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- !,eval(A,A1),
                    eval(B,B1),
                    C is A1 * B1.
eval(A,A).
```



```

eval(plus(A,B),C) :-
    !,eval(A,A1),
    eval(B,B1),
    C is A1 + B1.

eval(mult(A,B),C) :-
    !,eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).

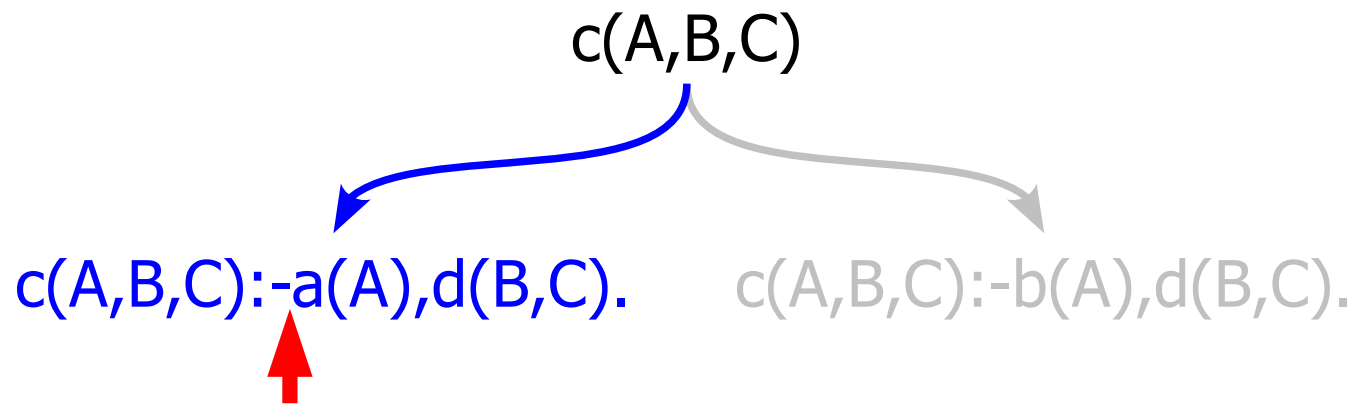
```

Cutting out choice

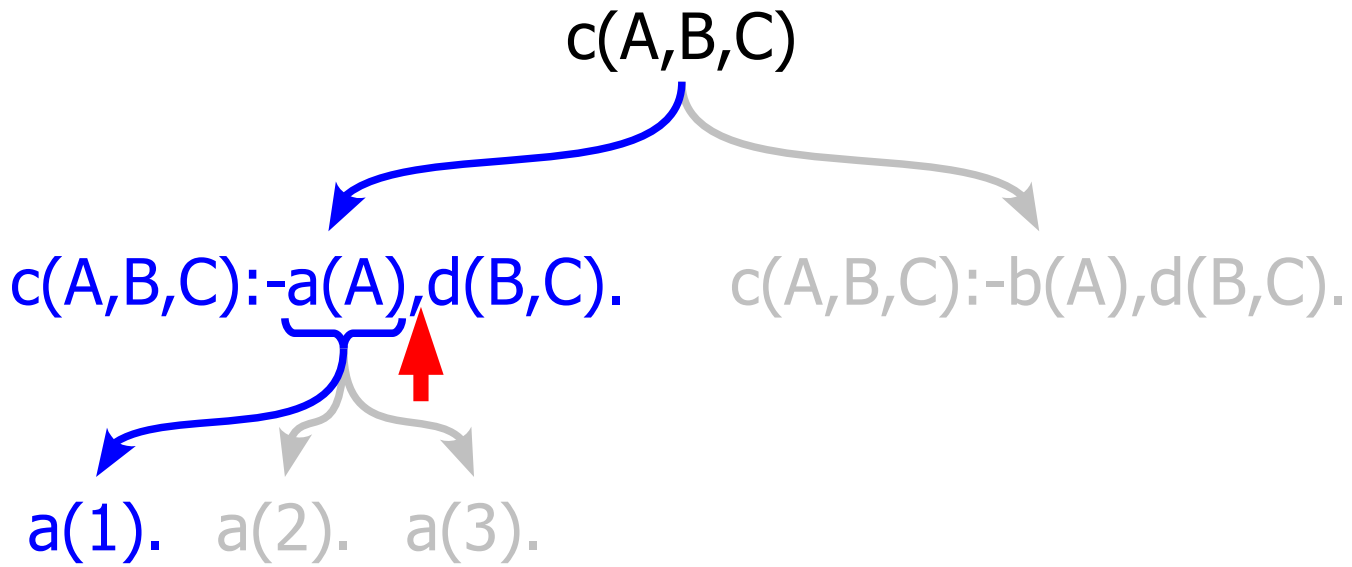
Whenever Prolog evaluates a cut it discards all choice points back to the parent clause

An example:

```
a(1).      c(A,B,C) :- a(A), d(B,C).
a(2).      c(A,B,C) :- b(A), d(B,C).
a(3).      d(B,C) :- a(B), !, a(C).
b(apple).  d(B,_) :- b(B).
b(orange).
```



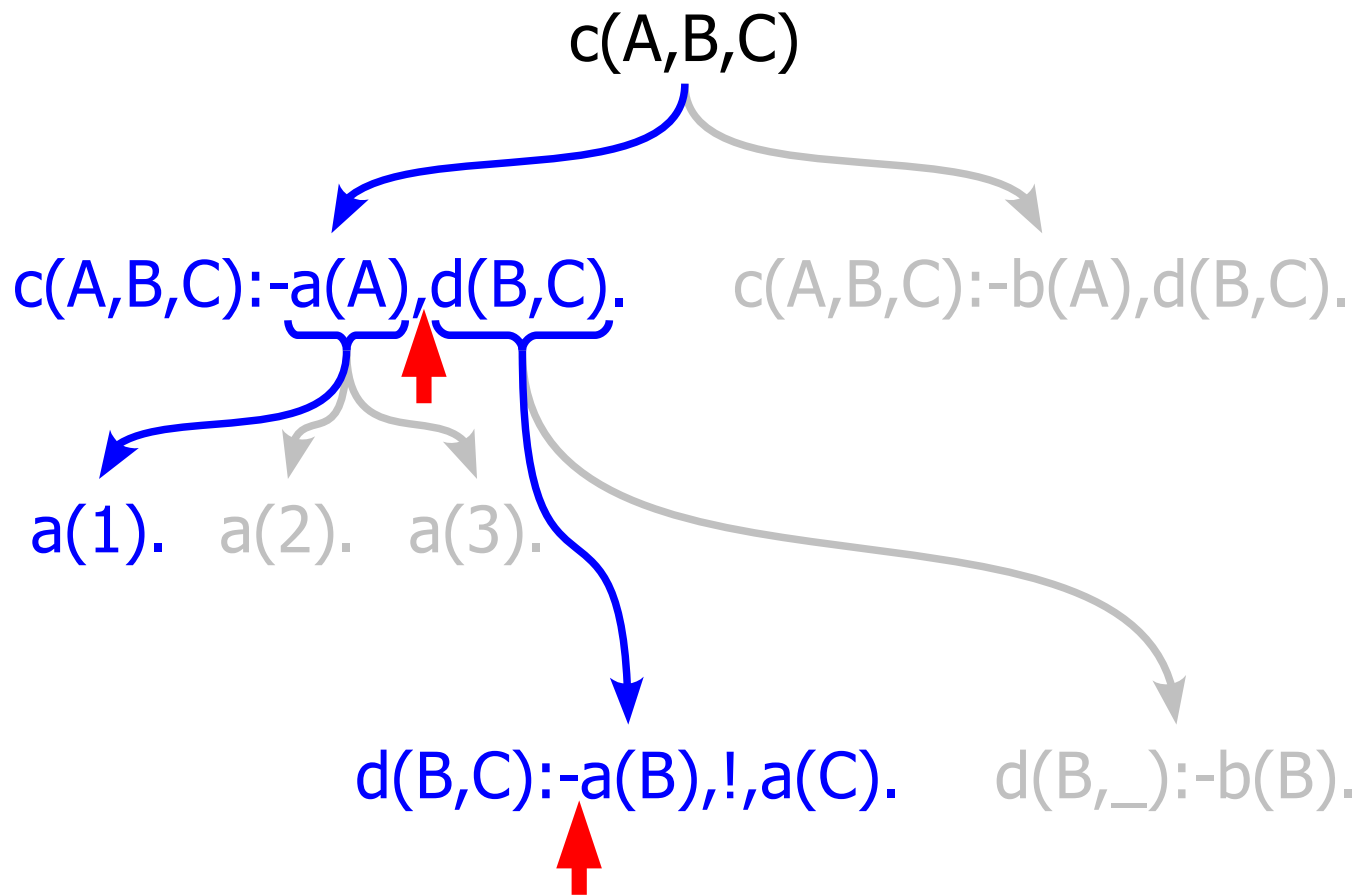
```
a(1).  
a(2).  
a(3).  
b(apple).  
b(orange).  
c(A,B,C):-a(A),d(B,C).  
c(A,B,C):-b(A),d(B,C).  
d(B,C):-a(B),!,a(C).  
d(B,_):-b(B).
```



```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

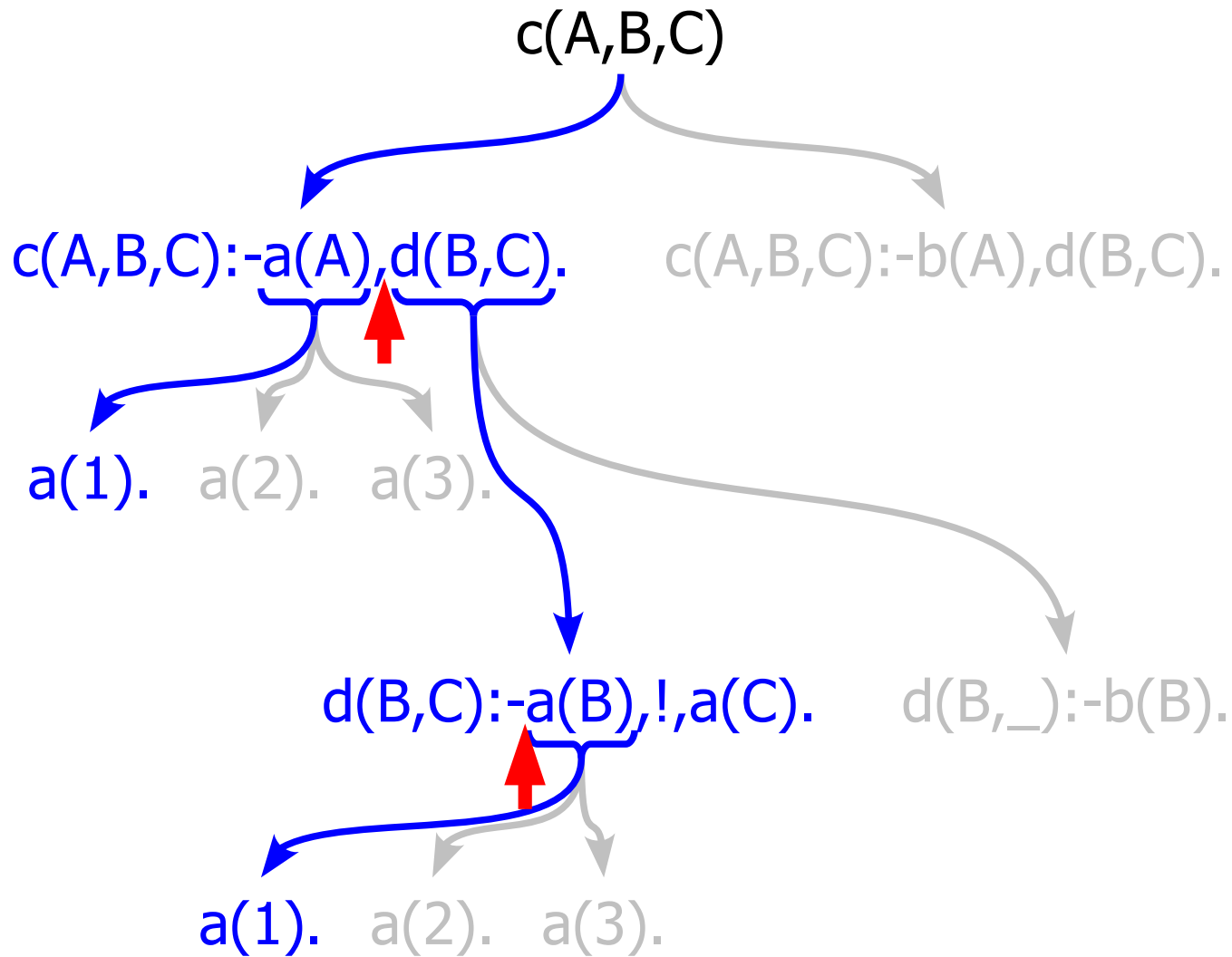
```



```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_) :- b(B).

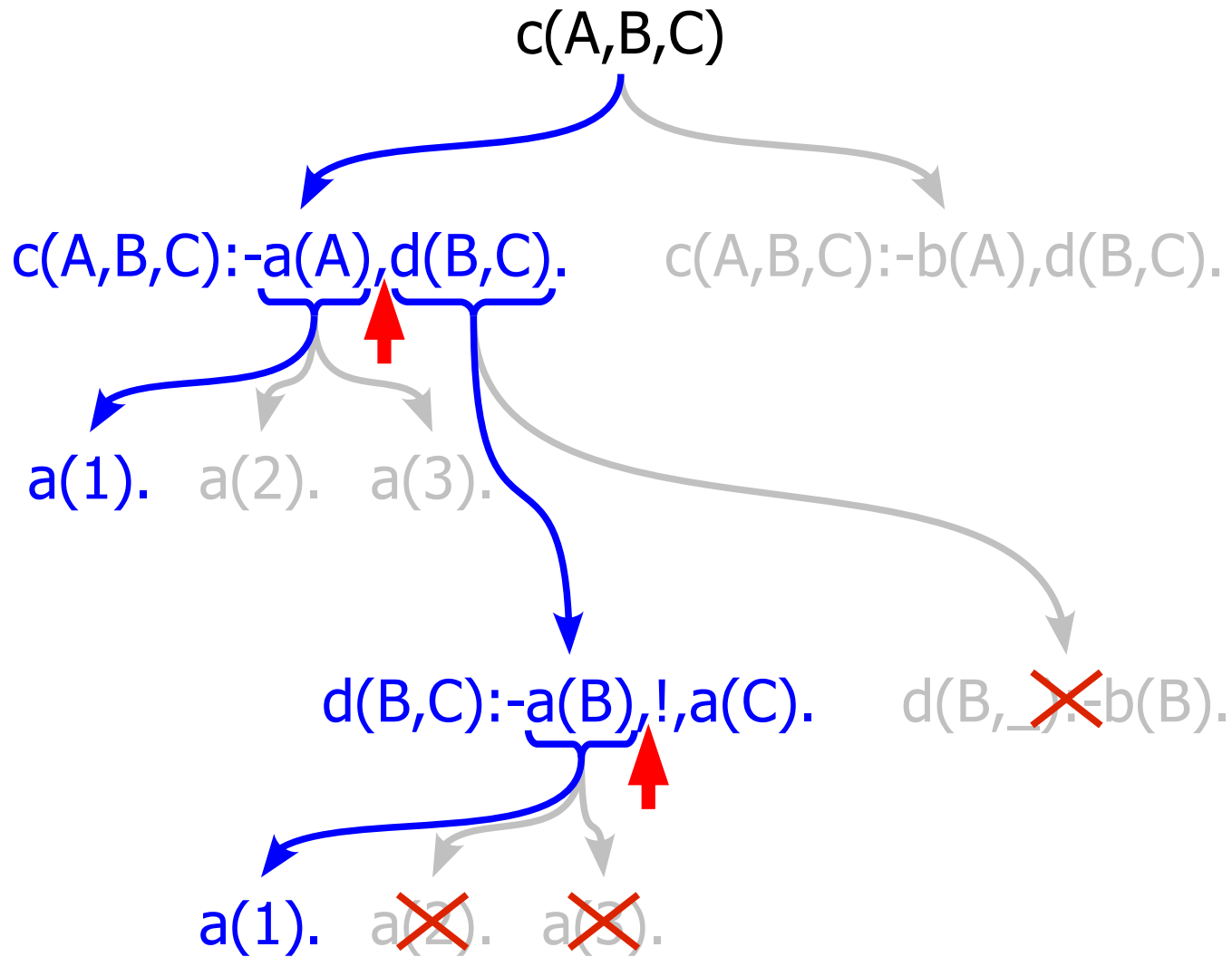
```



```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

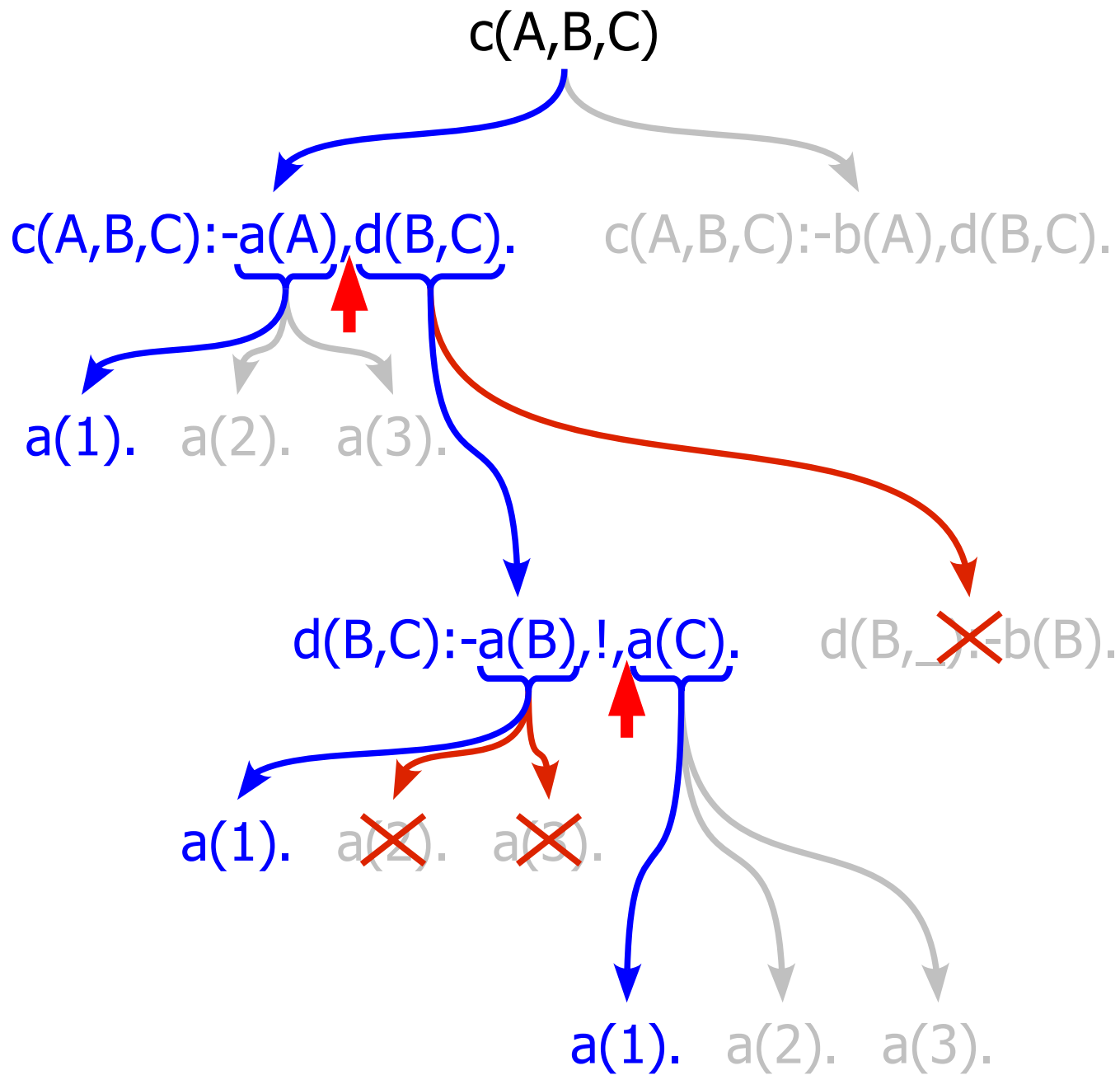
```



```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

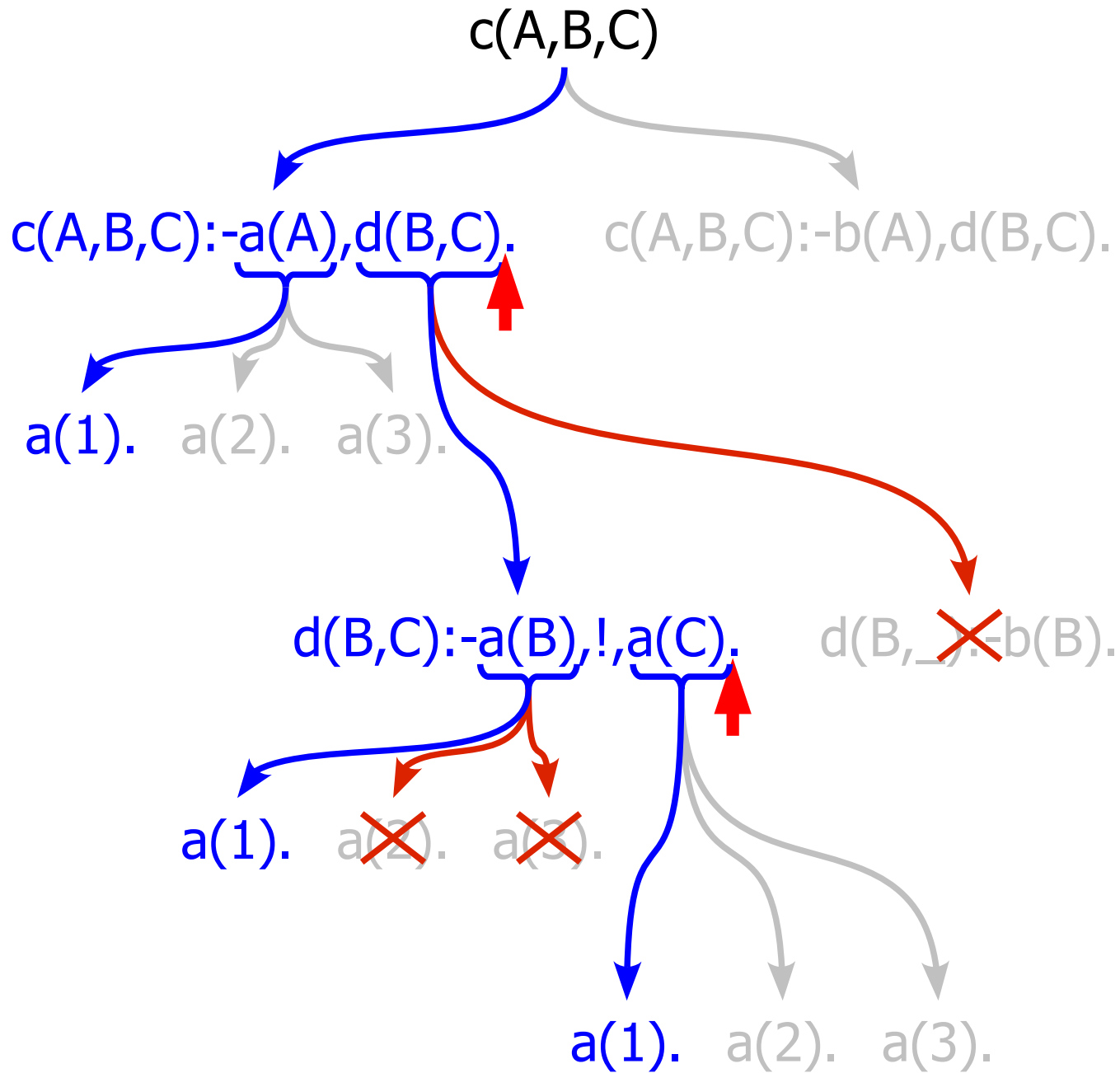
```

```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_) :-b(B).

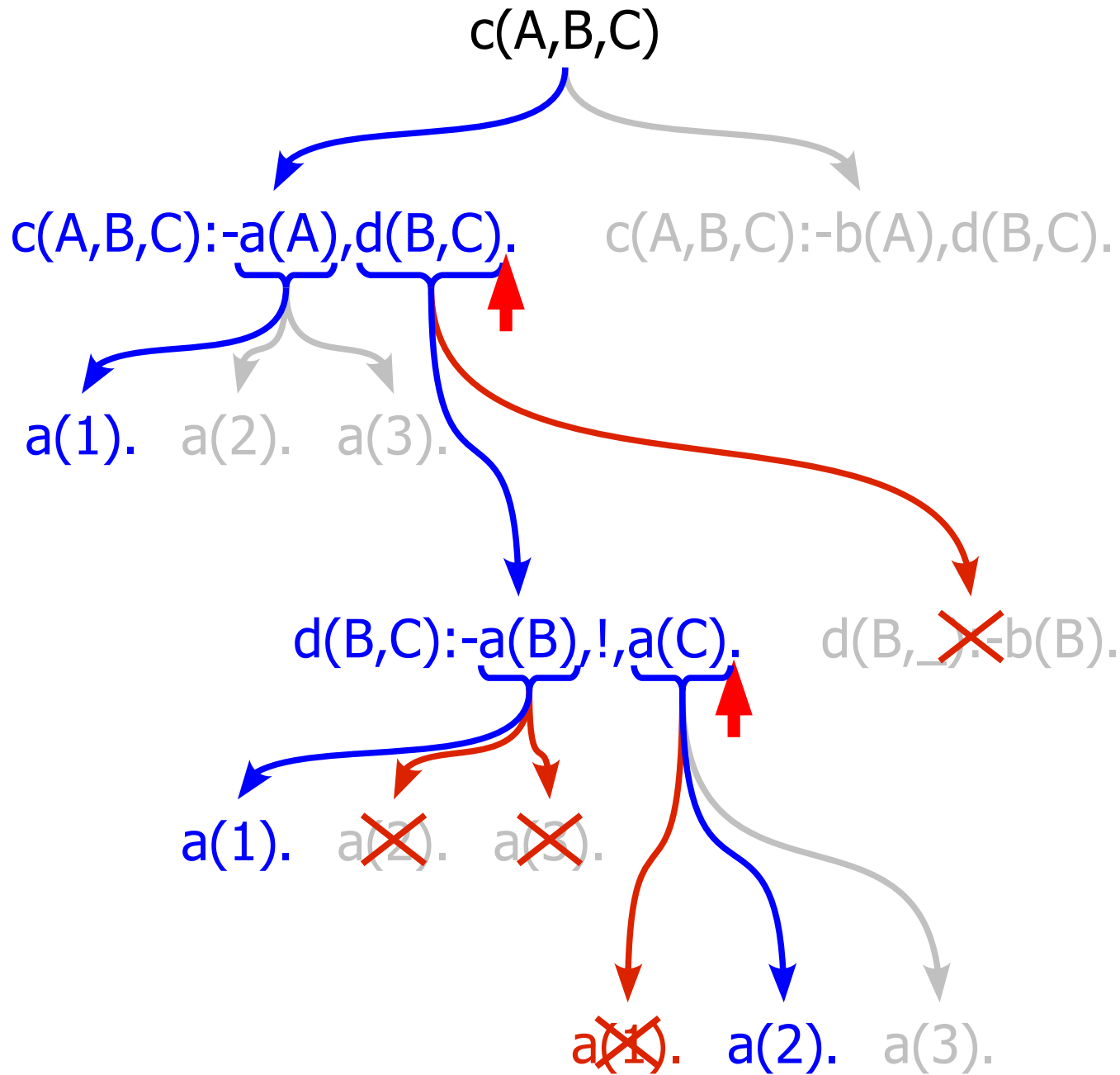
```



```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,-):-b(B).

```

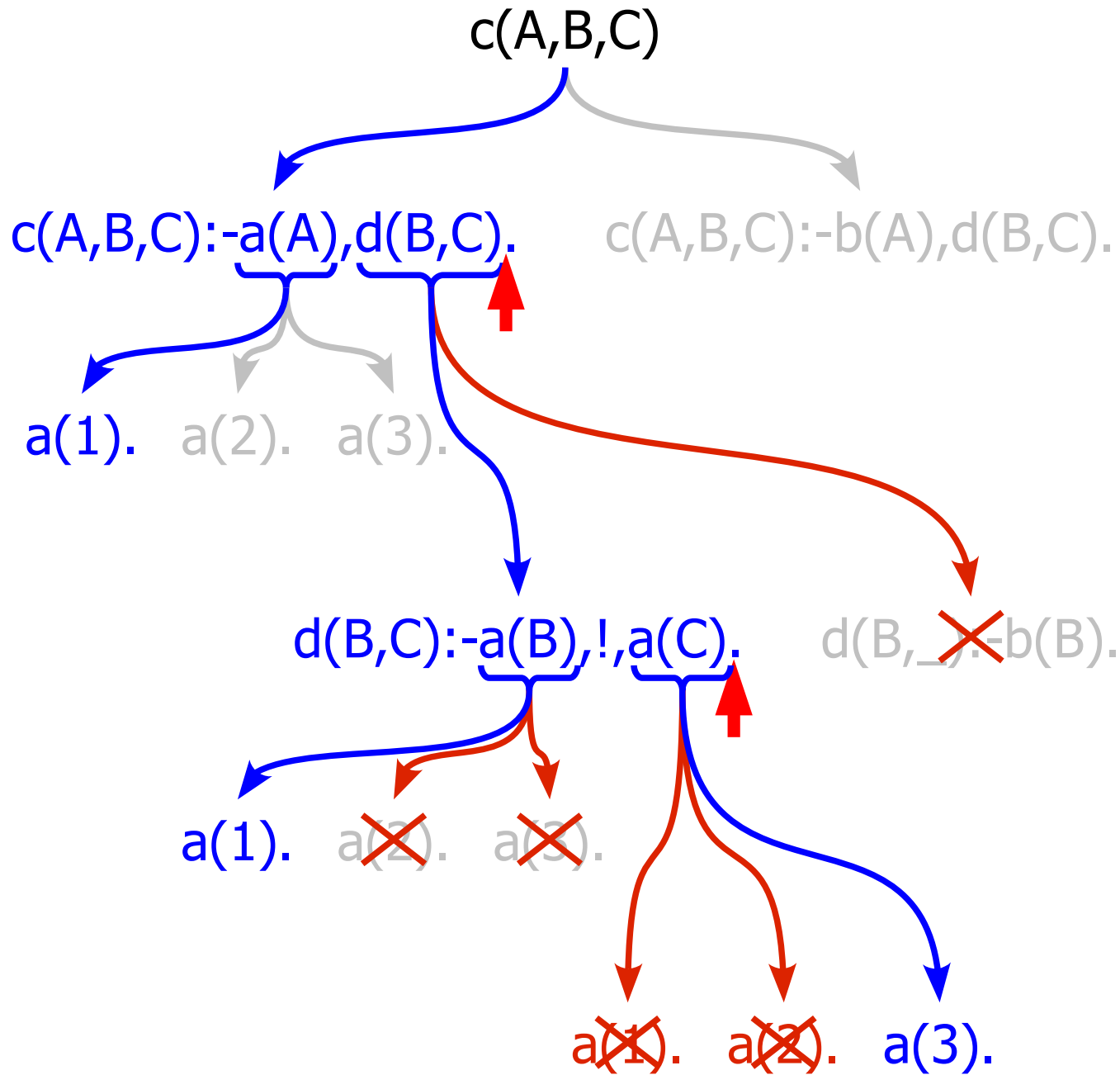


```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_) :- b(B).

```

Backtrack once

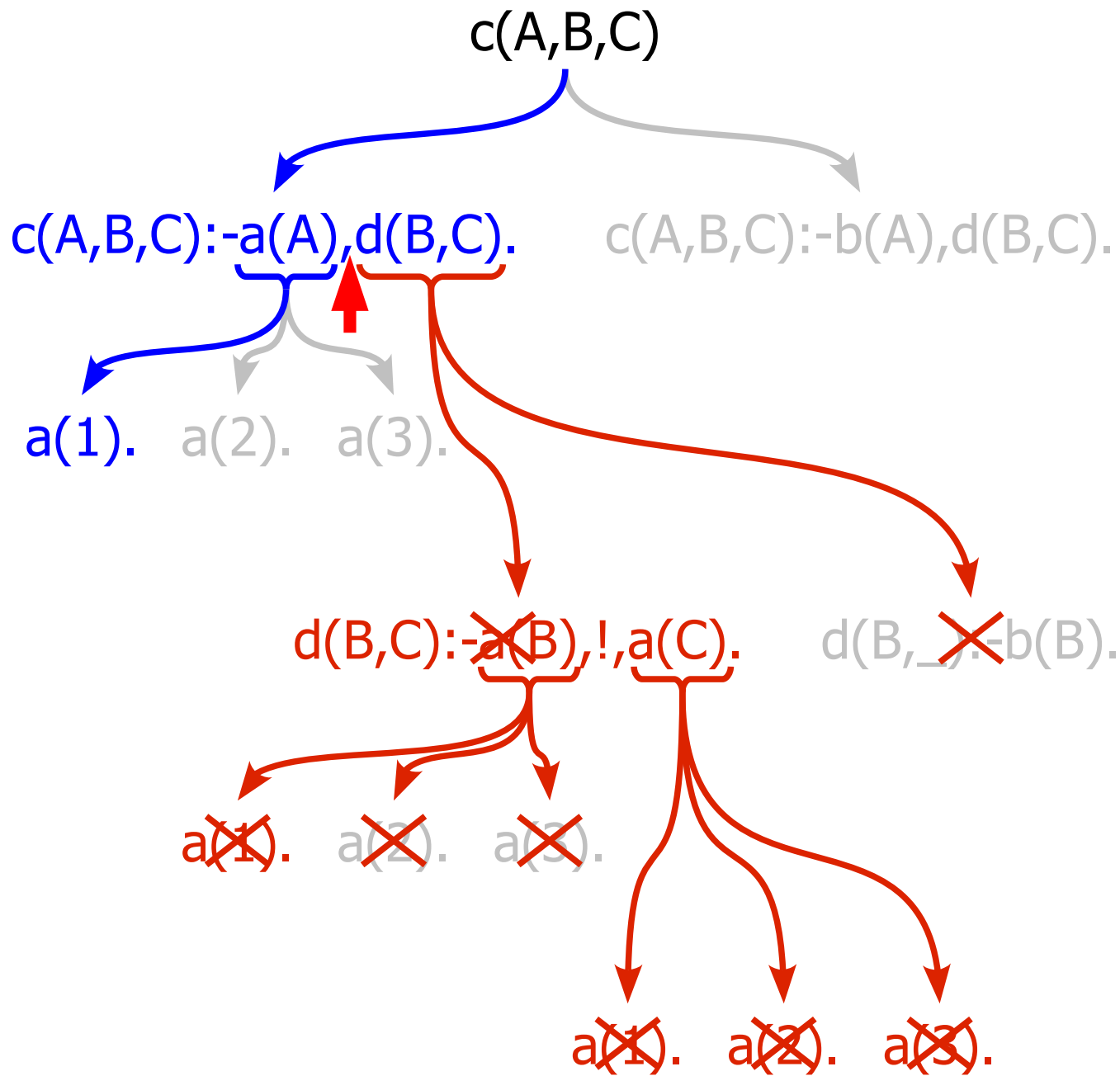


```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

```

Backtrack twice

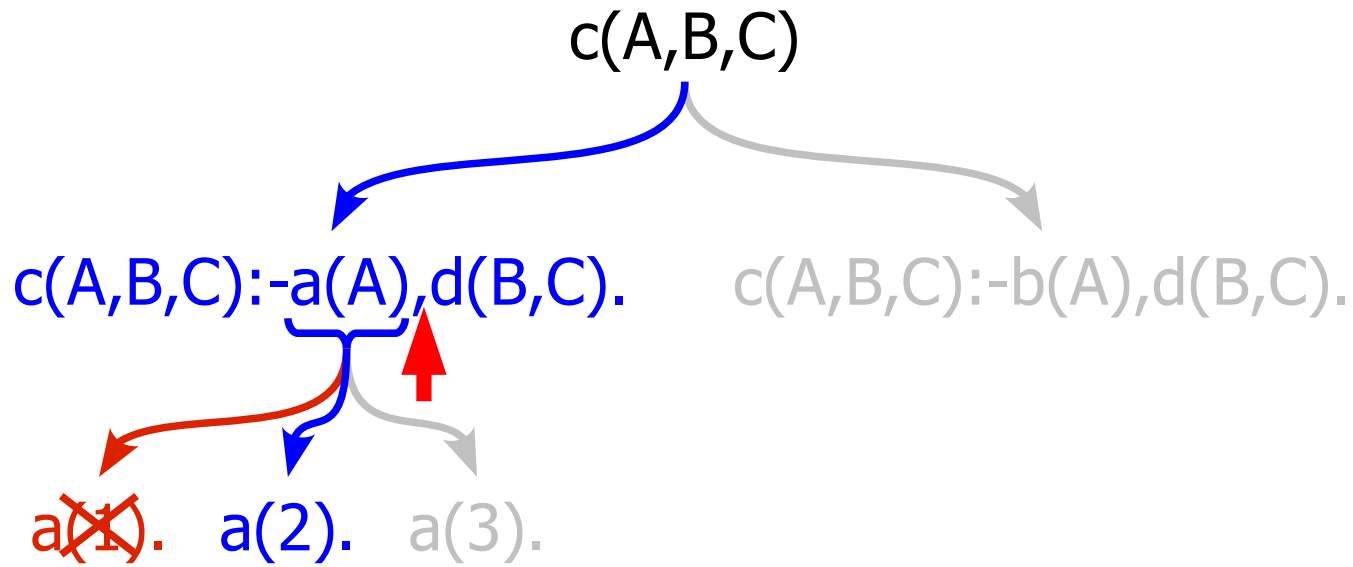


```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

```

Backtrack three times

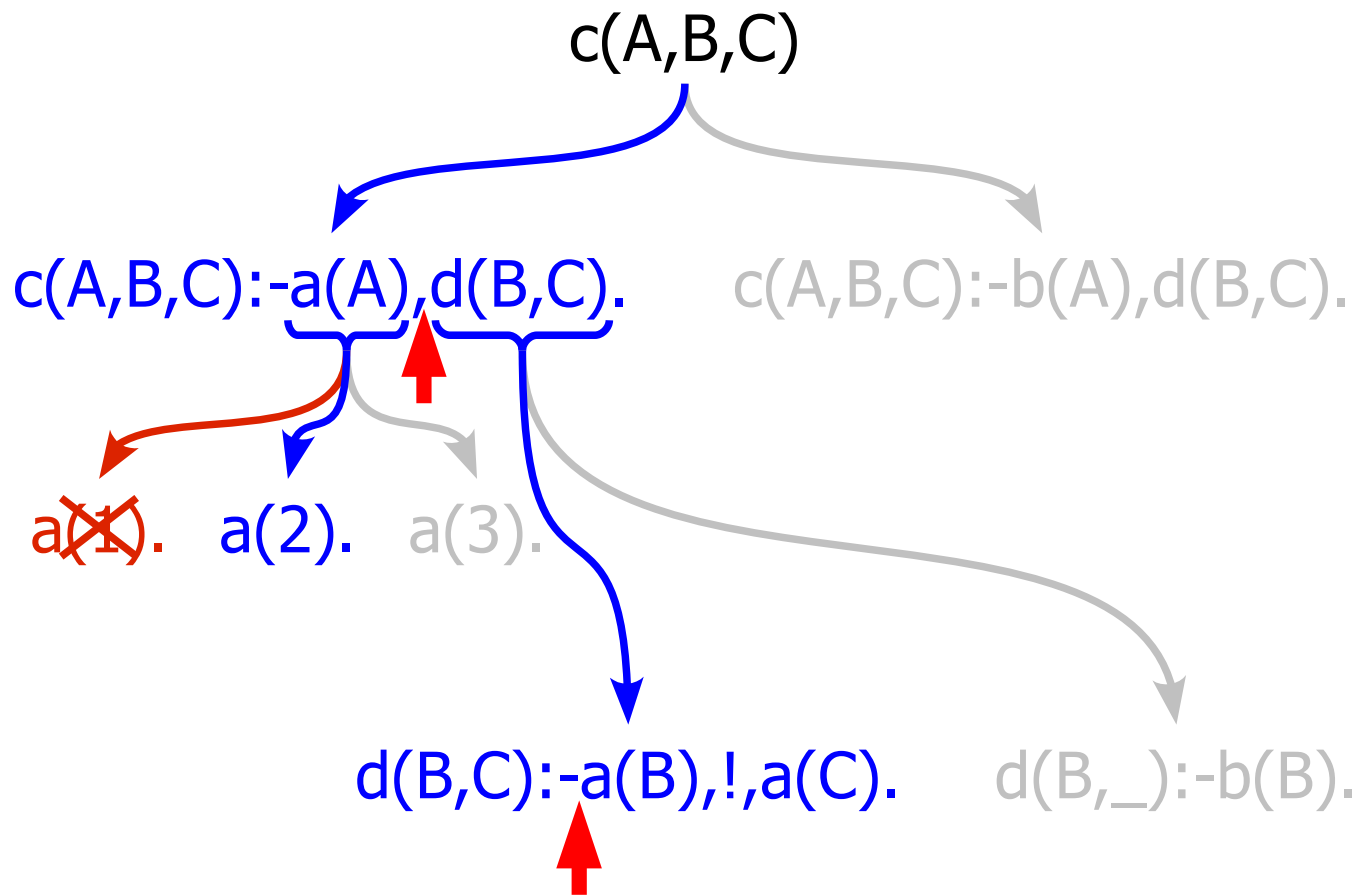


```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

```

First a/1 has other solutions



```

a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).

```

Can try to derive d/2 afresh...

Cut can change the logical meaning of your program

```
p :- a, b.  
p :- c.
```

$$p \Leftrightarrow (a \wedge b) \vee c$$

```
p :- a, !, b.  
p :- c.
```

$$p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$$

This is a **red** cut – **DANGER!** (p127/138)

Cut can be used for efficiency reasons

```
split([], [], []).  
split([H|T], [H|L], R) :- H < 5, split(T, L, R).  
split([H|T], L, [H|R]) :- H >= 5, split(T, L, R).
```

If the second clause succeeds the third cannot

- we don't need to keep a choice point
- yet the interpreter cannot infer this on its own

Cut can be used for efficiency reasons

```
split([], [], []).  
split([H|T], [H|L], R) :- H < 5, !, split(T, L, R).  
split([H|T], L, [H|R]) :- H >= 5, split(T, L, R).
```

Add a cut to make the orthogonality explicit

- This is a **green** cut – it just helps program execution go faster

We could go one step further at the expense of readability

```
split([], [], []).  
split([H|T], [H|L], R) :- H < 5, !, split(T, L, R).  
split([H|T], L, [H|R]) :- split(T, L, R).
```

The comparison in the third clause is no longer necessary

- but now each clause does not stand on its own
- stylistic preference – I avoid doing this

Programmers new to Prolog often cause determinism errors

Check that your predicates return the correct
number of answers!

- e.g. providing a correct answer multiple times
is likely to cause bugs that are difficult to find

Below is a predicate you can use in debugging

- (Note that findall is not discussed in lectures)

```
numsol(Predicate,NumberOfSolutions):-  
    findall(dummy,Predicate,AnsList),  
    length(AnsList,NumberOfSolutions).
```

Testing using the numsol/2 predicate

A warning will be generated about mergesplat/3.

- What is wrong with the mergesplat/3 predicate?

```
mergesplat([], [], []).
mergesplat(A, [], A).
mergesplat([], B, B).
mergesplat([A|As], [B|Bs], [A,B|Rest]) :-
    mergesplat(As, Bs, Rest).

:- numsol(merge([1,2], [a,b], _), 1).
:- numsol(mergesplat([1,2], [a,b], _), 1).
```

Cut gives us more expressive power

```
isDifferent(A,A) :- !,fail.  
isDifferent(_,_).
```

isDifferent(A,B) is true iff A and B do not unify

Questions that you should be able to answer:

- Is this a red or a green cut?
- How can you define the fail/0 predicate?

Using cut, we can implement “not” (Negation by failure)

```
not(A) :- A,!,fail.  
not(_).
```

not(A) is true if A cannot be shown to be true

- This is **negation by failure** (p124/135)

Negation by failure is based on the **closed world assumption**: (p129/138)

Everything that is true in the “world” is stated (or can be derived from) the clauses in the program

Negation Example

```
good_food(theWrestlers).  
good_food(theCambridgeLodge).  
expensive(theCambridgeLodge).  
  
bargain(R) :- good_food(R),  
              not(expensive(R)).
```

we can ask:

- bargain(R)

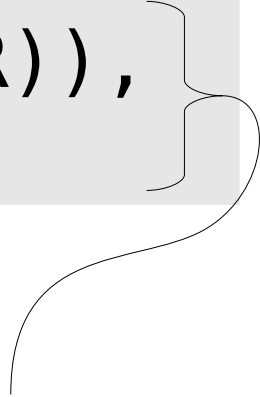
and Prolog replies:

- R = theWrestlers

Negation Gotcha!

```
good_food(theWrestlers).  
good_food(theCambridgeLodge).  
expensive(theCambridgeLodge).
```

```
bargain(R) :- not(expensive(R)),  
              good_food(R).
```



we can ask the same query:

- bargain(R)

and Prolog replies:

- false.

Clause body terms
have been
swapped around!

Why?

```
good_food(theWrestlers).  
good_food(theCambridgeLodge).  
expensive(theCambridgeLodge).  
  
bargain(R) :- not(expensive(R)),  
              good_food(R).
```

Prolog first tries to find an R such that `expensive(R)` is true.

- therefore `not(expensive(R))` will fail if there are **any** expensive restaurants

We sometimes identify the way to use parameters of a rule

Prolog's non-logical properties can make it important whether or not an argument to a predicate is bound

% indicates a comment to the end of that line

```
% this comment in some hypothetical code is  
% describing how to query myrule(+A,+B,-C,-D)
```

The convention for comments about rule parameters:

+X is a ground term (must be instantiated)

-X is a variable term (must be unbound)

?X means it does not matter (roughly)

Query "myrule" with two ground (input) terms A and B and two variable (output) terms C and D

Prolog variables and quantifiers

When R is not bound, quantifiers need attention

`expensive(R)`

- “**There exists** an R that is expensive”.

`not(expensive(R))`

- “**There does not exist** an R that is expensive”.
- In other words, “**for all** R, not expensive(R)”.

Databases

Information can be stored as tuples in Prolog's internal database

```
tName(dme26, 'David Eyers').  
tName(awm22, 'Andrew Moore').
```

```
tGrade(dme26, 'IA', 2.1).  
tGrade(dme26, 'IB', 1).  
tGrade(dme26, 'II', 1).  
tGrade(awm22, 'IA', 2.1).  
tGrade(awm22, 'IB', 1).  
tGrade(awm22, 'II', 1).
```

Databases

We can now write a program to find all names:

```
qName(N) :- tName(_,N).
```

Or a program to find the full name and all grades for dme26.

```
qGrades(F,C,G) :- tName(I,F), tGrade(I,C,G).
```

Further exercises are in the problem sheet...