Logic Programming

Part 3: All the way to Prolog

Hans Hüttel
Programming Paradigms, Autumn 2020
Aalborg University
(incorporating material by Magnus Madsen and Claus Brabrand)

Learning goals

- To understand the syntax of Prolog and how it extends that of Datalog
- To understand the difference between the semantics of Prolog and the semantics of Datalog
- To be able to model programming problems in Prolog, and in particular…
- To be able to write list predicates in Prolog and to be able to relate them to list functions in Haskell
- To understand the proof search carried out by the Prolog interpreter

Prolog Coffee Bar

Spisesteder Cafeer

Adresse

Høkerboderne 16

Kontakt

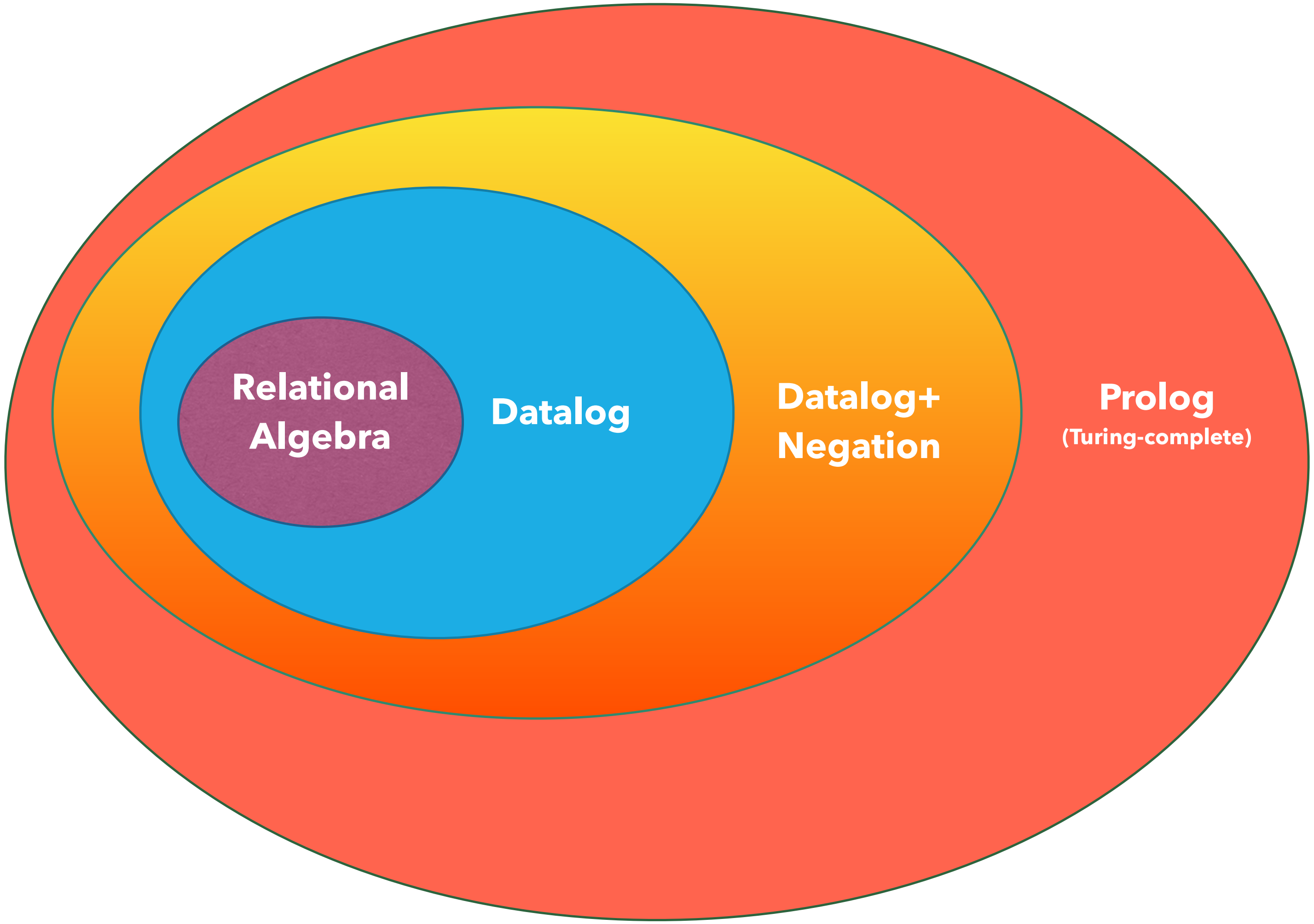
E-mail: hello@prologcoffeebar.com

Website

www.prolo

Spar pen

The logic programming family



The abstract syntax of Prolog

The formation rules in the abstract syntax of Prolog are

$$
\begin{array}{rll}
P & ::= C_1, \ldots, C_n & \text{Programs} \\
C & ::= A_0 \Leftarrow A_1, \ldots, A_n & \text{Clauses} \\
A & ::= p(t_1, \ldots, t_n) \mid \mathbf{not}\ p(t_1, \ldots, t_n) & \text{Atoms}, n \geq 0 \\
t & ::= k \mid x \mid f(t_1, \ldots, t_n) & \text{Terms}, n \geq 0
\end{array}
$$

Here f ranges over a finite set of *function symbols*. The rest is as in Datalog with negation, except that we now also allow facts that contain variables.

But it is the new syntax for terms that makes the big difference!

Another difference is that *the order of the clauses matters in Prolog* (this is not the case in Datalog). This is because the semantics of Prolog is that of goal-directed search.

Datalog and Prolog

- Datalog is a sublanguage of Prolog, but…
- Datalog corresponds to polynomial-time computability only, *Prolog is Turing-complete*
- Datalog queries always terminate, *Prolog queries do not always terminate*

The semantics of Prolog is different

- Every Datalog program has a minimal model that consists of only finitely many facts; Prolog programs can have *infinite models*
- The fixed-point semantics for Prolog is not useful for implementation purposes.
- We think of the execution of a Prolog query as goal-directed proof search. This is why the order of clauses in a Prolog program matters.

The difference illustrated

What would Datalog tell us? What would Prolog tell us?

```
edge(a,b).

edge(b,c).

edge(c,d).

path(X,Z) :- path(X,Y), path(Y,Z).

path(X,Y) :- edge(X,Y).
```

The power of Prolog

In Prolog, we can use function symbols to build composite terms.

```
monarch(queen(margrethe),denmark).

monarch(queen(elizabeth),unitedkingdom).

monarch(king(carlgustaf),sweden).

monarch(king(willemalexander),netherlands).

queendom(X) :- monarch(queen(Q),X).
```

Data types in Prolog

Haskell vs. Prolog

Remember algebraic datatypes in Haskell? We wrote

```
data Nat = Zero | Succ Nat
```

In Prolog we can write the clauses

```
nat(zero).

nat(succ(X)) :- nat(X).
```

But Prolog has no type system 😫, so *not everything is the same*. `nat` is a predicate symbol in Prolog, while **`Nat`** is a type constructor in Haskell.

Prolog meets Herbrand

(Actually, this is hr. Steffen Brandt.)

Prolog meets Herbrand

The program

```
nat(zero).
nat(succ(X)) :- nat(X).
```

has the *infinite Herbrand universe*

{ `zero`, `succ(zero)`, `succ(succ(zero))`,...}

and the *infinite Herbrand base*

{ `nat(zero)`, `nat(succ(zero))`, `nat(succ(succ(zero)))`,...}

What happens then?

- If the Herbrand base of our program is infinite, an interpretation can be infinite, and so can a model.
- Queries become more difficult to handle: If a model is infinite, we cannot compute it using the iterative fixed-point algorithm that we saw for Datalog; the algorithm will never terminate!
- Goal-directed search is a much better option, *though it may also fail to terminate* 😕

Proof search for beginners

- Is `4` a natural number?
- Is `mary` a natural number?

“Open” facts with variables.

- In Prolog we allow facts that contain variables.
- This allows us to write universal quantifiers: that a proposition holds for all terms.
- Let us use this to define ordering and addition for the natural numbers.
- In Haskell, functions were defined as equations of the form

$$f\ \mathsf{pattern} = e$$

- In Prolog, we represent functions as *predicates*

$$p_f(\mathsf{pattern}, e) : - \ldots$$

The good old days of Haskell

```
data Nat = Zero | Succ Nat deriving Show

leq (Succ x) (Succ y) = leq x y
leq zero y = True

add x Zero = x
add (Succ x) y = Succ (add x y)
add x y = add y x
```

Our new life with Prolog

```
nat(zero).
nat(succ(X)) :- nat(X).

leq(zero, Y) :- nat(Y).
leq(succ(X), succ(Y)) :- leq(X, Y), nat(X), nat(Y).

add(X, zero, X) :- nat(X).
add(succ(X), Y, succ(R)) :- add(X, Y, R), nat(X),
                            nat(Y), nat(R).
add(X,Y,F) :- add(Y,X,F).
```

Some actual arithmetic in Prolog

- We do not *have to* declare the natural numbers; Prolog has the usual arithmetic operations.
- The **`is`** predicate lets us compare values.
- Here is the factorial function defined by the predicate `fact`:

```
fact(0,1).

fact(N,V) :- N1 is N-1, fact(N1,V1), V is V1*N.
```

Prolog has lists, too

- The cons constructor is called | in Prolog
- The list notation `[1,2,3]` is short for `[1 | [2 [3 | []]]]`

Some *predicates* on lists

`member(X,L)`	`X` is an element of the list `L`
`append(L1,L2,L3)`	`L1` and `L2` append to `L3`
`reverse(L1,L2)`	`L2` is the reverse of `L1`
`length(L1,N)`	`L1` has length `N`
`insertionsort(L1,L2)`	`L2` is the sorted version of `L1` using insertion sort

Proof search in Prolog

- We have already seen the proof-theoretic semantics for Datalog.
- We use a similar approach to determine if a goal is a consequence of the facts and rules in a Prolog program.
- At any stage of the computation, the *resolvent* is the set of subgoals.
- At first, the resolvent is the overall goal, but as we search for a solution, the resolvent becomes a set of subgoals of a rule.
- The implementation always searches the new subgoals from the rule *from left to right*.

Matching a goal with a clause

Suppose we have the clauses

```
nat(zero).
nat(succ(X)) :- nat(X).
```

and the goal `nat(succ(succ(zero)))`. Which of these clauses will match the rule? We match the goal with the head of each clause from the first to the last.

Can it be the first rule? Then the equation `nat(succ(succ(zero)) = nat(zero)` should hold. Clearly it cannot.

Can it be the second rule? Then the equation `nat(succ(succ(zero)) = nat(succ(X))` should hold. *And it can*, if we substitute `X` by `succ(zero)`.

Unification

- Unification succeeds immediate if we match a variable with a term

  `X` and `nat(zero)` unify, returning the substitution $[\mathtt{X} \mapsto \mathtt{nat}(\mathtt{zero})]$

- Unification will fail, if the function symbols are different.

  `succ(X)` and `nat(zero)` do not unify.

- Unification will fail if the two terms mention the same variable

  `succ(X)` and `succ(succ((X))` do not unify.

Unification

- Solving equations of this form is called *unification*.
- A substitution σ is a mapping from variables to terms.
- The value $\sigma(t)$ for the term t is what we get by replacing every variable x in t by $\sigma(x)$
- Given two terms t_1 and t_2 with variables, the unification algorithm will attempt to find a substitution σ such that $\sigma(t_1) = \sigma(t_2)$
- In our case, we had that $\sigma(\mathtt{X}) = \mathtt{succ(zero)}$.

The unification algorithm in Haskell pseudocode

$$
\begin{aligned}
\texttt{UNIFY}\; x\; t &= [x \mapsto t] \quad \text{if } x \text{ does not occur in } t \\
\texttt{UNIFY}\; f(t_1, \ldots, t_n)\; f(u_1, \ldots, u_n) &= \texttt{UNIFY}\; \sigma_{n-1}(t_n)\; \sigma_{n-1}(u_n') \\
&\ \text{where} \\
&\ \sigma_0 = \texttt{UNIFY}\; t_1\; u_1 \\
&\ \ldots \\
&\ \sigma_i = \texttt{UNIFY}\; \sigma_{i-1}(t_i), \sigma_{i-2}(u_1\ ) \\
&\ \text{for } i \geq 1 \\
\texttt{UNIFY}\; t\; t &= \mathsf{id} \\
\texttt{UNIFY}\; t\; t' &= \mathsf{fail} \quad \text{otherwise}
\end{aligned}
$$

The most complicated case is the second one – we unify the subterms from left to right, applying the substitutions that we discover as we proceed.

The proof search algorithm in a Prolog interpreter

- **Input**: A goal G and a program P
- **Output**: An instance of G that is a logical consequence of P – or **fail**

Initialize the resolvent R to the set $\{G\}$

while $R \neq \emptyset$ **do**

    choose a goal A from the resolvent R

    choose a (renamed) clause $A' \Leftarrow B_1, \ldots, B_n$ *(in order from first to last)*

    such that $\sigma = \mathtt{unify}\ A\ A'$

    if no such clause exists, then exit the loop

    let the new resolvent be $(R \setminus \{A\}) \cup \{B_1, \ldots, B_n\}$

    apply the substitution σ to the resolvent and to G

$R = \emptyset$

if $R = \emptyset$, then output G else **fail**

Searching for `append`

Searching for `append`

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

```
append([a,b,c],[d,e,f],_G1)
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

```
append([a,b,c],[d,e,f],_G1)

_G1 = [a|_G2]      rule

append([b,c],[d,e,f],_G2)
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

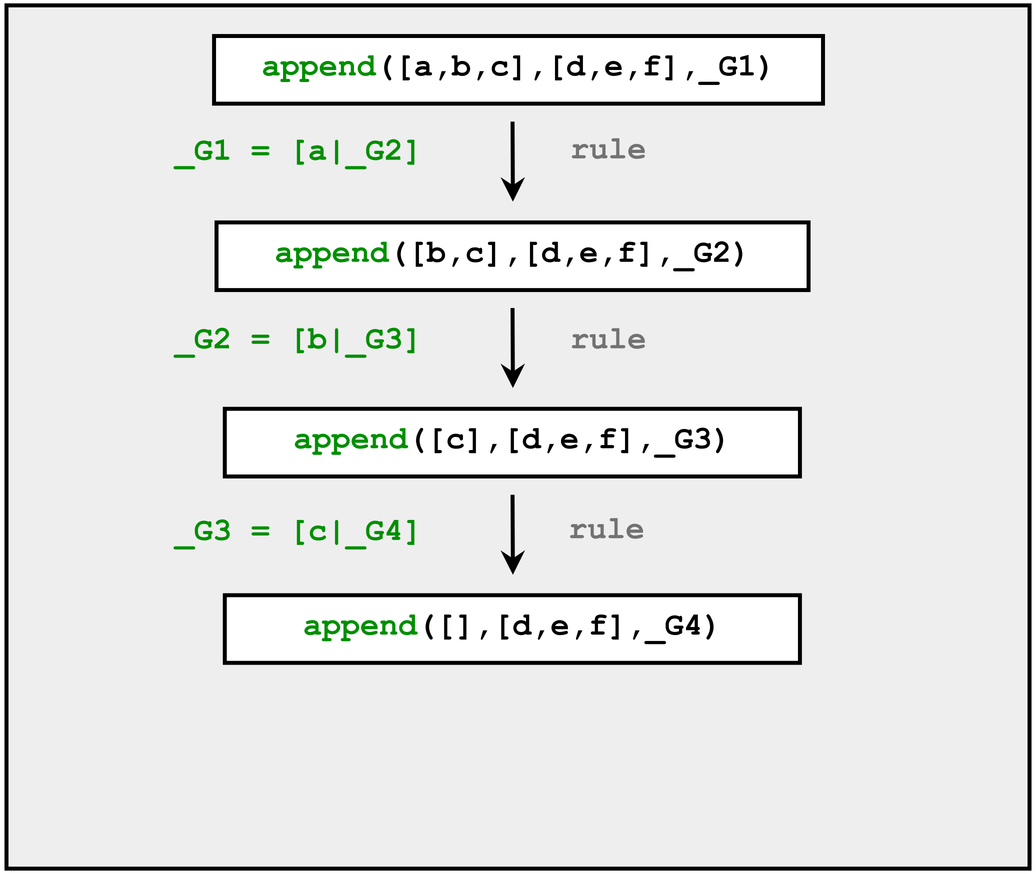
```
?- append([a,b,c], [d,e,f], R)
```

```
append([a,b,c],[d,e,f],_G1)
        |
_G1 = [a|_G2]   ↓   rule
        |
append([b,c],[d,e,f],_G2)
        |
_G2 = [b|_G3]   ↓   rule
        |
append([c],[d,e,f],_G3)
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```



Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

```
             append([a,b,c],[d,e,f],_G1)
_G1 = [a|_G2]          ↓   rule
             append([b,c],[d,e,f],_G2)
_G2 = [b|_G3]          ↓   rule
             append([c],[d,e,f],_G3)
_G3 = [c|_G4]          ↓   rule
             append([],[d,e,f],_G4)
_G4 = [d,e,f]          ↓   axiom
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

```
append([a,b,c],[d,e,f],_G1)
    _G1 = [a|_G2]      rule
append([b,c],[d,e,f],_G2)
    _G2 = [b|_G3]      rule
append([c],[d,e,f],_G3)
    _G3 = [c|_G4]      rule
append([],[d,e,f],_G4)
    _G4 = [d,e,f]      axiom
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

```
append([a,b,c],[d,e,f],_G1)
_G1 = [a|_G2]     rule
append([b,c],[d,e,f],_G2)
_G2 = [b|_G3]     rule
append([c],[d,e,f],_G3)
_G3 = [c|_G4]     rule
append([],[d,e,f],_G4)
_G4 = [d,e,f]     axiom
```

```
append([],[d,e,f],[d,e,f])
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

```
append([a,b,c],[d,e,f],_G1)
_G1 = [a|_G2]          rule
append([b,c],[d,e,f],_G2)
_G2 = [b|_G3]          rule
append([c],[d,e,f],_G3)
_G3 = [c|_G4]          rule
append([],[d,e,f],_G4)
_G4 = [d,e,f]          axiom
```

```
append([c],[d,e,f],[c,d,e,f])
append([],[d,e,f],[d,e,f])
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

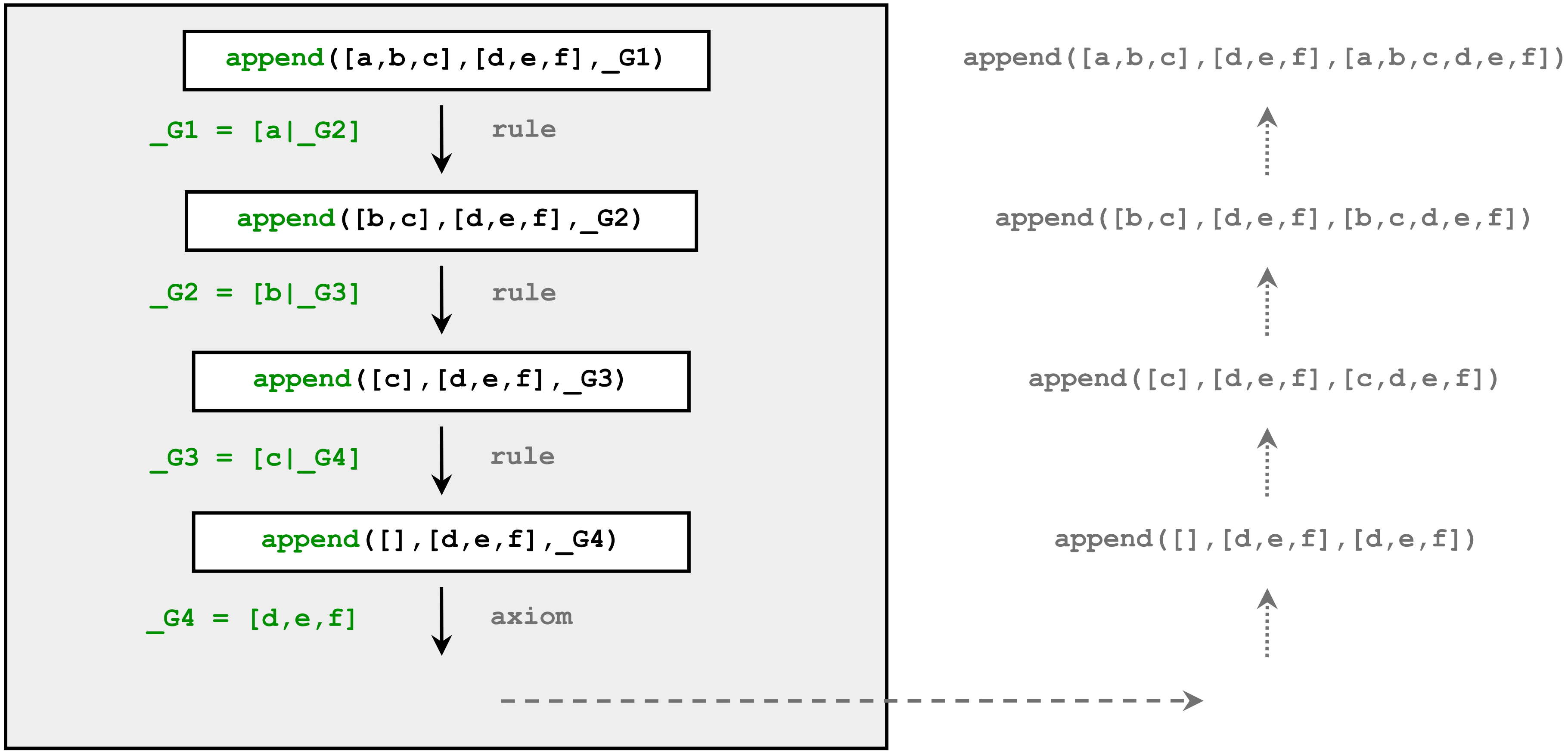
```
append([a,b,c],[d,e,f],_G1)
   _G1 = [a|_G2]     ↓   rule
append([b,c],[d,e,f],_G2)
   _G2 = [b|_G3]     ↓   rule
append([c],[d,e,f],_G3)
   _G3 = [c|_G4]     ↓   rule
append([],[d,e,f],_G4)
   _G4 = [d,e,f]     ↓   axiom
```

```
append([b,c],[d,e,f],[b,c,d,e,f])
              ↑
append([c],[d,e,f],[c,d,e,f])
              ↑
append([],[d,e,f],[d,e,f])
              ↑
```

Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3])  :-  append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```



Searching for append

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1,L2,L3).
```

```
?- append([a,b,c], [d,e,f], R)
```

```
R = [a,b,c,d,e,f]
```

```
append([a,b,c],[d,e,f],_G1)
    _G1 = [a|_G2]      ↓ rule
append([b,c],[d,e,f],_G2)
    _G2 = [b|_G3]      ↓ rule
append([c],[d,e,f],_G3)
    _G3 = [c|_G4]      ↓ rule
append([],[d,e,f],_G4)
    _G4 = [d,e,f]      ↓ axiom
```

```
append([a,b,c],[d,e,f],[a,b,c,d,e,f])
                 ↑
append([b,c],[d,e,f],[b,c,d,e,f])
                 ↑
append([c],[d,e,f],[c,d,e,f])
                 ↑
append([],[d,e,f],[d,e,f])
```

Negation in Prolog

- In Prolog we also have negation-as-failure. A negated atom **`not`** `A` is true if we are unable to prove `A`. In other words, if the search for `A` fails then we conclude `not A`.
- As Wikipedia suggests, this can be understood informally as:

```
if (not (goal p)), then (assert ¬p)
```

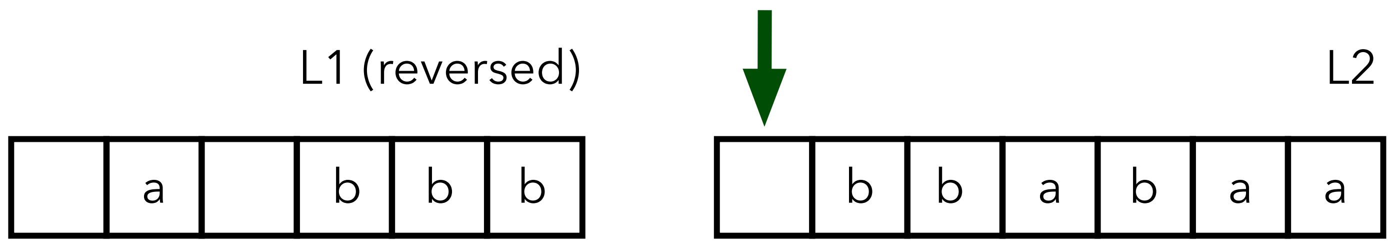
- Negation in logic programming is a difficult topic with a rich literature. We leave it here.

One last thing: Why is Prolog Turing-complete?

- It is straightforward to represent a Turing machine

  $(Q, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

- The tape is represented by two lists:



- So here `L1 = [b,b,b,B,a,B,B]` and `L2 = [B,b,b,a,b,a,a]`

- The transition function is implemented by a predicate `move` that describes how L1 and L2 should be updated. We can use this to implement a predicate `accept` to check if M accepts input w.

A holiday story

The beauty of Prolog

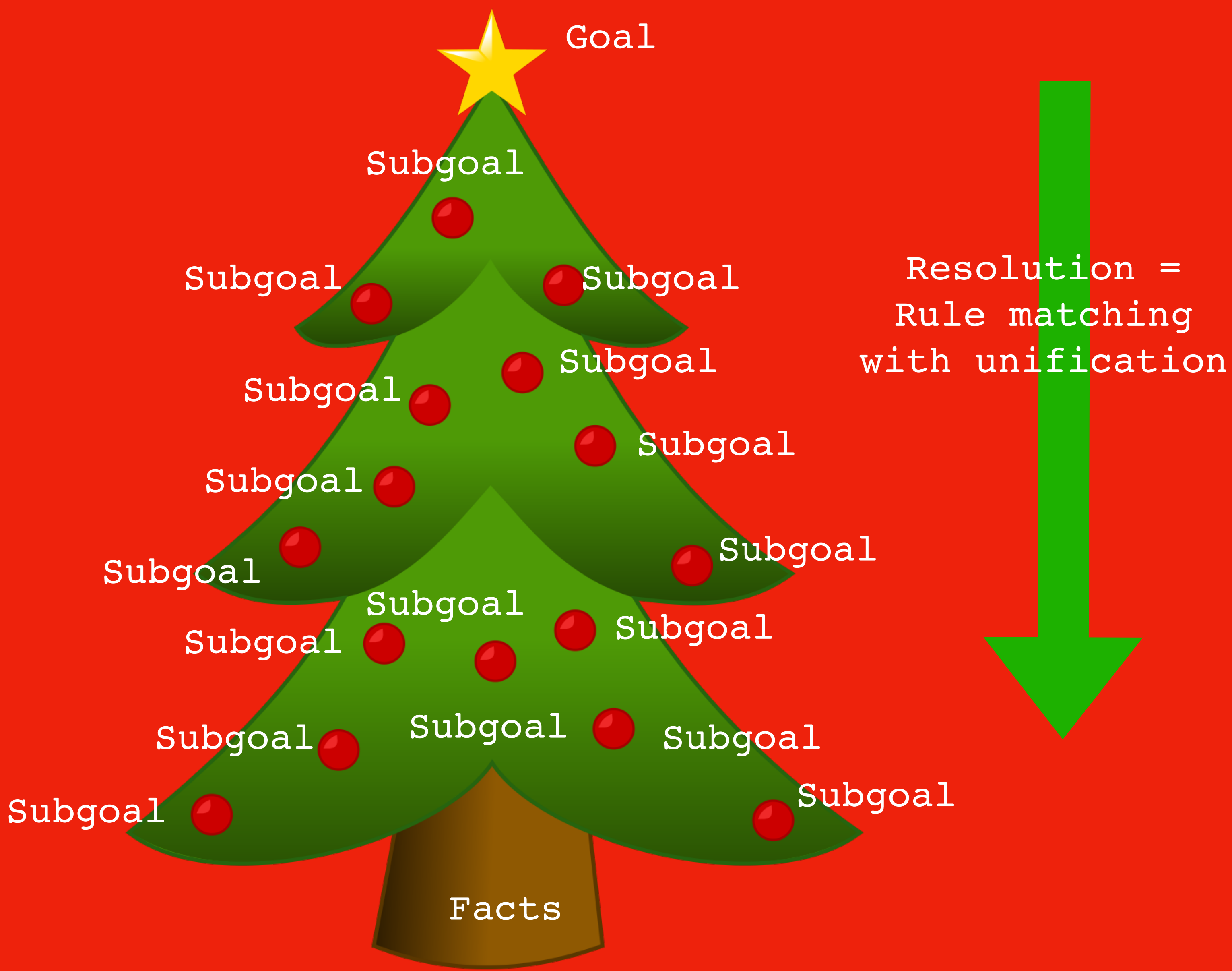
Why is Prolog such a nice language?

- Prolog is declarative! We can express a computation problem as the collection of conditions that we want from a solution.

- Prolog makes it easy to represent rule-based information. Turing machines are easy to implement. So are context-free grammars; some Prolog versions even have special grammar syntax. Transition rules from structural operational semantics can also be handled.

- Prolog is well-suited for problems in machine intelligence (that are often of a rule-based nature).

Proof search = building a tree



The soft white underbelly of Prolog

Is there anything about Prolog that may not be so nice?

- Prolog has no type system.
- Debugging a Prolog program can be tricky because of that and because you need to know the gory details of proof search.
- The complexity analysis of Prolog programs is involved.
- Efficient programming sometimes requires the cut symbol (not covered here) to optimize proof search, and that is not a very declarative approach at all.

On the other hand. . .

- **Every programming paradigm has its strengths and weaknesses.** That is why we must learn to master more than just one. Think about it – how easy is to handle, say,. . .
- A graphical interface in Prolog?
- Mutable data structures in Haskell?
- Rule-based programming in Java?
- Insertion sort in C?
- . . .or numerous other cases where programming problems and solution strategies (programming languages) do not seem to go well together.