Logic Programming

Part 2: What happens when you make a query? The semantics of logic programs.

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

Learning goals

- To understand and explain the model-theoretic semantics of logic programs, including the notions of Herbrand universe and Herbrand bases
- To understand and explain how the minimal model of a Datalog program can be computed iteratively as a fixed point
- To be able to apply goal-directed proof search to simulate a query
- To understand the negation-as-failure approach to a semantics of negation in Datalog
- To understand and be able to explain the importance of stratification for negation
- To be able to compute the precedence graph of a Datalog program in order to check if the program is stratisfiable

The three views of logic programs

A logic program can be thought of as

- A collection of formulas in first-order predicate logic.
- A relational database with rules for populating the database.
- A declaration of facts and proof rules.

This leads to *three different semantics* – that are actually equivalent but are useful in different ways.

Three different semantics

We shall study three types of semantics for logic programs:

- A model-theoretic semantics
- A fixed-point semantics
- A proof-theoretic semantics

Each of these will help us understand Datalog programs.

The model-theoretic approach

- A logic program is a collection of formulas in first-order predicate logic.

  When is a formula true?

The important notion is that of an *interpretation*.

Interpretations and facts

- In our setting, an *interpretation* $\mathcal{I}$ is a set of facts.
- We define the conditions under which a formula is true under an interpretation.
- An interpretation for which a formula is true is called a *model*.

Datalog - the abstract syntax

Here is the abstract syntax of Datalog:

$$P ::= C_1 \ldots C_n \qquad \text{Programs}$$

$$C ::= A_0 \Leftarrow A_1 \ldots A_m \qquad \text{Clauses} \quad (m \geq 0)$$

$$A ::= p(t_1, \ldots, t_k) \qquad \text{Atoms} \quad (k \geq 0)$$

$$t ::= k \mid x \qquad \text{Terms}$$

When is a clause true?

An example

Let us consider an example:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,X).
parent(margrethe,frederik).
parent(frederik,christian).
parent(frederik,isabella).
parent(frederik,vincent).
parent(frederik,josephine).
```

Jacques Herbrand

1908-1931

- Jacques Herbrand was a French mathematician.

- He finished his PhD thesis in 1929.

- In July 1931 he was mountain-climbing in the French Alps with two friends when he fell to his death in the granite mountains of Massif des Écrins.

The Herbrand universe

- The *Herbrand universe* U_P of a logic program P is the set consisting of all constants that appear anywhere in P .
- The Herbrand Universe of our example program is the set

$$\{\texttt{margrethe}, \texttt{frederik}, \texttt{christian}, \texttt{isabella}, \texttt{vincent}, \texttt{josephine}\}$$

The Herbrand base

Recall that a ground atom is an atom that contains no variables. The *Herbrand base* B_P of a logic program P is the set consisting of ground atoms that can be built over U_P using the predicates in P using the correct arity (number of arguments).

The Herbrand base of our example program is the set


```
{parent(margrethe,margrethe),parent(margrethe,frederik), parent(margrethe,christian),parent(margrethe,isabella),
parent(margrethe,vincent),parent(margrethe,josephine),parent(frederik,margrethe),parent(frederik,frederik),parent(fred
erik,christian),parent(frederik,isabella),parent(frederik,vincent),parent(frederik,josephine),parent(christian,margret
he),parent(christian,frederik),parent(christian,christian),parent(christian,isabella),parent(christian,vincent),parent
(christian,josephine),parent(isabella,margrethe),parent(isabella,frederik),parent(isabella,isabella),parent(isabella,c
hristian),parent(isabella,vincent),parent(isabella,josephine),parent(vincent,margrethe),parent(vincent,frederik),paren
t(vincent,christian),parent(vincent,isabella),parent(vincent,vincent),parent(vincent,josephine),parent(josephine,margr
ethe),parent(josephine,frederik),parent(josephine,christian),parent(josephine,isabella),parent(josephine,vincent),pare
nt(josephine,josephine),grandparent(margrethe,margrethe),grandparent(margrethe,frederik),
grandparent(margrethe,christian),grandparent(margrethe,isabella),
grandparent(margrethe,vincent),grandparent(margrethe,josephine),grandparent(frederik,margrethe),grandparent(frederik,f
rederik),grandparent(frederik,christian),grandparent(frederik,isabella),grandparent(frederik,vincent),grandparent(fred
erik,josephine),grandparent(christian,margrethe),grandparent(christian,frederik),grandparent(christian,christian),gran
dparent(christian,isabella),grandparent(christian,vincent),grandparent(christian,josephine),grandparent(isabella,margr
ethe),grandparent(isabella,frederik),grandparent(isabella,isabella),grandparent(isabella,christian),grandparent(isabel
la,vincent),grandparent(isabella,josephine),grandparent(vincent,margrethe),grandparent(vincent,frederik),grandparent(v
incent,christian),grandparent(vincent,isabella),grandparent(vincent,vincent),grandparent(vincent,josephine),grandparen
t(josephine,margrethe),grandparent(josephine,frederik),grandparent(josephine,christian),grandparent(josephine,isabella
),grandparent(josephine,vincent),grandparent(josephine,josephine)}
```


Many of these facts are of course meaningless (and not just to royalists).

What is in the Herbrand base?

The Herbrand contains three kinds of atoms:

1. The immediate facts in the extensional database – here e.g. `parent(margrethe,frederik)`

2. The derivable facts in the intensional database – here e.g. `grandparent(margrethe,isabella)`

3. All other facts – that cannot be derived, here e.g. `parent(frederik,margrethe)`

Interpretations once again

An *interpretation* $\mathcal{I}$ is a set of atoms from the Herbrand base.

For instance,

```
{parent(margrethe,frederik),
grandparent(margrethe,isabella)}
```

is an interpretation.

Truth under an interpretation

- We now define when a clause C is true under an interpretation $\mathcal{I}$. We write $\mathcal{I} \models C$.

- We have

$$\mathcal{I} \models p(k_1, \ldots, k_n) \qquad \text{if } p(k_1, \ldots, k_n) \in \mathcal{I}$$

$$\mathcal{I} \models A_0 \Leftarrow A_1, \ldots A_n \text{ if whenever}$$
$$\mathcal{I} \models A_i \text{ for all } 1 \leq i \leq n \text{ then } \mathcal{I} \models A_0$$

The last condition is equivalent to requiring that either the head is true or every clause in the body is false.

Truth under an interpretation

Given the interpretation

$$\mathcal{I} = \{\texttt{parent(margrethe,frederik)},\ \texttt{parent(frederik,isabella)}\}$$

we have that

$$\mathcal{I} \models \texttt{parent(margrethe,frederik)}$$

but also that

$$\mathcal{I} \not\models \texttt{grandparent(X,Y)} \Leftarrow \texttt{parent(X,Z)},\ \texttt{parent(Z,X)}.$$

since $\texttt{parent(margrethe,frederik)} \in \mathcal{I}$ and

and $\texttt{parent(frederik,isabella)} \in \mathcal{I}$ but

$\texttt{grandparent(margrethe,isabella)} \notin \mathcal{I}$

Models

- An interpretation $\mathcal{I}$ is called a *model* for a clause C if $\mathcal{I} \models C'$ for every *ground instance* C' of C
- An interpretation $\mathcal{I}$ is called a model for a program $C_1 \ldots C_k$ if it is a model for every clause in the program. We use the symbol $\mathcal{M}$ for models.
- The interpretation

  ```
  {parent(margrethe,frederik),
  parent(frederik,christian),parent(frederik,isabella),parent(fr
  ederik,vincent),parent(frederik,josephine),
  grandparent(margrethe,christian),grandparent(margrethe,isabell
  a),
  grandparent(margrethe,vincent),grandparent(margrethe,josephine
  )}
  ```

  is a model of our program.

Minimal models

- A model $\mathcal{M}$ for a program P is *minimal* if for every other model $\mathcal{M}'$ of P we have that $\mathcal{M} \subseteq \mathcal{M}'$
- The model we just saw, was minimal.
- Here is a model that is *not* minimal!

```
{parent(margrethe,frederik),
parent(frederik,christian),parent(frederik,isabella),par
ent(frederik,vincent),parent(frederik,josephine),
grandparent(margrethe,christian),grandparent(margrethe,i
sabella),
grandparent(margrethe,vincent),grandparent(margrethe,jos
ephine),grandparent(isabella,isabella)}
```

Why are we interested in the minimal model?

- Think about the non-minimal model we just saw:

```
{parent(margrethe,frederik),
parent(frederik,christian),parent(frederik,isabella),parent(fred
erik,vincent),parent(frederik,josephine),
grandparent(margrethe,christian),grandparent(margrethe,isabella)
,
grandparent(margrethe,vincent),grandparent(margrethe,josephine),
grandparent(isabella,isabella)}
```

It mentions a fact that is not a consequence of the clauses in the program (i.e. is not in its intensional database). It should only matter what we can derive from our program.

The intensional database of our program is the minimal model.

Finding the minimal model

Theorem

If $\mathcal{M}$ and $\mathcal{M}'$ are models of P, then so is $\mathcal{M} \cap \mathcal{M}'$

Proof: Exercise (think about what intersection means!)

Corollary

The intersection of all models of P is also a model of P

– so this is the minimal model of P

Finding the minimal model

There is a way to compute the minimal model for a program.

The algorithm relies on the fact that we can also think of a logic program as the recursive definition of a family of relations.

We define a function T_P that, given a program P and an interpretation $\mathcal{I}$ returns the set of facts that can be found from it by using any one of the clauses *exactly once*.

Finding the minimal model

The function T_P is a continuous function over the complete partial order of all interpretations ordered under set inclusion. This means that it has a least fixed point, and this fixed point is what Datalog computes. Here is how:

We start with the empty interpretation $\emptyset$ and compute the sequence

$$\emptyset, T_P(\emptyset), T_P(T_P(\emptyset)), \ldots, T_P^k(\emptyset), \ldots$$

We have that this is an increasing sequence:

$$\emptyset \subseteq T_P(\emptyset) \subseteq T_P(T_P(\emptyset)) \subseteq \ldots \subseteq T_P^k(\emptyset) \subseteq \ldots$$

Because there are only finitely many atoms in the Herbrand universe, the sequence is finite, so the algorithm must terminate. The last element of the sequence is the union of all the previous sets in the sequence.

Computing the minimal model

Datalog program

```
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).


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

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

Iteration	Facts added

Iteration 1	`edge(a, b).`
	`edge(b, c).`
	`edge(c, d).`
	`edge(d, e).`

Iteration 2	`path(a, b).`
	`path(b, c).`
	`path(c, d).`
	`path(d, e).`

Iteration 3	`path(a, c).`
	`path(b, d).`
	`path(c, e).`

Iteration 4	`path(a, d).` `path(b, e).`
Iteration 5	`path(a, e).`

Note: To make the table more readable, we only include the new facts that are added in each iteration.

Finding the minimal model

Theorem

The least fixed point of the function T_P is the minimal model.

Proof idea: In iteration k we have found all the facts that can be derived using k applications of some clause. Every fact in a model must be derivable using some number of applications of the clauses.

The proof-theoretic semantics

- Let us return to the program

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

- What if we are only interested in a simple query, e.g. `path(a,c)` ?
- We do not want to find all paths in our graph; this would be what the fixed-point computation would give us.

Proof search

- The proof-theoretic semantics gives us an algorithm for satisfying a **goal** (aka a query)
- A goal is an atom. It can be ground or contain free variables.
- In the latter case, we want to find the values of the free variables.
- Our aim is to construct *a proof tree that derives the goal*.

Building a proof tree for a goal

```
path(a,d).
```

Datalog program

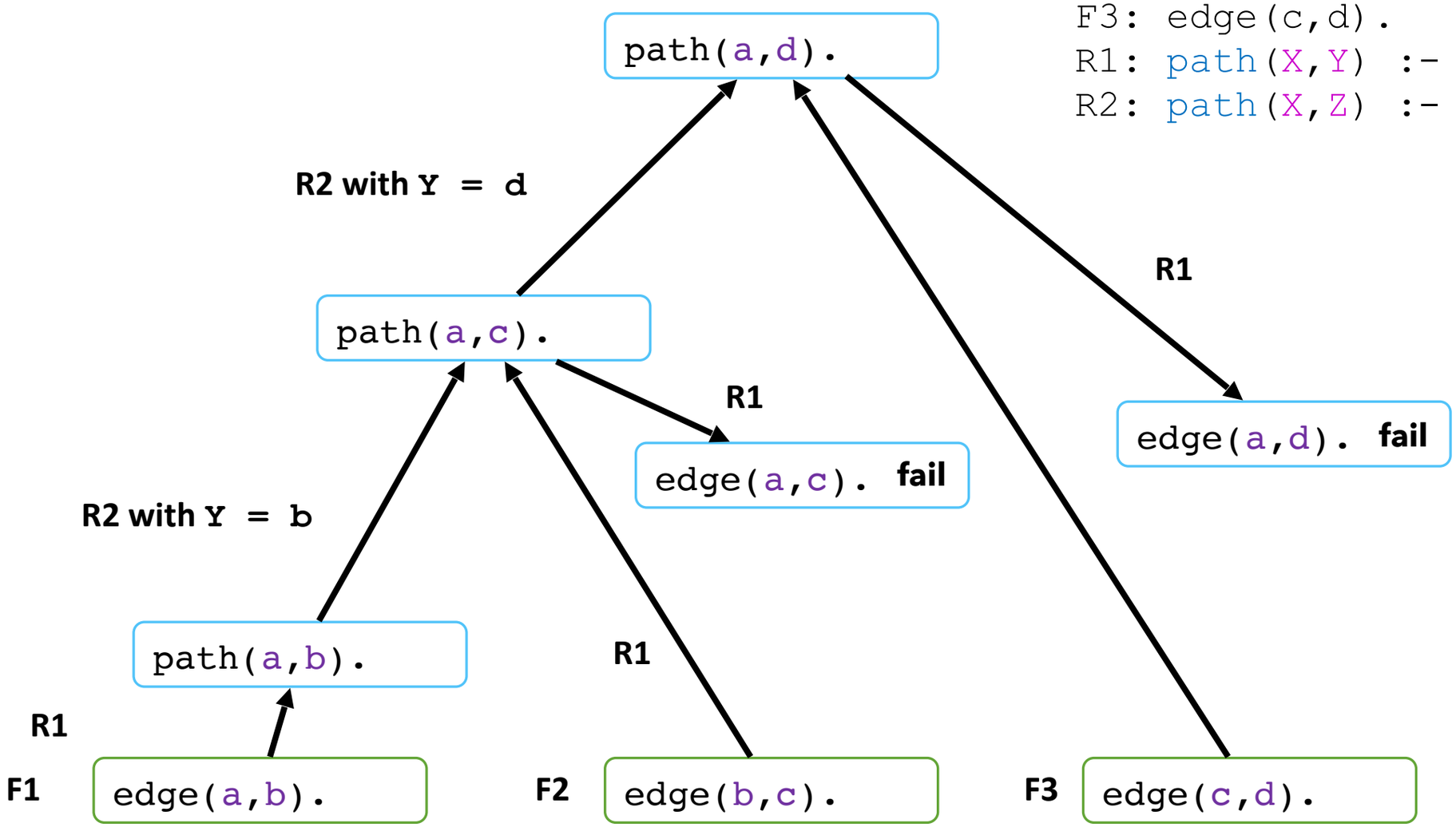
```
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).


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

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

Building a proof tree for a goal

```
F1: edge(a,b).
F2: edge(b,c).
F3: edge(c,d).
R1: path(X,Y) :- edge(X,Y).
R2: path(X,Z) :- path(X,Y),edge(Y,Z).
```



Negation

- In logic, we also have negation. So of course we would like to have that in logic programming as well.
- Example: Suppose we want to express that two vertices in a graph are not connected by a path. We would write

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), edge(Z,Y).
unconnected(X,Y) :- vertex(X), vertex(Y), not path(X,Y).
```

How should we interpret negation?

Negation-as-failure

- Only atoms can be negated. We introduce new formation rules so that we now distinguish between *positive* and *negative* atoms

$$A ::= p(t) \mid \mathbf{not}\ p(t)$$

- Negative atoms *cannot* appear in the head of a rule, only in the body.

- We also have to amend the semantics. For negative atoms we define that

$$\mathcal{I} \models \mathbf{not}\ p(t) \quad \text{if } p(t) \notin \mathcal{I}$$

Circularity rears its ugly head

- What should we think of the following bizarre program?

```
P(x) :- not Q(x).
Q(x) :- not P(x).
```

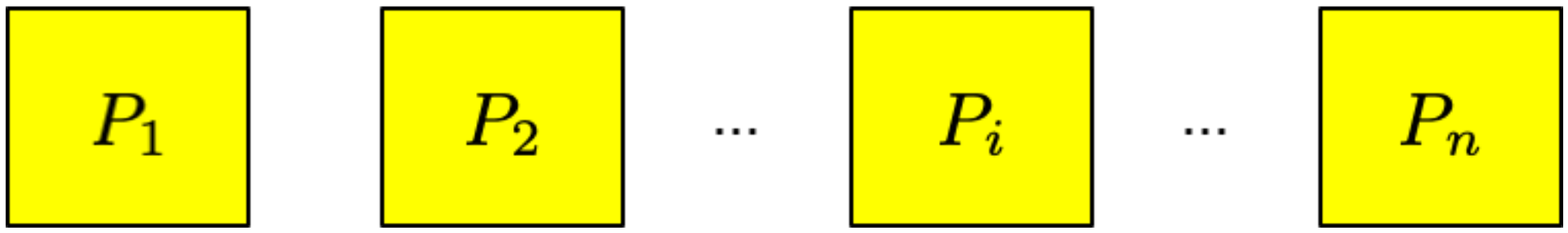
- Suppose the Herbrand universe contains one term, 17.
- Because we use negation-as-failure we get *two models* of our program:

$$\mathcal{M}_1 = \{P(17)\} \text{ and } \mathcal{M}_2 = \{Q(17)\}$$

- They are incomparable. So neither of them is minimal!

Stratification

- The solution is to disallow circularities of this kind. We can achieve this by requiring the predicates of a Datalog program to be ordered in *strata* (layers). Each stratum gives rise to a sub-program P_i that contains the clauses defining the predicates in stratum i .



- We then require that for every $i \geq 1$, the clauses for the predicates in stratum P_i can only mention negative atoms from predicates that were defined in earlier strata.

Stratification

Definition A *stratification* of Datalog program is a partition of its predicates such that

1. If there is a rule $A(\cdots) \Leftarrow \cdots B(\cdots) \cdots$ in the program where the predicate A is in stratum i and the predicate B is in stratum j then $i \geq j$.

2. If there is a rule $A(\cdots) \Leftarrow \cdots \mathbf{not}\ B(\cdots) \cdots$ in the program where the predicate A is in stratum i and the predicate B is in stratum j then $i > j$.

- In this way, whenever a predicate A depends on another predicate B, we know that the facts of B have already been computed.

Stratification

- Not every Datalog program can be stratified. The counterexample is our program from before:

```
P(x) :- not Q(x).
Q(x) :- not P(x).
```

- A stratification would require that `P` appeared in an earlier stratum than `Q` but also that `Q` appeared in an earlier stratum than `P` !!!

Safety wrt. negation

We also need to require that variables in a negated atom also appear in some positive atom in the same rule. Otherwise, we could not compute the values of such variables.

For example,

```
unconnected(X,Y) :- not path(X,Y).
```

is not safe. But

```
unconnected(X,Y) :- vertex(X), vertex(Y), not path(X,Y).
```

is safe, since we can find the values of `X` and `Y` before checking that `path(X,Y)` does not hold.

Back to the example

- Before, we saw the program

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Y), edge(Y,Z).
unconnected(X,Y) :- vertex(X), vertex(Y), not path(X,Y).
```

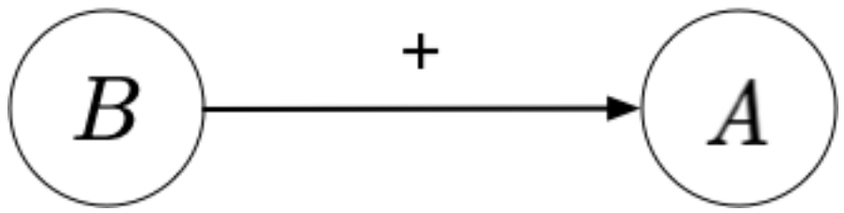
It has three predicates: `path`, `edge`, `vertex` and `unconnected`. We can stratify them as follows:

$$P_0 = \{\texttt{path}, \texttt{edge}, \texttt{vertex}\}$$
$$P_1 = \{\texttt{unconnected}\}$$

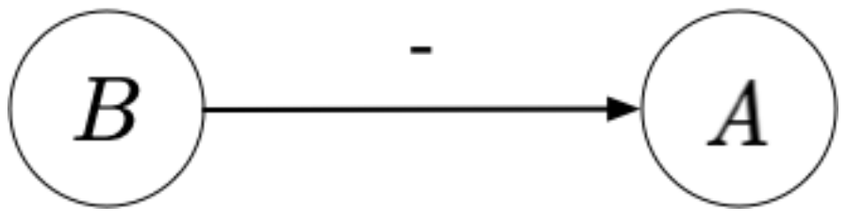
How can we find a stratification?

We define a *precedence graph*. This is a directed graph that shows how the predicates depend on each other. The vertices of the graph are the predicates, and the edges are defined as follows:

If there is a rule $A(\cdots) \Leftarrow \cdots B(\cdots) \cdots$ there is an edge



If there is a rule $A(\cdots) \Leftarrow \cdots \mathbf{not}\ B(\cdots) \cdots$ there is an edge



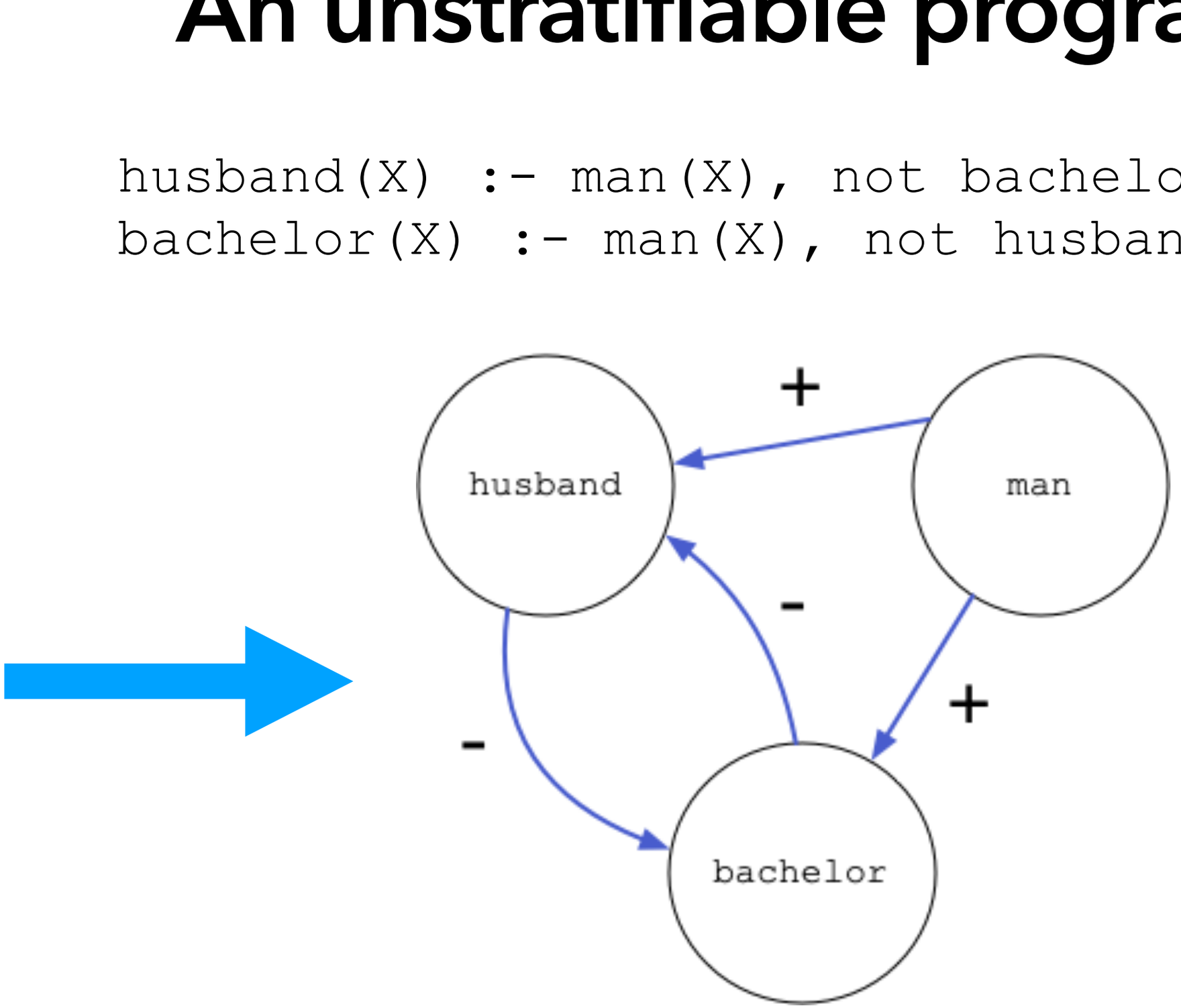
How can we find a stratification?

If the dependency graph contains a cycle with a negative edge, then we know that there is a predicate that depends on its own negation: There is a contradicting circularity and no stratification is possible.

Otherwise, every strongly connected component in the graph is a stratum. We can then order the strata by using a topological sort algorithm.

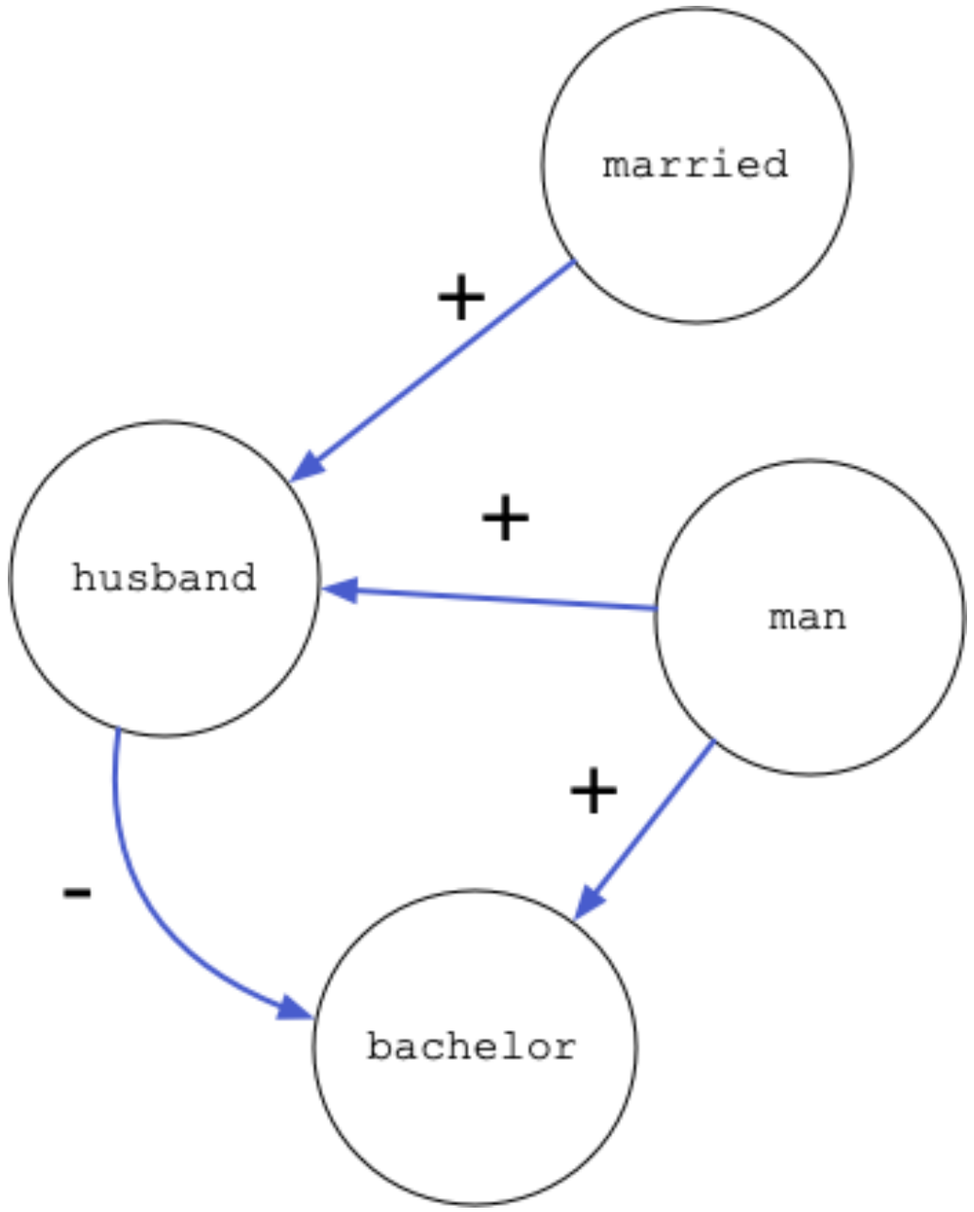
An unstratifiable program

```
husband(X) :- man(X), not bachelor(X).
bachelor(X) :- man(X), not husband(X).
```



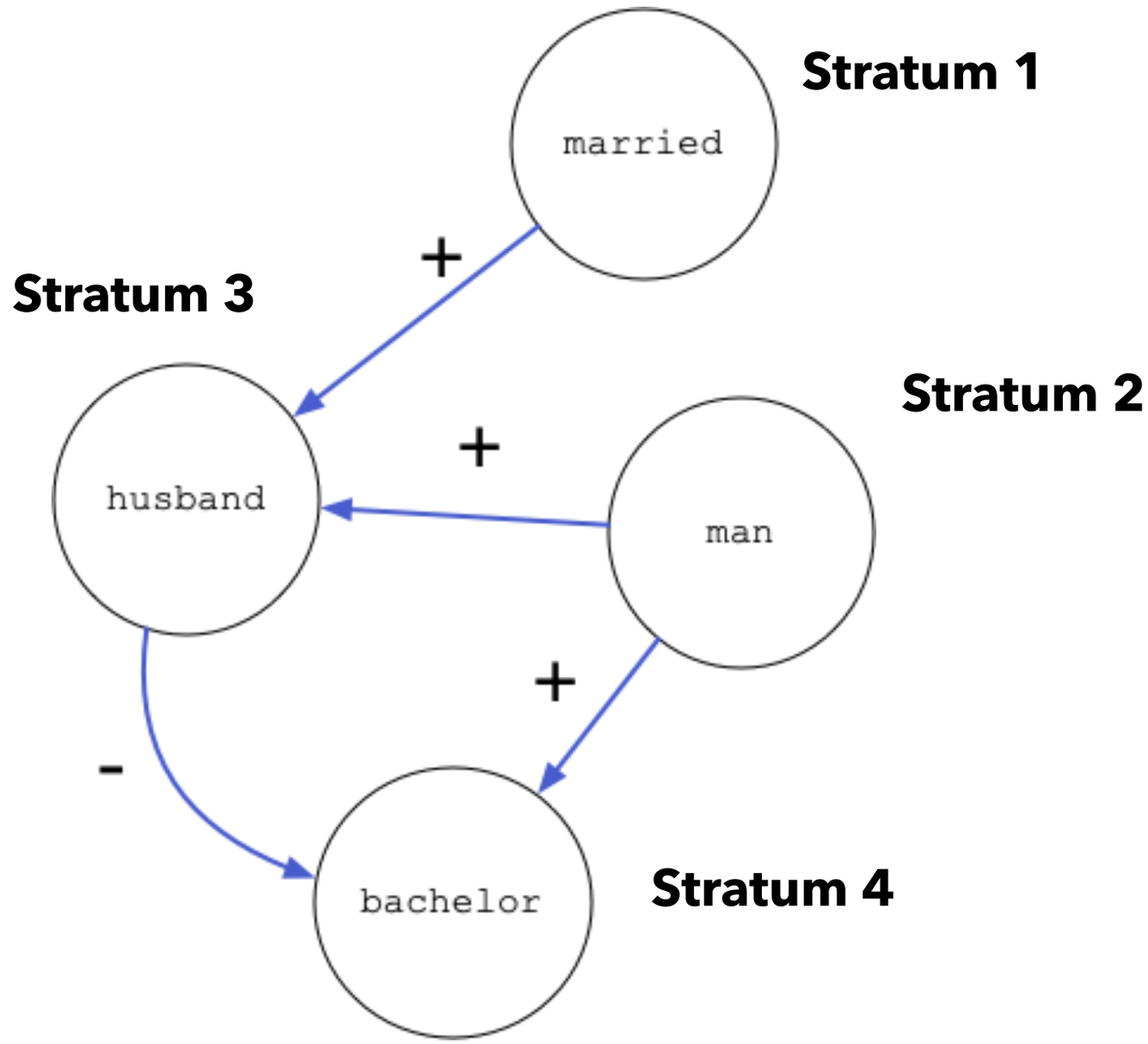
A stratifiable program

```
husband(X) :- man(X), married(X).
bachelor(X) :- man(X), not husband(X).
```



A stratifiable program

```
husband(X) :- man(X), married(X).
bachelor(X) :- man(X), not husband(X).
```



Evaluating a Datalog program

Given a program P

1. Compute the stratification $P_1, \ldots, P_n$ if possible
2. If the program is not stratifiable, ***complain!!***
3. Otherwise evaluate the subprograms $P_1, \ldots, P_n$ in sequence by computing their minimal models as described earlier. The IDB of $P_{i-1}$is the EDB of P_i ; this is the union of the minimal models of the previous subprograms.

The three semantics of logic programs

We have seen

The model-theoretic semantics.

- Herbrand universe and base, interpretations, models, and minimal models.

The fixed-point semantics.

- The immediate consequence operator and bottom-up evaluation.

The proof-theoretic semantics.

- Proof trees and top-down evaluation.

These three semantics are equivalent (we do not show this result here)

On the road to Prolog

In Datalog, everything works out fine, since the minimal model of a safe Datalog program is finite.

Next time, we turn our attention to Prolog, and here minimal models can be *infinite*. When we want to reason about the behaviour of Prolog programs, it is more useful to think of the semantics of Prolog as that of goal-directed proof search.

HORROR

Story

Terminology!

There are no methods or functions in Datalog or Prolog!

The terminology used in logic programming is different, because the paradigm is very different.

How should one read a clause?

It is tempting to read a clause such as

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

as "first, we call path, and then we call edge..."

but please do not. There are no "calls" in logic programming. Datalog and Prolog are declarative languages.

Think of the clause as defining a condition that must be satisfied: There is a path from X to Z if there is a path from X to some Y and an edge from that Y to Z.