Lectures 13 and 14 (Week 8)
Getting Started Reasoning with FOL

COMP24412: Symbolic AI

Giles Reger

March 2020

This Week

The Datalog fragment of first-order logic and efficient reasoning

Prolog as a fragment of first-order logic and how its reasoning works

Introducing Resolution for general reasoning in first-order logic

Aim and Learning Outcomes

The aim of these two lectures is to:

Introduce the main reasoning approaches for reasoning in first-order logic and the Datalog and Prolog fragments

Learning Outcomes

By the end you will be able to:

1. Describe the syntax of Datalog and Prolog knowledge bases
2. Explain the differences between first-order logic, Datalog and Prolog
3. Apply the forward and backward chaining algorithms
4. Explain and apply the resolution rule

These slides contain detailed algorithms. You do not need to recall these or apply them exactly (e.g. step-by-step) but you do need to be able to apply them generally (e.g. compute closure by forward chaining)

Reminder: Decidablility

Many of you will have already met the concept of decidability.

Decidability

A yes/no problem is decidable if there exists an algorithm that given any instance of that problem will give the correct answer.

Many problems in Computer Science are decidable:

- Searching a list for an item
- Finding the optimal solution to the Knapsack Problem
- Colouring a graph such that no two nodes have the same colour

Some (well-known) problems are not:

- The Halting Problem (does a program necessarily halt)
- Deciding whether a set of $n \times n$ matrices containing integers can be multiplied to get the zero matrix
- Computing the Kolmogorov complexity of an object

First-Order Logic is Undecidable

We can reduce the Halting problem to the problem of checking consistency of first-order logic e.g. if we can solve the second we can solve the first. But we know we cannot solve the Halting problem, ergo we cannot solve FOL consistency.

The details aren't important but in essence we can encode the transition function of a Turing machine in FOL and then add the conjecture that for all inputs there exists an accepting state. We will see next week that something of the form $\forall x \exists y$ implies the existence of a function from x things to y things. Given the axioms, this function would be one that solve the halting problem.

So what do I need to know? That FOL is undecidable in general.

Fragments of First-Order Logic are Decidable

However, clearly propositional logic is a subset of FOL and it is decidable although NP-hard in general.

There are other fragments of FOL that are decidable e.g. there exists an algorithm that can always answer the yes/no question for consistency (the core reasoning question).

We will look at one fragment - the Datalog fragment today.

Warning: the fragment I introduce is similar to Datalog in spirit but the Datalog language contains non-logical elements. If you search for Datalog online you will find concepts inconsistent with what I am teaching here.

Other Decidable Fragments of FOL

Monadic Fragment
Every predicate has arity at most 1 and there are no function symbols.

Two-variable Fragment
The formula can be written using at most 2 variables.

Guarded Fragment
If the formula is built using $\neg$ and $\wedge$, or is of the form $\exists \overline{x}.(G[\overline{y}] \wedge \phi[\overline{z}])$ such that G is an atom and $\overline{z} \subseteq \overline{y}$. Intuitively all usage of variables are *guarded* by a something positive.

Prenex Fragments
If a function-free formula is in prenex normal form and can be written as $\exists^* \forall^*.F$ it is in the BernaysSchönfinkel fragment.

At the end of the course I will briefly discuss Description Logics and Modal logics, which are often equivalent to one of these.

Datalog Fragment: Syntax

A Formula is in the Datalog fragment if it is of the form

$$\forall X.((p_1[X_1] \wedge \ldots \wedge p_n[X_n]) \Rightarrow p_{n+1}[X_{n+1}]))$$

where $p_j[X_j]$ is a predicate symbol applied to a list of constants or variables in X_j such that $X_1 \subseteq X$ and $X_{n+1} \subseteq X_1 \cup \ldots \cup X_n$ e.g.

1. All variables are universally quantified, and
2. The right-hand-side of the implication does not use variables not used on the left-hand-side

We call the right-hand-side of the implication the head and the left-hand-side the body.

If the body is empty then we call it a fact, otherwise we call it a rule.

Note that a fact cannot contain variables.

Terminology and Conventions

There's some extra terminology and conventions I'll use here and generally

- I will use the term relation interchangeably with predicate as predicates define a relation between things
- I might use the term property to refer to a unary predicate
- I might use the term object to refer to constants when we assume the unique names assumption
- Predicates, functions, and constants will always be written with a lower-case letter
- When talking about Datalog I will often drop the explicit quantification and then write variables with capital letters e.g. X, Y

Datalog Fragment: Expressiveness

Note that we cannot use the equality symbol in Datalog formulas.

The restrictions also stop us from introducing the quality axioms as we cannot express the rule $\forall.\text{equals}(X, X)$

We're also prevented from using disjunction, functions, or existential quantification.

But we can do powerful things such as defining recursive relations

Example Datalog Knowledge Base

The Facts

comp24412 teaches Logic
comp24412 is about AI
Prolog is a programming language
comp24412 teaches Prolog
AI is cool
Yachts cost lots of money

Example Datalog Knowledge Base

The Facts

comp24412 teaches Logic
comp24412 is about AI
Prolog is a programming language
comp24412 teaches Prolog
AI is cool
Yachts cost lots of money

The Rules

If you take a course and it teaches X then you know X
If you take a course about X and X is cool then you are cool
If you know a programming language then you can program
If you can program and know logic you can get a good job
If you have a good job you get lots of money
If you have X and Y costs X then you can have Y

Example Datalog Knowledge Base

The Facts

teaches(comp24412, logic)	comp24412 teaches Prolog
comp24412 is about AI	AI is cool
Prolog is a programming language	Yachts cost lots of money

The Rules

If you take a course and it teaches X then you know X
If you take a course about X and X is cool then you are cool
If you know a programming language then you can program
If you can program and know logic you can get a good job
If you have a good job you get lots of money
If you have X and Y costs X then you can have Y

Example Datalog Knowledge Base

The Facts

teaches(comp24412, logic)	teaches(comp24412, prolog)
about(comp24412, ai)	cool(ai)
language(prolog)	costs(yacht, lotsOfMoney)

The Rules

If you take a course and it teaches X then you know X
If you take a course about X and X is cool then you are cool
If you know a programming language then you can program
If you can program and know logic you can get a good job
If you have a good job you get lots of money
If you have X and Y costs X then you can have Y

Example Datalog Knowledge Base

The Facts

teaches(comp24412, logic)	teaches(comp24412, prolog)
about(comp24412, ai)	cool(ai)
language(prolog)	costs(yacht, lotsOfMoney)

The Rules

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$

If you take a course about X and X is cool then you are cool

If you know a programming language then you can program

If you can program and know logic you can get a good job

If you have a good job you get lots of money

If you have X and Y costs X then you can have Y

Example Datalog Knowledge Base

The Facts

$\text{teaches}(\text{comp24412}, \text{logic})$ $\quad$ $\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{about}(\text{comp24412}, \text{ai})$ $\quad$ $\text{cool}(\text{ai})$
$\text{language}(\text{prolog})$ $\quad$ $\text{costs}(\text{yacht}, \text{lotsOfMoney})$

The Rules

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

Example Datalog Knowledge Base

teaches(comp24412, logic)
about(comp24412, ai)
language(prolog)

teaches(comp24412, prolog)
cool(ai)
costs(yacht, lotsOfMoney)

$$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$$
$$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$$
$$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$$
$$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$$
$$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$$
$$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$$

Interpretations of Datalog Knowledge Bases

We will assume the Unique Names Assumption and the Domain Closure Assumption even though these are not expressible in the Datalog fragment.

By making these assumptions every Datalog KB has a unique minimal model (up to renaming of domain elements).

A Datalog KB must have an interpretation as there are no negative facts or rule heads.

Note that, in general (e.g. for full FOL), we can define an interpretation by the set of all ground predicates (e.g. facts) made true by that interpretation.

The single interpretation of a Datalog KB induces a single set of facts, which we shall call the closure of the KB.

Reminder: Unification and Matching

Substitution

A substitution σ is a map (finite function) from variables to terms.

We can apply a substitution to a term or formula to replace free variables e.g. given substitution $\sigma = \{x \mapsto a, y \mapsto f(x)\}$

$$f(x)\sigma = f(a) \qquad p(x,y)\sigma = p(a, f(x)) \qquad (\forall x.p(x,y))\sigma = (\forall x.p(x, f(x)))$$

The last one is bad (why?) and is why we would normally rename the bound variable to be distinct from the domain of σ.

Unification and Matching

Two terms t_1 and t_2 unify if there exists a substitution (the unifier) σ such that $t_1\sigma = t_2\sigma$ and t_2 matches t_1 if $t_1\sigma = t_2$.

A unifier is most general if we cannot drop any information and preserve the unification/matching.

Reminder: Unification and Matching

Substitution

A substitution σ is a map (finite function) from variables to terms.

We can apply a substitution to a term or formula to replace free variables e.g. given substitution $\sigma = \{x \mapsto a, y \mapsto f(x)\}$

$f(x)\sigma = f(a)$ $\quad$ $p(x,y)\sigma = p(a, f(x))$ $\quad$ $(\forall x.p(x,y))\sigma = (\forall x.p(x, f(x)))$

The last one is bad (why?) and is why we would normally rename the bound variable to be distinct from the domain of σ.

Unification and Matching

Two terms t_1 and t_2 unify if there exists a substitution (the unifier) σ such that $t_1\sigma = t_2\sigma$ and t_2 matches t_1 if $t_1\sigma = t_2$.

A unifier is most general if we cannot drop any information and preserve the unification/matching.

Example Datalog Knowledge Base

$\text{teaches}(\text{comp24412}, \text{logic})$ $\quad$ $\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{about}(\text{comp24412}, \text{ai})$ $\quad$ $\text{cool}(\text{ai})$
$\text{language}(\text{prolog})$ $\quad$ $\text{costs}(\text{yacht}, \text{lotsOfMoney})$

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

Example Datalog Knowledge Base

$\text{teaches}(\text{comp24412}, \text{logic})$ $\quad$ $\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{about}(\text{comp24412}, \text{ai})$ $\quad$ $\text{cool}(\text{ai})$
$\text{language}(\text{prolog})$ $\quad$ $\text{costs}(\text{yacht}, \text{lotsOfMoney})$

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

$\text{take}(\text{you}, \text{comp24412})$

Example Datalog Knowledge Base

teaches(comp24412, logic)
about(comp24412, ai)
language(prolog)

teaches(comp24412, prolog)
cool(ai)
costs(yacht, lotsOfMoney)

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

take(you, comp24412)

know(you, Logic) know(you, Prolog)

Example Datalog Knowledge Base

teaches(comp24412, logic) teaches(comp24412, prolog)
about(comp24412, ai) cool(ai)
language(prolog) costs(yacht, lotsOfMoney)

$$take(U, C), teaches(C, X) \Rightarrow know(U, X)$$
$$take(U, C), about(C, X), cool(X) \Rightarrow cool(U)$$
$$know(U, X), language(X) \Rightarrow canProgram(U)$$
$$canProgram(U) \wedge know(U, logic) \Rightarrow hasGoodJob(U)$$
$$hasGoodJob(U) \Rightarrow has(U, lotsOfMoney)$$
$$has(U, X), costs(Y, X) \Rightarrow has(U, Y)$$

take(you, comp24412)

know(you, Logic) know(you, Prolog) cool(you)

Example Datalog Knowledge Base

teaches(comp24412, logic)
about(comp24412, ai)
language(prolog)

teaches(comp24412, prolog)
cool(ai)
costs(yacht, lotsOfMoney)

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

take(you, comp24412)

know(you, Logic) know(you, Prolog) cool(you) canProgram(you)

Example Datalog Knowledge Base

teaches(comp24412, logic)
about(comp24412, ai)
language(prolog)

teaches(comp24412, prolog)
cool(ai)
costs(yacht, lotsOfMoney)

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

take(you, comp24412)

know(you, Logic) know(you, Prolog) cool(you) canProgram(you)
hasGoodJob(you)

Example Datalog Knowledge Base

teaches(comp24412, logic)
about(comp24412, ai)
language(prolog)

teaches(comp24412, prolog)
cool(ai)
costs(yacht, lotsOfMoney)

$$take(U, C), teaches(C, X) \Rightarrow know(U, X)$$
$$take(U, C), about(C, X), cool(X) \Rightarrow cool(U)$$
$$know(U, X), language(X) \Rightarrow canProgram(U)$$
$$canProgram(U) \wedge know(U, logic) \Rightarrow hasGoodJob(U)$$
$$hasGoodJob(U) \Rightarrow has(U, lotsOfMoney)$$
$$has(U, X), costs(Y, X) \Rightarrow has(U, Y)$$

take(you, comp24412)

know(you, Logic) know(you, Prolog) cool(you) canProgram(you)

hasGoodJob(you) has(you, LotsOfMoney)

Example Datalog Knowledge Base

teaches(comp24412, logic)
about(comp24412, ai)
language(prolog)

teaches(comp24412, prolog)
cool(ai)
costs(yacht, lotsOfMoney)

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

take(you, comp24412)

know(you, Logic) know(you, Prolog) cool(you) canProgram(you)
hasGoodJob(you) has(you, LotsOfMoney) has(you, Yacht)

Example Datalog Knowledge Base

teaches(comp24412, logic)
about(comp24412, ai)
language(prolog)

teaches(comp24412, prolog)
cool(ai)
costs(yacht, lotsOfMoney)

$$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$$
$$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$$
$$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$$
$$\text{canProgram}(U) \wedge \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$$
$$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$$
$$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$$

take(you, comp24412)

know(you, Logic) know(you, Prolog) cool(you) canProgram(you)
hasGoodJob(you) has(you, LotsOfMoney) has(you, Yacht)

Consequences and Closures

A consequence of a KB is any fact entailed by that KB e.g. any fact in the closure of the KB

We can define the closure as follows.

Let our knowledge base $\mathcal{KB}$ consist of facts $\mathcal{F}_0$ and rules $\mathcal{RU}$

Define the *next* set of facts as follows

$$\mathcal{F}_i = \mathcal{F}_{i-1} \cup \left\{ (head)\sigma \mid \begin{array}{l} body \Rightarrow head \in \mathcal{RU} \\ (body)\sigma \in \mathcal{F}_{i-1} \end{array} \right\}$$

This reaches a fixed point when $\mathcal{F}_j = \mathcal{F}_{j+1}$, this is the closure.

Consequences and Closures

A consequence of a KB is any fact entailed by that KB e.g. any fact in the closure of the KB

We can define the closure as follows.

Let our knowledge base $\mathcal{KB}$ consist of facts $\mathcal{F}_0$ and rules $\mathcal{RU}$

Define the *next* set of facts as follows

$$\mathcal{F}_i = \mathcal{F}_{i-1} \cup \left\{ (head)\sigma \mid \begin{array}{l} body \Rightarrow head \in \mathcal{RU} \\ (body)\sigma \in \mathcal{F}_{i-1} \end{array} \right\}$$

This reaches a fixed point when $\mathcal{F}_j = \mathcal{F}_{j+1}$, this is the closure.

How do we find σ? How do we compute $\mathcal{F}_{i+1}$ efficiently?

Computing Matching Substitutions

Match two facts given an existing substitution

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**
- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** $length(args_1)$ **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

If names are different, no match. For each parameter of f_1, if it is an unseen variable then extend σ, otherwise check that things are consistent.

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** *name*$_1$ *and name*$_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*($args_1$) **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ giles$\}$)

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*($args_1$) **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto \text{giles}\}$)

- $f_1 = \text{parent}(X, Y)$
- $f_2 = \text{parent}(\text{giles}, \text{mark})$
- $\sigma = \{X \mapsto \text{giles}\}$

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** *$name_1$ and $name_2$ are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*$(args_1)$ **do**
  - **if** *$args_1[i]$ is a variable and $args_1[i] \notin \sigma$* **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto \text{giles}\}$)

- $f_1 = \text{parent}(X, Y)$
- $f_2 = \text{parent}(\text{giles}, \text{mark})$
- $\sigma = \{X \mapsto \text{giles}\}$

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** $length(args_1)$ **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ giles$\}$)

- $f_1 =$ parent(X, Y)
- $f_2 =$ parent(giles, mark)
- $\sigma = \{X \mapsto$ giles$\}$
- $args_1[0] = X$
- $args_2[0] =$ giles

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** $length(args_1)$ **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto \text{giles}\}$)

- $f_1 = \text{parent}(X, Y)$
- $f_2 = \text{parent}(\text{giles}, \text{mark})$
- $\sigma = \{X \mapsto \text{giles}\}$
- $args_1[0] = X$
- $args_2[0] = \text{giles}$
- $(args_1[0])\sigma = \{X \mapsto \text{giles}\}(X) = \text{giles}$

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** *name*$_1$ *and* *name*$_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*($args_1$) **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ giles$\}$)

- $f_1 =$ parent(X, Y)
- $f_2 =$ parent(giles, mark)
- $\sigma = \{X \mapsto$ giles$\}$

- $args_1[1] = Y$
- $args_2[1] =$ mark

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** $length(args_1)$ **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ giles$\}$)

- $f_1 =$ parent(X, Y)
- $f_2 =$ parent(giles, mark)
- $\sigma = \{X \mapsto$ giles, $Y \mapsto$ mark$\}$
- $args_1[1] = Y$
- $args_2[1] =$ mark

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*($args_1$) **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto \text{giles}\}$)

- $f_1 = \text{parent}(X, Y)$
- $f_2 = \text{parent}(\text{giles}, \text{mark})$
- $\sigma = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** $length(args_1)$ **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ bob$\}$)

- $f_1 =$ parent(X, Y)
- $f_2 =$ parent(giles, mark)
- $\sigma = \{X \mapsto$ bob$\}$

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** *$name_1$ and $name_2$ are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*$(args_1)$ **do**
  - **if** *$args_1[i]$ is a variable and $args_1[i] \notin \sigma$* **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ bob$\}$)

- $f_1 =$ parent(X, Y)
- $f_2 =$ parent(giles, mark)
- $\sigma = \{X \mapsto$ bob$\}$

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** *name*$_1$ *and* *name*$_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*($args_1$) **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ bob$\}$)

- $f_1 =$ parent(X, Y)
- $f_2 =$ parent(giles, mark)
- $\sigma = \{X \mapsto$ bob$\}$
- $args_1[0] = X$
- $args_2[0] =$ giles

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** $name_1$ *and* $name_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** $length(args_1)$ **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto \text{bob}\}$)

- $f_1 = \text{parent}(X, Y)$
- $f_2 = \text{parent}(\text{giles}, \text{mark})$
- $\sigma = \{X \mapsto \text{bob}\}$
- $args_1[0] = X$
- $args_2[0] = \text{giles}$
- $(args_1[0])\sigma = \{X \mapsto \text{bob}\}(X) = \text{bob}$

Computing Matching Substitutions

def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma$)**:**

- **if** *name*$_1$ *and* *name*$_2$ *are different* **then return** $\perp$;
- **for** $i \leftarrow 0$ **to** *length*($args_1$) **do**
  - **if** $args_1[i]$ *is a variable and* $args_1[i] \notin \sigma$ **then**
    - $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
  - **else if** $(args_1[i])\sigma \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

match(parent(X, Y), parent(giles, mark), $\{X \mapsto$ bob$\}) = \perp$

Matching A Rule Body

We lift the matching algorithm to match a list of facts (the rule body) against a set of ground facts (the known consequences).

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches $= \{\emptyset\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $= \emptyset$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $= \emptyset$

$\sigma_1 = \emptyset$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $= \{\emptyset\}$
$f_1 = \text{parent}(X, Y)$
new $= \emptyset$
$\sigma_1 = \emptyset$
$f_2 = \text{parent}(\text{giles}, \text{mark})$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match*(body, $\mathcal{F}$)*:
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $= \{\emptyset\}$
$f_1 = \text{parent}(X, Y)$
new $= \emptyset$
$\sigma_1 = \emptyset$
$f_2 = \text{parent}(\text{giles}, \text{mark})$
$\sigma_2 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $=$

$\{ \quad \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \quad \}$

$\sigma_1 = \emptyset$

$f_2 = \text{parent}(\text{giles}, \text{mark})$

$\sigma_2 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $=$

$\{ \quad \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \quad \}$

$\sigma_1 = \emptyset$

$f_2 = \text{man}(\text{giles})$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $=$

$\{ \quad \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \quad \}$

$\sigma_1 = \emptyset$

$f_2 = \text{man}(\text{giles})$

$\sigma_2 = \perp$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $=$

$\{ \quad \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \quad \}$

$\sigma_1 = \emptyset$

$f_2 = \text{man}(\text{giles})$

$\sigma_2 = \bot$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match*(body, $\mathcal{F}$)*:
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $=$

$\{ \quad \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \quad \}$

$\sigma_1 = \emptyset$

$f_2 = \text{parent}(\text{bob}, \text{sara})$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $=$

$\{ \quad \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \quad \}$

$\sigma_1 = \emptyset$

$f_2 = \text{parent}(\text{bob}, \text{sara})$

$\sigma_2 = \{X \mapsto \text{bob}, Y \mapsto \text{sara}\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$\text{matches} = \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$

$\sigma_1 = \emptyset$

$f_2 = \text{parent}(\text{bob}, \text{sara})$

$\sigma_2 = \{X \mapsto \text{bob}, Y \mapsto \text{sara}\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$\text{matches} = \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$\sigma_1 = \emptyset$

$f_2 = \text{man}(\text{bob})$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$\text{matches} = \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$

$\sigma_1 = \emptyset$

$f_2 = \text{man}(\text{bob})$

$\sigma_2 = \bot$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $= \{\emptyset\}$

$f_1 = \text{parent}(X, Y)$

new $= \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$

$\sigma_1 = \emptyset$

$f_2 = \text{man}(\text{bob})$

$\sigma_2 = \perp$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{parent}(X, Y)$

$\text{new} = \emptyset$

$\sigma_1 =$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

new $= \emptyset$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

$\text{new} = \emptyset$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{parent}(\text{giles}, \text{mark})$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$
new $= \emptyset$
$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$
$f_2 = \text{parent}(\text{giles}, \text{mark})$
$\sigma_2 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man}(X)$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{parent}(\text{giles}, \text{mark})$

$\sigma_2 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

def match(*body*, $\mathcal{F}$)**:**
- matches $= \{\emptyset\}$
- **for** $f_1 \in$ *body* **do**
  - new $= \emptyset$
  - **for** $\sigma_1 \in$ *matches* **do**
    - **for** $f_2 \in \mathcal{F}$ **do**
      - $\sigma_2 = \text{match}(f_1, f_2, \sigma_1)$
      - **if** $\sigma_2 \neq \perp$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

matches $=$
$$\left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

new $=$
$$\left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{man(giles)}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man}(X)$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{man}(\text{giles})$

$\sigma_2 = \perp$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{man(giles)}$

$\sigma_2 = \bot$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man}(X)$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{parent}(\text{bob}, \text{sara})$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{parent}(\text{bob}, \text{sara})$

$\sigma_2 = \bot$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{parent}(\text{bob}, \text{sara})$

$\sigma_2 = \bot$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{man(bob)}$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches =
$$\left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

new =
$$\left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{man(bob)}$

$\sigma_2 = \bot$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man(X)}$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}$

$f_2 = \text{man(bob)}$

$\sigma_2 = \bot$

Matching A Rule Body

$$\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\ \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}) \end{array} \right\})$$

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

$$\text{matches} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \{X \mapsto \text{bob}, Y \mapsto \text{sara}\} \end{array} \right\}$$

$f_1 = \text{man}(X)$

$$\text{new} = \left\{ \begin{array}{l} \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \\ \end{array} \right\}$$

$\sigma_1 = \{X \mapsto \text{bob}, Y \mapsto \text{sara}\}$

We're not finished...

Matching A Rule Body

```
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

Clearly inefficient

The order in which we check elements in the body can effect the complexity as we can get a large set of initial fact on the first item and find that most are inconsistent with the next one

In reality implementations do something cleverer

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in$ match$(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences.

def forward(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$)**:**
- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **do**
  - $\mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$
  - **for** $body \Rightarrow head \in \mathcal{RU}$ **do**
    - **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**
      - **if** $(head)\sigma \notin \mathcal{F}$ **then** $new.add((head)\sigma)$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- **return** $\mathcal{F}$

Efficient Matching

Observation: The current algorithm for matching against known consequences is inefficient; it involves multiple iterations over all known consequences.

Solution 1: Use heuristics to select the order in which facts in the body are matched e.g. pick least frequently occurring name first.

Solution 2: Store known facts in a data structure that facilitates quick lookup of matching facts. Such data structures are called *term indexes* and are extremely important for efficient reasoning.

Incremental Forward Chaining

Observation: On each step the only new additions come from rules that are triggered by new facts.

Solution: Use the previous set of new facts as an initial filter to identify which rules are relevant and which further facts need to match against existing facts

Queries

Given a knowledge base we want to ask queries

These can be ground e.g. is ancestor(giles, adam) true?

Or, more interestingly, they can contain variables e.g. give me all ancestors of giles or more formally all X such that ancestor(giles, X) is true.

A query is a fact, possibly containing variables.

The Semantics of Queries

The answer to a query q of a knowledge base $\mathcal{KB}$ is the set

$$ans(q) = \{\sigma \mid \mathcal{KB} \models q\sigma\}$$

e.g. the set of all substitutions, which when applied to q produce a ground fact that is a consequence of $\mathcal{KB}$.

Equivalently we can define it in terms of the closure of the KB e.g.

$$ans(q) = \{\sigma \mid q\sigma \in \text{closure}(\mathcal{KB})$$

and use the forward-chaining algorithm.

The Semantics of Queries

The answer to a query q of a knowledge base $\mathcal{KB}$ is the set

$$ans(q) = \{\sigma \mid \mathcal{KB} \models q\sigma\}$$

e.g. the set of all substitutions, which when applied to q produce a ground fact that is a consequence of $\mathcal{KB}$.

Equivalently we can define it in terms of the closure of the KB e.g.

$$ans(q) = \{\sigma \mid q\sigma \in \text{closure}(\mathcal{KB})$$

and use the forward-chaining algorithm.

If the query has no answers then *ans* is empty. Can this happen?

The Semantics of Queries

The answer to a query q of a knowledge base $\mathcal{KB}$ is the set

$$ans(q) = \{\sigma \mid \mathcal{KB} \models q\sigma\}$$

e.g. the set of all substitutions, which when applied to q produce a ground fact that is a consequence of $\mathcal{KB}$.

Equivalently we can define it in terms of the closure of the KB e.g.

$$ans(q) = \{\sigma \mid q\sigma \in \text{closure}(\mathcal{KB})$$

and use the forward-chaining algorithm.

If the query has no answers then *ans* is empty. Can this happen?

What will happen if q is ground?

The Semantics of Queries

The answer to a query q of a knowledge base $\mathcal{KB}$ is the set

$$ans(q) = \{\sigma \mid \mathcal{KB} \models q\sigma\}$$

e.g. the set of all substitutions, which when applied to q produce a ground fact that is a consequence of $\mathcal{KB}$.

Equivalently we can define it in terms of the closure of the KB e.g.

$$ans(q) = \{\sigma \mid q\sigma \in \text{closure}(\mathcal{KB})$$

and use the forward-chaining algorithm.

If the query has no answers then *ans* is empty. Can this happen?

What will happen if q is ground? The substitution will be empty

The Semantics of Queries

The answer to a query q of a knowledge base $\mathcal{KB}$ is the set

$$ans(q) = \{\sigma \mid \mathcal{KB} \models q\sigma\}$$

e.g. the set of all substitutions, which when applied to q produce a ground fact that is a consequence of $\mathcal{KB}$.

Equivalently we can define it in terms of the closure of the KB e.g.

$$ans(q) = \{\sigma \mid q\sigma \in \text{closure}(\mathcal{KB})$$

and use the forward-chaining algorithm.

If the query has no answers then *ans* is empty. Can this happen?

What will happen if q is ground? The substitution will be empty

Will $ans(q)$ always be finite?

The Semantics of Queries

The answer to a query q of a knowledge base $\mathcal{KB}$ is the set

$$ans(q) = \{\sigma \mid \mathcal{KB} \models q\sigma\}$$

e.g. the set of all substitutions, which when applied to q produce a ground fact that is a consequence of $\mathcal{KB}$.

Equivalently we can define it in terms of the closure of the KB e.g.

$$ans(q) = \{\sigma \mid q\sigma \in \text{closure}(\mathcal{KB})$$

and use the forward-chaining algorithm.

If the query has no answers then *ans* is empty. Can this happen?

What will happen if q is ground? The substitution will be empty

Will $ans(q)$ always be finite? Yes - there are finite consequences

Query Answering

We can simply wrap-up the forward algorithm to answer queries

def query(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$, *query* q)**:**
    $\mathcal{F} = \text{forward}(\mathcal{F}_0, \mathcal{RU})$
    $ans = \emptyset$
    for $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
    return *ans*

How efficient is this?

Query Answering

We can simply wrap-up the forward algorithm to answer queries

def query(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$, *query* q)**:**

$\mathcal{F} = \text{forward}(\mathcal{F}_0, \mathcal{RU})$

$ans = \emptyset$

for $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$

return *ans*

How efficient is this?

Query Answering

We can simply wrap-up the forward algorithm to answer queries

def query(*facts* $\mathcal{F}_0$, *rules* $\mathcal{RU}$, *query* q)**:**
    $\mathcal{F} = \text{forward}(\mathcal{F}_0, \mathcal{RU})$
    $ans = \emptyset$
    for $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
    return *ans*

How efficient is this?

Dealing with Irrelevant Facts

Observation: We can derive a lot of facts that are irrelevant to the query

Solution 1: Rewrite the knowledge base to remove/reduce rules that produce irrelevant facts. Computationally expensive but may be worth it if similar queries executed often. Similar to query optimisation in database.

Solution 2: Backward Chaining. Start from the query and work backwards to see which facts support it. This is what Prolog does.

Open vs Closed World

The closed world assumption forces the single interpretation where the minimum possible is true and everything else is false.

In an open world setting that minimal truth is still true but we do not constrain the truth of anything else.

Sometimes the former can be useful, sometimes it can be overly restrictive. It is important to know which setting you are working in.

Open vs Closed World

The closed world assumption forces the single interpretation where the minimum possible is true and everything else is false.

In an open world setting that minimal truth is still true but we do not constrain the truth of anything else.

Sometimes the former can be useful, sometimes it can be overly restrictive. It is important to know which setting you are working in.

Closed-world reasoning is generally non-monotonic i.e. if you learn new facts to be true then things that were previously true may become false.

Open vs Closed World

The closed world assumption forces the single interpretation where the minimum possible is true and everything else is false.

In an open world setting that minimal truth is still true but we do not constrain the truth of anything else.

Sometimes the former can be useful, sometimes it can be overly restrictive. It is important to know which setting you are working in.

Closed-world reasoning is generally non-monotonic i.e. if you learn new facts to be true then things that were previously true may become false.

We don't need to differentiate when dealing with entailment as Datalog KBs have unique minimal models e.g. all models share a core (defined by the KB's closure), which we can compute.

Database Semantics

Datalog is equivalent to relational algebra (e.g. SQL) with recursion.

The so-called Database Semantics has the following three assumptions

- Closed World
- Domain Closure
- Unique Names

In databases we generally only check entailments. If I wanted to check consistency with a Datalog KB without the closed world assumption then I need more than forward chaining.

Extensional/Intensional Relations and Tables

A relation is extensional if it is defined by facts alone e.g. it does not appear in the head of a rule.

A relation is intensional if it is (partially) defined by rules e.g. it appears in the head of a rule.

An intensional definitions gives meaning by specifying necessary and sufficient conditions

Conversely, extensional definitions enumerate everything

If a KB is completely extensional then it is directly equivalent to a set of database tables.

Prolog Fragment: Syntax

A Formula is in the Prolog fragment if it is of the form

$$\forall X.((p_1 \wedge \ldots \wedge p_n) \Rightarrow p_{n+1}))$$

where p_j is one of

1. a predicate symbol applied to a list of terms using X
2. An equality $x = y$ for $x, y \in X$
3. A disequality $x \neq y$ for $x, y \in X$

for $j \leq n$ and only the first case for $j = n + 1$.

We extend Datalog with terms and we lift the variable restriction.

Note that this does not allow negation or disjunction.

It also does not capture predicates such as diff.

Equality in Prolog and Model Implications

We have a notion stronger than the unique names assumption in Prolog.

In Prolog we assume that all syntactically different terms are non-equal e.g. equality is given by unifiability. But in FOL (with equality) we can have $a = b$ or $f(X) = X$.

Theoretically, this is equivalent to restricting to Herbrand Interpretations where every symbols is interpreted as 'itself' e.g. with a one-to-one correspondence with the domain. The result is that we either have one model or no model. However, similar to Datalog, we must have a model as we don't have negation.

Note that we can also capture this assumption explicitly in first-order logic by modelling terms as term algebras e.g. we add a set of axioms that forces the only interpretation to be a Herbrand interpretation.

Forward Chaining and Prolog

So can we apply the forward chaining approach in Prolog?

Consider the simple Prolog KB

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

or

```
even(zero).
even(X) :- X = succ(succ(Y)), even(Y).
```

that captures all of the even natural numbers.
What if I wanted to find out if even(succ(succ(zero))) is true?

Forward Chaining and Prolog

So can we apply the forward chaining approach in Prolog?

Consider the simple Prolog KB

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

or

```
even(zero).
even(X) :- X = succ(succ(Y)), even(Y).
```

that captures all of the even natural numbers.
What if I wanted to find out if even(succ(succ(zero))) is true?
What if I wanted to find out if even(succ(zero)) is true?

Forward Chaining and Prolog

So can we apply the forward chaining approach in Prolog?

Consider the simple Prolog KB

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

or

```
even(zero).
even(X) :- X = succ(succ(Y)), even(Y).
```

that captures all of the even natural numbers.
What if I wanted to find out if even(succ(succ(zero))) is true?
What if I wanted to find out if even(succ(zero)) is true?

Whilst we can answer some queries, the closure of this KB is infinite and we will not reach a fixed-point with forward-chaining.

When do we get a finite model/closure with a Prolog KB?

Backward Chaining: General Idea

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

Backward Chaining: General Idea

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

As an example, if we consider

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

with the query even(succ(succ(zero))) we should

1. Look to see if it is already a fact - is it?
2. Look to see if it unifies with the head of any rule - does it?

Backward Chaining: General Idea

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

As an example, if we consider

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

with the query even(succ(succ(zero))) we should

1. Look to see if it is already a fact - is it?
2. Look to see if it unifies with the head of any rule - does it?

Unifying even(succ(succ(zero))) and even(X) gives $X \mapsto \text{succ}(\text{succ}(\text{zero}))$

Backward Chaining: General Idea

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

As an example, if we consider

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

with the query even(succ(succ(zero))) we should

1. Look to see if it is already a fact - is it?
2. Look to see if it unifies with the head of any rule - does it?

Unifying even(succ(succ(zero))) and even(X) gives $X \mapsto \text{succ}(\text{succ}(\text{zero}))$

We have reduced solving our query to solving the body when we subsitute for X e.g.succ(succ(zero)) = succ(succ(Y)) $\wedge$ even(Y)

Backward Chaining: General Idea

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

As an example, if we consider

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

with the query even(succ(succ(zero))) we should

1. Look to see if it is already a fact - is it?
2. Look to see if it unifies with the head of any rule - does it?

Unifying even(succ(succ(zero))) and even(X) gives $X \mapsto \text{succ}(\text{succ}(\text{zero}))$

We have reduced solving our query to solving the body when we subsitute for X e.g.$\text{succ}(\text{succ}(\text{zero})) = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)$

This reduces to even(zero) which is already a fact - success!

Backward Chaining: Failing Queries

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

Backward Chaining: Failing Queries

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

Now let's look to see what happens with

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

and the query even(succ(zero)).

Backward Chaining: Failing Queries

The general idea of backward chaining is to do subgoal reduction e.g. to reduce the goal into subgoals until we solve them.

Now let's look to see what happens with

$$\text{even}(\text{zero}) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \Rightarrow \text{even}(X)$$

and the query even(succ(zero)).

Unifying even(succ(zero)) and even(X) gives $X \mapsto \text{succ}(\text{zero})$

Our new goal is $\text{succ}(\text{zero}) = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)$

But $\text{succ}(\text{zero}) = \text{succ}(\text{succ}(Y))$ fails as they do not unify

There's nothing else to try and we conclude our query fails

Backward Chaining: Backtracking

What about if we changed our knowledge base to

$$X = \text{zero} \Rightarrow \text{even}(X) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \wedge \Rightarrow \text{even}(X)$$

and asked the query even(succ(zero)).

Backward Chaining: Backtracking

What about if we changed our knowledge base to

$$X = \text{zero} \Rightarrow \text{even}(X) \qquad (X = \text{succ}(\text{succ}(Y)) \wedge \text{even}(Y)) \wedge \Rightarrow \text{even}(X)$$

and asked the query even(succ(zero)).

We have two different rules where we unify with the head.

We need to try each and backtrack to our choice point on failure

You should be familiar with this idea from observing how Prolog worked.... if not then hopefully this gives you more insight into how Prolog works!

Backward Chaining: Backtracking

Here's an example where backtracking is necessary for success

$$\begin{aligned}&\text{parent}(\text{david}, \text{giles})\\&\text{parent}(\text{giles}, \text{mark})\\&\text{parent}(X, Y) \Rightarrow \text{ancestor}(X, Y)\\&(\text{parent}(X, Z) \wedge \text{ancestor}(Z, Y)) \Rightarrow \text{ancestor}(X, Y)\end{aligned}$$

with the query ancestor(david, mark)

If we try the first rule we will fail as parent(david, mark) is not a fact

We must backtrack and then try the subgoal parent(david, Z)

Which succeeds with $Z =$ giles so we try the subgoal ancestor(giles, mark) and succeed (using the first ancestor rule)

Backward Chaining: Multiple Answers

Now let us consider the same knowledge base

$$\begin{aligned}
&\text{parent}(\text{david}, \text{giles})\\
&\text{parent}(\text{giles}, \text{mark})\\
&\text{parent}(X, Y) \Rightarrow \text{ancestor}(X, Y)\\
&(\text{parent}(X, Z) \wedge \text{ancestor}(Z, Y)) \Rightarrow \text{ancestor}(X, Y)
\end{aligned}$$

and the query $\text{ancestor}(\text{david}, X)$

The first rule will give us an answer $X = \text{giles}$

We must then backtrack and try the second rule (following similar steps as before) to get the second answer $X = \text{mark}$

Again, hopefully this is familiar from Prolog.

Example Datalog Knowledge Base

$\text{teaches}(\text{comp24412}, \text{logic})$
$\text{about}(\text{comp24412}, \text{ai})$
$\text{language}(\text{prolog})$
$\text{take}(\text{you}, \text{comp24412})$

$\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{cool}(\text{ai})$
$\text{costs}(\text{yacht}, \text{lotsOfMoney})$

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U), \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

$\text{has}(\text{you}, \text{yacht})$

Example Datalog Knowledge Base

$\text{teaches}(\text{comp24412}, \text{logic})$
$\text{about}(\text{comp24412}, \text{ai})$
$\text{language}(\text{prolog})$
$\text{take}(\text{you}, \text{comp24412})$

$\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{cool}(\text{ai})$
$\text{costs}(\text{yacht}, \text{lotsOfMoney})$

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U), \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

$\text{has}(\text{you}, \text{yacht})$
$\text{has}(\text{you}, X)$, $\text{costs}(\text{yacht}, X)$

Example Datalog Knowledge Base

$\text{teaches}(\text{comp24412}, \text{logic})$ $\quad$ $\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{about}(\text{comp24412}, \text{ai})$ $\quad$ $\text{cool}(\text{ai})$
$\text{language}(\text{prolog})$ $\quad$ $\text{costs}(\text{yacht}, \text{lotsOfMoney})$
$\text{take}(\text{you}, \text{comp24412})$

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U), \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

$\text{has}(\text{you}, \text{yacht})$
$\text{has}(\text{you}, X)$, $\text{costs}(\text{yacht}, X)$
$\text{hasGoodJob}(\text{you})$, $\text{costs}(\text{yacht}, \text{lotsOfMoney})$

Example Datalog Knowledge Base

$\text{teaches}(\text{comp24412}, \text{logic})$ $\quad$ $\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{about}(\text{comp24412}, \text{ai})$ $\quad$ $\text{cool}(\text{ai})$
$\text{language}(\text{prolog})$ $\quad$ $\text{costs}(\text{yacht}, \text{lotsOfMoney})$
$\text{take}(\text{you}, \text{comp24412})$

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U), \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

$\text{has}(\text{you}, \text{yacht})$
$\text{has}(\text{you}, X)$, $\text{costs}(\text{yacht}, X)$
$\text{hasGoodJob}(\text{you})$, $\text{costs}(\text{yacht}, \text{lotsOfMoney})$
$\text{canProgram}(\text{you})$, $\text{know}(\text{you}, \text{logic})$, $\text{costs}(\text{yacht}, \text{lotsOfMoney})$

Example Datalog Knowledge Base

$\text{teaches}(\text{comp24412}, \text{logic})$ $\quad$ $\text{teaches}(\text{comp24412}, \text{prolog})$
$\text{about}(\text{comp24412}, \text{ai})$ $\quad$ $\text{cool}(\text{ai})$
$\text{language}(\text{prolog})$ $\quad$ $\text{costs}(\text{yacht}, \text{lotsOfMoney})$
$\text{take}(\text{you}, \text{comp24412})$

$\text{take}(U, C), \text{teaches}(C, X) \Rightarrow \text{know}(U, X)$
$\text{take}(U, C), \text{about}(C, X), \text{cool}(X) \Rightarrow \text{cool}(U)$
$\text{know}(U, X), \text{language}(X) \Rightarrow \text{canProgram}(U)$
$\text{canProgram}(U), \text{know}(U, \text{logic}) \Rightarrow \text{hasGoodJob}(U)$
$\text{hasGoodJob}(U) \Rightarrow \text{has}(U, \text{lotsOfMoney})$
$\text{has}(U, X), \text{costs}(Y, X) \Rightarrow \text{has}(U, Y)$

$\text{has}(\text{you}, \text{yacht})$
$\text{has}(\text{you}, X)$, $\text{costs}(\text{yacht}, X)$
$\text{hasGoodJob}(\text{you})$, $\text{costs}(\text{yacht}, \text{lotsOfMoney})$
$\text{canProgram}(\text{you})$, $\text{know}(\text{you}, \text{logic})$, $\text{costs}(\text{yacht}, \text{lotsOfMoney})$
$\text{know}(\text{you}, X)$, $\text{language}(X)$, $\text{know}(\text{you}, \text{logic})$, $\text{costs}(\text{yacht}, \text{lotsOfMoney})$

Backward Chaining is Depth-First Search

When we do backward chaining we are performing a depth-first search of the the set of proofs that have the query as a conclusion.

If a particular direction is infinite then we will get stuck in non-terminating behaviour.

For example, with the rule

$$(\text{ancestor}(X, Z) \wedge \text{parent}(Z, Y)) \Rightarrow \text{ancestor}(X, Y)$$

where we will repeatedly apply the same rule to the left premise.

Next week we will see a breadth-first form of proof search that avoids getting stuck in this way.

Negation as Failure

If we have a ground fact f and we fail to show that it holds by backward chaining then we can conclude its negation $\neg f$ e.g. it does not hold.

We could lift this idea to allow negation more generally e.g.

$$\textbf{not}\ \text{dangerous}(\text{mercury}) \Rightarrow \text{safeToUse}(\text{mercury})$$

says that if we cannot use the rest of our KB to show that mercury is dangerous then it is safe to use.

However, this can become unintuitive when using variables e.g. taking

$$\textbf{not}\ \text{dangerous}(X) \Rightarrow \text{safeToUse}(X)$$

if dangerous(mercury) appears in the KB then **not**dangerous(X) will always fail as there exists an X that makes the subgoal succeed.

Negation as failure effectively means

$$\forall X.((\exists X.\neg\text{dangerous}(X)) \Rightarrow \text{safeToUse}(X))$$

Negation as Failure

If we have a ground fact f and we fail to show that it holds by backward chaining then we can conclude its negation $\neg f$ e.g. it does not hold.

We could lift this idea to allow negation more generally e.g.

$$\textbf{not}\ \text{dangerous}(\text{mercury}) \Rightarrow \text{safeToUse}(\text{mercury})$$

says that if we cannot use the rest of our KB to show that mercury is dangerous then it is safe to use.

However, this can become unintuitive when using variables e.g. taking

$$\textbf{not}\ \text{dangerous}(X) \Rightarrow \text{safeToUse}(X)$$

if dangerous(mercury) appears in the KB then **not**dangerous(X) will always fail as there exists an X that makes the subgoal succeed.

Negation as failure effectively means

$$\forall X.((\exists Y.\neg\text{dangerous}(Y)) \Rightarrow \text{safeToUse}(X))$$

Generalising Modus Ponens

Both the forward and backward chaining approaches worked on rules with a single fact in the head. Logically these are called definite clauses.

Such implications are also what the well-known Modus Ponens rule works on

$$\frac{A \rightarrow B \qquad C}{B\theta} \quad \theta = \text{mgu}(A, C)$$

What do we do if we have a rule of the general form

$$(A \wedge \ldots \wedge B) \rightarrow (C \vee \ldots \vee D)$$

(note that next week we will see that all first-order formulas can be written in this form)

Reasoning with General Implications

If someone is rich then they are happy
I am rich or delusional

Therefore, ?

Reasoning with General Implications

$rich(X) \rightarrow happy(X)$
$rich(giles) \vee delusional(giles)$

Therefore, ?

Reasoning with General Implications

$rich(X) \rightarrow happy(X)$
$rich(giles) \vee delusional(giles)$

$happy(giles) \vee delusional(giles)$

Reasoning with General Implications

$rich(X) \rightarrow happy(X)$
$rich(giles) \vee delusional(giles)$

$happy(giles) \vee delusional(giles)$

This is captured by generalisation of Modus Ponens called Resolution

$$\frac{A \rightarrow B \qquad A \vee D}{B \vee D}$$

Reasoning with General Implications

$rich(X) \rightarrow happy(X)$
$rich(giles) \vee delusional(giles)$

$happy(giles) \vee delusional(giles)$

This is captured by generalisation of Modus Ponens called Resolution

$$\frac{A \rightarrow B \qquad C \vee D}{(B \vee D)\theta} \quad \theta = \text{mgu}(A, C)$$

Reasoning with General Implications

$rich(X) \rightarrow happy(X)$
$rich(giles) \vee delusional(giles)$

$happy(giles) \vee delusional(giles)$

This is captured by generalisation of Modus Ponens called Resolution

$$\frac{\neg A \vee B \qquad C \vee D}{(B \vee D)\theta} \quad \theta = \text{mgu}(A, C)$$

We resolve on A and C

Reasoning with General Implications

$rich(X) \rightarrow happy(X)$
$rich(giles) \vee delusional(giles)$

$happy(giles) \vee delusional(giles)$

This is captured by generalisation of Modus Ponens called Resolution

$$\frac{\neg A \vee B \qquad C \vee D}{(B \vee D)\theta} \quad \theta = \text{mgu}(A, C)$$

We resolve on A and C

$(\neg A \vee C)\theta$ must be valid as A and C unify

A model of the premises cannot satisfy both $\neg A\theta$ and $C\theta$

Clauses

A literal is an an atom or its negation. A clause is a disjunction of literals.

Clauses are implicitly universally quantified.

We can think of a clause as a conjunction implying a disjunction e.g.

$$(a_1 \wedge \ldots a_n) \rightarrow (b_1 \vee \ldots \vee b_m)$$

An empty clause is false.

If $m \leq 1$ then a clause is Horn - this is the goal in Prolog.

If $m = 1$ then a clause is definite - this is what we've used in KBs so far.

From now on we write t, s for terms, l for literals and C, D for clauses.

Resolution

Resolution works on clauses

$$\frac{l_1 \vee C \qquad \neg l_2 \vee D}{(C \vee D)\theta} \quad \theta = \text{mgu}(l_1, l_2)$$

For example

$$\frac{p(a,x) \vee r(x) \qquad \neg r(f(y)) \vee p(y,b)}{p(a,f(y)) \vee p(y,b)}$$

Do these two clauses resolve?

$$s(x,a,x) \vee p(x,b) \qquad \neg s(b,y,c) \vee \neg p(f(b),b)$$

Resolution

Resolution works on clauses

$$\frac{l_1 \vee C \qquad \neg l_2 \vee D}{(C \vee D)\theta} \quad \theta = \text{mgu}(l_1, l_2)$$

For example

$$\frac{p(a,x) \vee r(x) \qquad \neg r(f(y)) \vee p(y,b)}{p(a,f(y)) \vee p(y,b)}$$

Do these two clauses resolve?

$$\frac{s(x,a,x) \vee p(x,b) \qquad \neg s(b,y,c) \vee \neg p(f(b),b)}{s(f(b),a,f(b) \vee \neg s(b,y,c)}$$

Refutational Based Reasoning

We are going to look at a reasoning method that works by refutation.

Recall $\Gamma \models \phi$ if and only if $\Gamma \cup \{\neg\phi\} \models \mathit{false}$

If we want to show that ϕ is entailed by Γ we can show that $\Gamma \cup \{\neg\phi\}$ is inconsistent

Refutational Based Reasoning

We are going to look at a reasoning method that works by refutation.

Recall $\Gamma \models \phi$ if and only if $\Gamma \cup \{\neg\phi\} \models \mathit{false}$

If we want to show that ϕ is entailed by Γ we can show that $\Gamma \cup \{\neg\phi\}$ is inconsistent

This is refutational based reasoning.

We will saturate $\Gamma \cup \{\neg\phi\}$ until there is nothing left to add or we have derived *false*.

If we do not find *false* then $\Gamma \not\models \phi$.

There are some caveats we will meet later.

Resolving to false

$$\begin{array}{l} \neg rich(x) \vee happy(x) \\ rich(giles) \end{array} \models happy(giles)$$

Resolving to false

$\neg rich(x) \vee happy(x)$
$rich(giles)$
$\neg happy(giles)$

Resolving to false

$\neg rich(x) \vee happy(x)$
$rich(giles)$
$\neg happy(giles)$

Resolving to false

$\neg rich(x) \vee happy(x)$
$rich(giles)$
$\neg happy(giles)$
$\neg rich(giles)$

Resolving to false

$\neg rich(x) \vee happy(x)$

$rich(giles)$

$\neg happy(giles)$

$\neg rich(giles)$

Resolving to false

$\neg rich(x) \vee happy(x)$
$rich(giles)$
$\neg happy(giles)$
$\neg rich(giles)$
$false$

Resolving to false

$$\neg rich(x) \vee happy(x)$$
$$rich(giles)$$
$$\neg happy(giles)$$
$$\neg rich(giles)$$
$$false$$

We could have done it in the other order (picked $\neg rich(x)$ first). We'll find out later that it's better to organise proof search to avoid this redundancy.

Summary

This week we have seen

- Datalog as a syntactic fragment
- Forward chaining for answering queries in Datalog
- Prolog as a syntactic fragement
- Backward chaining for answering queries in Prolog
- The Resolution Rule

Next week:

- How do we get general first-order formulas into clausal form?
- How do we use resolution within proof search effectively?
  - Forward chaining is unguided BFS
  - Backward chaining is partially guided DFS
  - Next week we will see The Given Clause Algorithm that parametrises search by a priority function e.g. a kind of best-first search